

# ARM Cortex-M Embedded Design from 0 to 1

```

MessageStateType QueryMessageStatus(u32 u32Token_)
{
    MessageStateType eStatus = NOT_FOUND;
    MessageStateType* pListParser = &Msg_asStatusQueue[0];

    /* Brute force search for the token - the queue will never be large enough
    on this system to require a more intelligent search algorithm */
    while( (pListParser->u32Token != u32Token_) &&
        (pListParser != &Msg_asStatusQueue[U8_STATUS_QUEUE_SIZE]) )
    {
        pListParser++;
    }

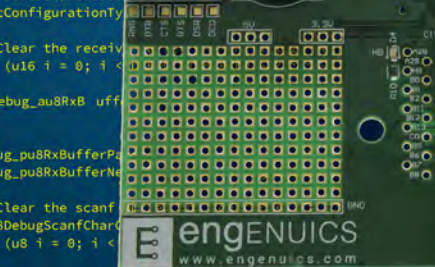
    /* If the token was found pListParser is pointing at it */
    if(pListParser != &Msg_asStatusQueue[U8_STATUS_QUEUE_SIZE])
    {
        /* Save the status
        eStatus = pListParser->eStatus;

        /* Release the slot
        if( (eStatus == AD) || (eStatus == AL) )
        {
            pListParser->u32Token = 0;
            pListParser->eStatus = NOT_FOUND;
            pListParser->u8Status = 0;
        }

        /* Return what was found
        return(eStatus);
    }

    /* end QueryMessageStatus
}

```



```

void DebugInitialize()
{
    u8 au8FirmwareVersion;
    UARTConfiguration_t uUARTConfig;

    /* Clear the receive buffer */
    for (u16 i = 0; i < UART_RX_BUFFER_SIZE; i++)
    {
        Debug_au8RxBuffer[i] = 0;
    }

    Debug_pu8RxBufferPtr = &Debug_au8RxBuffer[0];
    Debug_pu8RxBufferNextChar = &Debug_au8RxBuffer[0];

    /* Clear the scan buffer */
    G_u8DebugScanChar = 0;
    for (u8 i = 0; i < G_u8DebugScanBufferSize; i++)
    {
        G_au8DebugScanBuffer[i] = 0;
    }

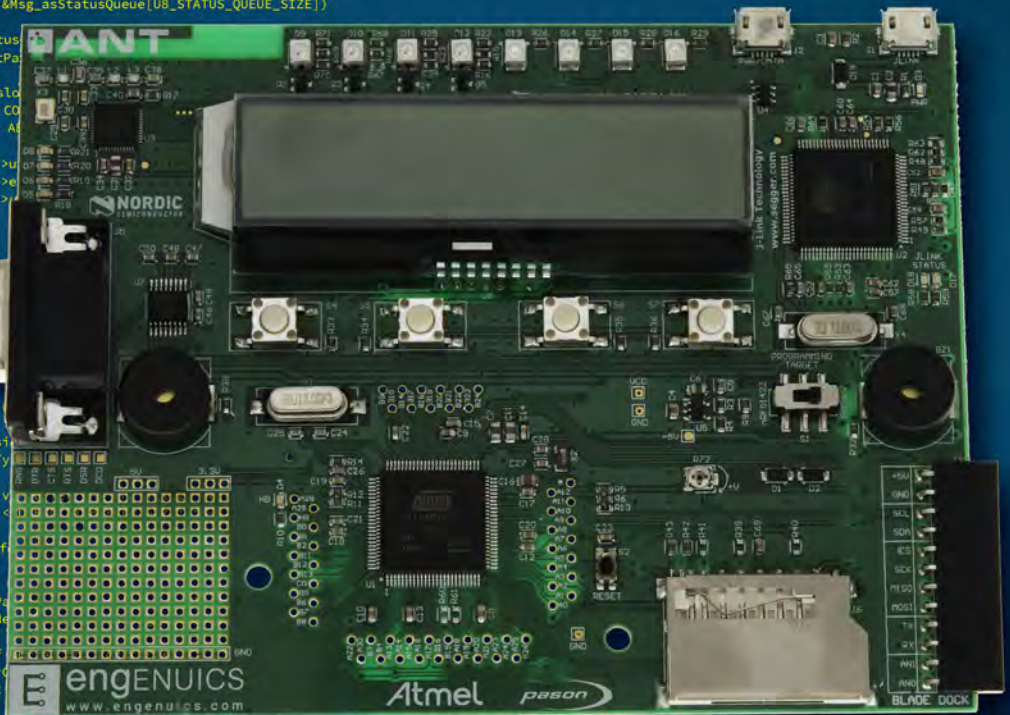
    /* Request the UART resource to be used for the Debug application
    sUARTConfig.uartPeripheral = DEBUG_UART;
    sUARTConfig.pu8RxBufferAddress = &Debug_au8RxBuffer[0];
    sUARTConfig.pu8RxNextByte = &Debug_pu8RxBufferNextChar;
    sUARTConfig.u16RxBufferSize = DEBUG_RX_BUFFER_SIZE;
    sUARTConfig.fnRxCallback = DebugRxCallback;

    Debug_UART = UARTRequest(&sUARTConfig);

    /* If the UART resource is not available, return an error state if the UART request failed */
}
    
```



## EMBEDDED IN EMBEDDED



# Jason Long



ektor

LEARN > DESIGN > SHARE

---

# Embedded in Embedded

ARM Cortex-M embedded design from 0 to 1



Jason Long



an Elektor Publication

LEARN > DESIGN > SHARE

● This is an Elektor Publication. Elektor is the media brand of  
Elektor International Media B.V.

78 York Street

London W1H 1DP, UK

Phone: (+44) (0)20 7692 8344

© Elektor International Media BV 2018

First published in the United Kingdom 2018

179014-1/EN

● All rights reserved. No part of this book may be reproduced in any material form, including photocopying, or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication, without the written permission of the copyright holder except in accordance with the provisions of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's written permission to reproduce any part of this publication should be addressed to the publishers. The publishers have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, and hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident or any other cause.

● British Library Cataloguing in Publication Data  
Catalogue record for this book is available from the British Library

● **ISBN 978-1-907920-73-8**

Prepress production: DMC | [daverid.com](http://daverid.com)

Printed in the Netherlands by Wilco



Elektor is part of EIM, the world's leading source of essential technical information and electronics products for pro engineers, electronics designers, and the companies seeking to engage them. Each day, our international team develops and delivers high-quality content - via a variety of media channels (e.g., magazines, video, digital media, and social media) in several languages - relating to electronics design and DIY electronics. [www.elektor.com](http://www.elektor.com)

LEARN > DESIGN > SHARE

## **Dedication**

To Adym, Ben, and Eryn: those who matter most.

To Amy Strutt, Barry Moore, and Hanson Zheng who have shown unconditional confidence and belief in me. Also to Dr. Mike Smith at the Schulich School of Engineering for all the fundamental knowledge you provided, and the two decades of continued support.

My reviewers and friends, Jon, Vipin and Pouyan. You are amazing people.

To the disproportionately high number of influential people in my life named Steve that the Universe seems to produce at just the right time and in just the right way.

Finally, a big thank you to Ferdinand and Don at Elektor for their patience with my “off by an order of magnitude” estimates, and my editor Dave for his tireless efforts in somehow deciphering my napkin sketches and putting this all together.



## Acknowledgements

The following material is reproduced in this book with the kind permission of the respective copyright holders and may not be reprinted, or reproduced in any way without their prior consent.

Figure 1-1: Weekly World News

Figure 1-2: Stockphoto

Figure 1-5: [www.microchip.com](http://www.microchip.com)

Figure 1-6: <http://learnprotocol.blogspot.ca/2014/07/learn-ic-packages-01.html>

Figure 1-7: ARM Cortex Microcontrollers

Figure 1-9, Figure 3-1, Figures 4-12 to 4-14, Figures 5-5 to 5-6 and 5-9, Figures 6-7 to 6-9, Figures 7-1, 7-9, 7-10, Figures 8-3, 8-5, Figure 9-5, Figures 10-8, 10-9, 10-11 and 10-13, Figures 11-2, 11-6, 11-7, Figure 12-5 and Figures 13-5, 13-6, 13-9, and 13-12:

Atmel-6430G-ATARM-SAM3U-Series-Datasheet\_31-Mar-15

Figure 1-10: <https://www.wayneandlayne.com/projects/blinky/design/>

Figure 1-14: <https://www.digikey.com/product-detail/en/tag-connect-llc/TC2050-IDC-430/TC2050-IDC-430-ND/2605368>

Figure 1-15: Engenuics Technologies

Figure 1-19: <https://pitchengine.com/dd/2017/11/20/crystal-oscillators-market-report-2016-2023/002518911240077147806>

Figure 3-2: SAM3U User Guide 6430F-ATARM-21-FEB-12, p, 59

Figure 5-1: <http://www.teledynemicro.com/space/fine-pitch-wire-bonding>

Figure 5-8: Register table for the PIO controller SAM3U Series [DATASHEET]

Figure 6-3: 6430F-ATARM-21-Feb-12, p. 5

Figures 12-14, 12-17, 12-18 and 12-20:

Sitronix ST7036 Dot Matrix LCD Controller/Driver data sheet, p. 16

Figure 14-1: Interfacing with ANT General Purpose Chipsets and Modules v2.0, p.4, Dynastream Innovations

Figures 14-5 and 14-6:

ANT message protocol and usage 5.1.pdf page 34, Dynastream Innovations Inc.

## Table of Contents

<b>Preface.</b>	<b>15</b>
<b>Chapter 1 • Getting started</b>	<b>19</b>
1.1 • A Micro what?	19
1.2 • The 32-bit Processor	24
1.3 • Microcontroller Programs	25
1.4 • The Clock	30
1.5 • The Fundamentals	32
1.5.1 • Number Systems	32
1.5.2 • Ohm's Law	36
1.6 • Switches	37
1.7 • Light Emitting Diodes (LEDs)	39
1.8 • Transistors	42
1.9 • Power Supplies	46
1.9.1 • Linear Regulators	47
1.9.2 • Switching Power Supplies	49
1.10 • Development Board Hardware	50
1.11 • Block Diagram	51
1.12 • Power Input	52
1.13 • Main processor, JTAG, clocks, and connections	53
1.14 • ANT 2.4GHz Transceiver	55
1.15 • User IO: LCD, LEDs, Buttons, Beeper	56
1.16 • J-Link On-Board	56
1.17 • Summary	59
<b>Chapter 2 • Development Environment &amp; Version Control</b>	<b>61</b>
2.1 • IAR Integrated Development Environment	61
2.2 • IAR Installation	61
2.3 • Setting Up a New Project	63
2.4 • Files in a New Project	71
2.5 • IAR Simulator and Debugger	75
2.6 • Other Development Tools	81

2.6.1 • Tera Term . . . . .	81
2.6.2 • GitHub Desktop. . . . .	82
2.6.3 • ANTWare II . . . . .	82
2.6.4 • nRFgo Studio . . . . .	82
2.6.5 • Windows Settings . . . . .	83
2.7 • Version control with Git . . . . .	83
<b>Chapter 3 • ARM Cortex-M3 Assembly Language. . . . .</b>	<b>91</b>
3.1 • Core Registers . . . . .	91
3.2 • Instructions . . . . .	93
3.3 • Assembly Language Syntax . . . . .	94
3.4 • Load and Store Instructions . . . . .	101
3.5 • Hello World in Assembly. . . . .	103
<b>Chapter 4 • Embedded C. . . . .</b>	<b>115</b>
4.1 • Documentation . . . . .	115
4.2 • Doxygen . . . . .	117
4.2.1 • Documenting the “right information” . . . . .	117
4.2.2 • Create a Configuration File . . . . .	118
4.2.3 • Special Comment Blocks . . . . .	120
4.2.4 • Doxygen tags in code . . . . .	122
4.2.5 • Doxygen Example . . . . .	123
4.3 • Coding Conventions. . . . .	125
4.3.1 • Type Definitions . . . . .	125
4.3.2 • Hungarian Notation . . . . .	126
4.3.3 • Preprocessor Symbol Definitions . . . . .	127
4.3.4 • Braces { } . . . . .	127
4.3.5 • Switch statements. . . . .	127
4.3.6 • White Space . . . . .	128
4.3.7 • Global Variables . . . . .	128
4.3.8 • Doxygen Tags . . . . .	129
4.3.9 • Function Declarations. . . . .	129
4.3.10 • State Declarations . . . . .	130
4.3.11 • Tabs and Indenting . . . . .	130

---

4.3.12 • Operator Precedence . . . . .	130
4.4 • C project file overview . . . . .	131
4.4.1 • Accessing Registers . . . . .	134
4.5 • How A Processor Starts Up . . . . .	137
4.5.1 • Watchdog Timer . . . . .	137
4.5.2 • Clock and Power Initialization . . . . .	139
4.5.3 • Implementing the clock setup . . . . .	141
4.6 • GPIO Initialization . . . . .	143
4.7 • Program Structures . . . . .	144
4.7.1 • The Infinite Loop . . . . .	144
4.7.2 • Operating Systems . . . . .	145
4.7.3 • State Machine Super Loop . . . . .	147
4.8 • Implementing the SM Super Loop . . . . .	150
4.8.1 • Initialization . . . . .	151
4.8.2 • State Machine Super loop . . . . .	151
4.9 • helloworld.c . . . . .	154
4.10 • Next Steps . . . . .	158
<b>Chapter 5 • GPIO &amp; LED Driver . . . . .</b>	<b>161</b>
5.1 • SAM3U2 General Purpose Input Output . . . . .	161
5.2 • External Hardware . . . . .	162
5.2.1 • Pin Allocation . . . . .	163
5.3 • The PIO Peripheral . . . . .	168
5.4 • PIO Internal Hardware . . . . .	169
5.4.1 • Logic Block Diagram . . . . .	170
5.5 • PIO Registers . . . . .	174
5.6 • Adding a New Task . . . . .	178
5.7 • The LED Driver . . . . .	181
5.8 • Driver Implementation . . . . .	184
5.9 • Blinking . . . . .	188
5.9.1 • Map File . . . . .	190
5.10 • Chapter Exercise . . . . .	192

<b>Chapter 6 • Interrupts &amp; Button Drivers</b>	<b>193</b>
6.1 • Interrupts	193
6.2 • Interrupts on the SAM3U2	194
6.2.1 • Interrupts depend on hardware.	195
6.2.2 • Interrupts need to be configured by firmware	195
6.2.3 • Interrupts can be enabled and disabled globally	196
6.2.4 • An interrupt forces the processor to run an Interrupt Service Routine	196
6.2.5 • Interrupts have priorities	197
6.2.6 • Interrupts can (and will) occur anytime, anywhere	198
6.2.7 • Interrupts require context preservation	198
6.2.8 • Interrupts set flags that need to be cleared	199
6.2.9 • ISRs should be short and fast.	199
6.3 • Interrupt User Guide Resources	199
6.4 • Interrupts and C	203
6.4.1 • Vector Table	203
6.4.2 • Priorities	203
6.4.3 • Enabling and Disabling Peripheral Interrupt Sources	204
6.5 • Peripheral Interrupts	207
6.5.1 • GPIO Interrupts	207
6.5.2 • Timer / Counter Interrupts	207
6.5.3 • Communication Peripheral Interrupts.	207
6.5.4 • Other Peripheral Interrupts.	208
6.6 • Button Driver Overview and Setup	208
6.6.1 • Debouncing	208
6.6.2 • Button history or edge detection	209
6.6.3 • Button held	209
6.7 • Button Operation	209
6.7.1 • Button Typedefs	211
6.8 • PIO Interrupts	212
6.9 • Button API	222
6.10 • Chapter Exercise	224

<b>Chapter 7 • Sleep, System Tick and Timer Peripheral</b>	<b>225</b>
7.1 • Sleep	225
7.2 • System Tick Configuration	226
7.2.1 • Tick Time and CTRL INIT value	227
7.3 • Timer Peripheral	233
7.4 • Timer Counter Highlights	235
7.5 • Timer Counter Registers	237
7.6 • Timer Driver	239
7.7 • Timer API	240
7.8 • Chapter Exercise	245
<b>Chapter 8 • Pulse Width Modulation</b>	<b>247</b>
8.1 • PWM Concepts	247
8.2 • PWM the Easy Way: SAM3U2 PWM Peripheral	249
8.3 • Peripheral Highlights	250
8.4 • PWM and Audio	251
8.5 • EiE Audio Hardware	252
8.6 • PWM Registers	253
8.7 • Development Board Audio Driver	256
8.7.1 • Audio function initialization	256
8.7.2 • Audio API Functions	258
8.7.3 • PWMAudioOn() and PWMAudioOff()	261
8.8 • PWM the Hard Way: Bit Bashing	262
8.9 • LED PWM Design	263
8.10 • Audio Bits	268
8.11 • Multiple User Tasks	270
8.12 • Chapter Exercise	272
<b>Chapter 9 • DMA and Messaging</b>	<b>273</b>
9.1 • Data Transmission	273
9.2 • Resource Conflicts	275
9.3 • Direct Memory Access – DMA	276
9.3.1 • PDC Registers	279
9.3.2 • PDC Interrupts	281

9.3.3 • Transmitting with DMA . . . . .	282
9.3.4 • Receiving with DMA . . . . .	283
9.4 • Linked Lists . . . . .	284
9.5 • Hard Faults . . . . .	286
9.6 • EiE Messaging Task . . . . .	289
9.6.1 • Message Task Data Structures . . . . .	290
9.6.2 • Message Task Protected Functions . . . . .	295
9.7 • Messaging Public Functions . . . . .	305
9.7.1 • QueryMessageStatus(). . . . .	305
9.8 • Messaging State Machine . . . . .	306
9.9 • Chapter Exercise . . . . .	307
<b>Chapter 10 • Serial and Bugs for Breakfast . . . . .</b>	<b>309</b>
10.1 • RS-232 Overview . . . . .	309
10.1.1 • Clocking . . . . .	312
10.1.2 • Signaling . . . . .	313
10.2 • Data Errors . . . . .	315
10.3 • ASCII . . . . .	316
10.4 • Storing and Displaying Characters . . . . .	317
10.5 • SAM3U2 UART Peripheral . . . . .	320
10.5.1 • Peripheral Highlights . . . . .	321
10.5.2 • Baud Rate Generator . . . . .	322
10.6 • UART Registers . . . . .	324
10.6.1 • EiE UART Driver . . . . .	326
10.6.2 • UART Task Data Structures . . . . .	327
10.6.3 • UART Driver Functions . . . . .	329
10.7 • UART Interrupts . . . . .	335
10.8 • UART Driver Design . . . . .	336
10.8.1 • Data Transmit . . . . .	336
10.8.2 • Data Receive . . . . .	340
10.9 • Dynamic Memory Allocation . . . . .	342
10.10 • Debug Task . . . . .	345
10.11 • Debug API . . . . .	346



10.11.1 • DebugPrintf()	346
10.11.2 • DebugPrintNumber()	347
10.12 • Reading Character Input.	349
10.12.1 • DebugInitialize()	350
10.12.2 • DebugRxCallback()	351
10.13 • Debug Programmer Access.	352
10.14 • Terminal Control Codes	361
10.15 • Chapter Exercise	362
<b>Chapter 11 • I SPI with my I2C</b>	<b>363</b>
11.1 • SPI Signaling	363
11.2 • SPI Hardware	368
11.3 • SPI Registers	369
11.4 • EiE SPI Driver	373
11.5 • Master Transmit	375
11.6 • Master Receive	375
11.7 • Slave Transmit	376
11.8 • Slave Receive	376
11.9 • Slave Transmit with Flow Control.	376
11.10 • Slave Receive with Flow Control	377
11.11 • Chip Select	377
11.12 • SPI Data Structures	378
11.13 • SPI Driver Functions in Common with UART	380
11.14 • New SPI Driver Functions	381
11.14.1 • SspRequest()	381
11.14.2 • SspRelease()	382
11.14.3 • SspAssertCS() / SspDeassertCS()	383
11.14.4 • SspReadByte() / SspReadData()	384
11.14.5 • SspQueryReceiveStatus()	385
11.14.6 • SspGenericHandler()	386
11.15 • Ssp State Machine	392
11.16 • Blade Daughter Board Interface	396
11.17 • Blade Example Project	398

11.17.1 • Blade Firmware Configuration Defaults and Interface . . . . .	398
11.17.2 • UserApp1Initialize() . . . . .	401
11.17.3 • UserApp1SM . . . . .	402
11.18 • Chapter Exercise . . . . .	403
<b>Chapter 12 • I<sup>2</sup>C &amp; ASCII LCD . . . . .</b>	<b>405</b>
12.1 • Inter-Integrated Circuit (I <sup>2</sup> C) Communication . . . . .	405
12.2 • I <sup>2</sup> C Hardware . . . . .	406
12.3 • I <sup>2</sup> C Signaling . . . . .	408
12.4 • EiE TWI Hardware . . . . .	411
12.5 • I <sup>2</sup> C (TWI) on SAM3U2 . . . . .	411
12.6 • TWI and PDC . . . . .	412
12.7 • TWI Registers . . . . .	414
12.8 • TWI Driver . . . . .	416
12.8.1 • TWI Data Structures . . . . .	417
12.8.2 • TWI Driver Functions . . . . .	418
12.9 • TWI State Machine and ISR . . . . .	423
12.9.1 • TWI Transmit . . . . .	423
12.10 • TWI Receive . . . . .	428
12.10.1 • NACK and Errors . . . . .	431
12.11 • ASCII LCD . . . . .	433
12.11.1 • LCD Hardware . . . . .	433
12.11.2 • LCD Controllers . . . . .	435
12.11.3 • LCD Interface . . . . .	436
12.12 • Character and Control Data . . . . .	437
12.13 • Using the LCD Controller . . . . .	438
12.14 • Control byte with Co and Rs . . . . .	440
12.14.1 • Character RAM Addresses . . . . .	441
12.14.2 • LCD Command Set . . . . .	442
12.14.3 • LCD Initialization . . . . .	443
12.15 • LCD Application . . . . .	446
12.15.1 • LcdCommand() . . . . .	447
12.15.2 • LcdMessage() . . . . .	447

12.15.3 • LcdClearChars(). . . . .	448
12.16 • Chapter Exercise . . . . .	449
<b>Chapter 13 • Analog to Digital Conversion . . . . .</b>	<b>451</b>
13.1 • ADC background . . . . .	451
13.1.1 • Quantization . . . . .	451
13.1.2 • Sampling . . . . .	451
13.1.3 • Bandwidth and Aliasing . . . . .	452
13.1.4 • Nyquist Frequency . . . . .	453
13.1.5 • Resolution . . . . .	454
13.1.6 • Clipping . . . . .	455
13.2 • Characteristics of ADCs . . . . .	455
13.2.1 • Precision, Error, and ENOB . . . . .	456
13.2.2 • Missing codes . . . . .	457
13.2.3 • Reference Voltages . . . . .	457
13.2.4 • Noise . . . . .	457
13.2.5 • Single vs. Differential Measurement. . . . .	458
13.2.6 • Signal Conditioning . . . . .	458
13.3 • EiE ADC Hardware. . . . .	460
13.4 • SAM3U2 12-bit ADC Peripheral . . . . .	460
13.4.1 • ADC Registers. . . . .	462
13.5 • EiE ADC Driver . . . . .	464
13.5.1 • ADC Initialization. . . . .	466
13.5.2 • ADC Interrupt . . . . .	467
13.5.3 • ADC State Machine . . . . .	468
13.6 • EiE ADC API . . . . .	468
13.6.1 • void Adc12AssignCallback( ) . . . . .	468
13.6.2 • bool Adc12StartConversion() . . . . .	469
13.7 • Chapter Exercise . . . . .	470
<b>Chapter 14 • ANT Radio System. . . . .</b>	<b>473</b>
14.1 • ANT Wireless Radio . . . . .	474
14.2 • Building the ANT Stack . . . . .	475
14.2.1 • ANT Physical Layer . . . . .	476

14.3 • ANT Message Protocol and Usage . . . . .	477
14.3.1 • ANT Protocol: Sections 1 thru 4. . . . .	477
14.3.2 • ANT Protocol Section 5: Channel Parameters . . . . .	478
14.3.3 • ANT Channel ID. . . . .	479
14.3.4 • Transmit Data Types . . . . .	479
14.3.5 • ANT Protocol Section 6: Pairing . . . . .	481
14.3.6 • ANT Protocol Section 7: ANT Interface . . . . .	482
14.3.7 • ANT Protocol Sections 8 and 9: Examples and Appendix. . . . .	483
14.4 • Message Handling . . . . .	483
14.4.1 • Messaging with Channel Closed. . . . .	484
14.4.2 • Messages When Channel is Open. . . . .	485
14.5 • Debugging an ANT System . . . . .	486
14.6 • Programming the ANT Sub-System . . . . .	488
14.6.1 • Firmware Design ant.c and ant_api.c . . . . .	488
14.6.2 • Data Structures. . . . .	490
14.6.3 • Serial Drivers and ANT hardware interface . . . . .	491
14.6.4 • Task access to send and receive . . . . .	517
14.6.5 • ANT_TICK and ANT_DATA. . . . .	518
14.7 • ANT State Machine . . . . .	526
14.7.1 • Initializing the ANT SM. . . . .	526
14.8 • Implementing the ANT State Machine . . . . .	529
14.9 • API Summary . . . . .	532
14.9.1 • ANT Configuration and Status Message . . . . .	533
14.9.2 • ANT Data Messages . . . . .	536
14.10 • Chapter Exercise . . . . .	539
14.11 • Conclusion . . . . .	540
<b>Glossary . . . . .</b>	<b>541</b>
<b>Index . . . . .</b>	<b>547</b>

## Preface

Microcontrollers are everywhere in the world today providing intelligence to billions of electrical systems that people interact with – or don't interact with — every day.

Microcontrollers are programmed with firmware and combined into circuits to get their job done. These devices are what we call “Embedded Systems.” Chances are any electronic device in use today contains at least one microcontroller, and complicated systems may have dozens of micros embedded in them to handle different tasks.

There are thousands of different microcontrollers to choose from. These choices can be sorted out rather quickly based on what you are looking to design. As you design more products, you will no doubt end up with some favorite microcontrollers that you become an expert at using. You will also likely design products for similar applications or with similar capabilities and therefore develop expertise in a particular field or application of microcontrollers.

What you should gather from that is that your own experience is unique, and what you know and how you have learned will always be different than the people you work with. The first four years of my own embedded experience started with programming PIC microcontrollers in assembly language where literally every byte of flash and RAM had to be used as effectively as possible. This shaped me into being very pragmatic and “low level” in my designs and thinking.

Even as I evolved to writing in C for 32-bit processors, my established fundamentals remained. I still think a lot about assembly language even when coding in C and strongly believe that a programmer should have a solid understanding of the instruction set of the processor being used. Though I can code in both C++ and Java, I have not done any commercial designs using those languages. However as you will see, I incorporate many of the concepts of high-level object-oriented programming in my C code albeit sometimes only by a written rule or process that I follow. I see more and more developers doing this.

The diversity in people's experiences is extremely important to recognize because it means that there are always opportunities to learn from and share what you know with other people. Collaboration is one of the greatest benefits of being alive, so open your mind and bring ideas together. Also, remember that in hindsight it is very easy to pick out elements of a design that can be improved – you should always do this with your own designs. If you do not find anything that you would do differently if you did it all over again, you probably are not looking hard enough.

Embedded in Embedded tries to teach the fundamental, foundational concepts of embedded systems development as generically as possible. Indeed many of the concepts and processes that you will learn here can be applied to a variety of processors for a variety of applications. To provide a resource that is useful and with discussion of intricate details of design decisions, a specific processor on a specific development board with a specific development environment is used. I cannot stress enough that I totally understand that EIE is just ONE way, not THE way, of writing the drivers and programs that I will show you.

As with all engineering decisions, there are a vast number of tradeoffs when it comes to making decisions about how to complete a design, many of which will be discussed in this book as we come across them. In the firmware world, a general rule of thumb is that the easier and more robust or feature-rich a piece of code or application interface is to use, the more complicated the source code behind it will be.

A great example I like to use is comparing RS-232 to USB as standard serial interfaces. Simple RS-232 functionality can be written in just a few lines of code, but if you have ever had the pleasure of trying to get RS-232 serial devices to work properly on a


computer you will know how frustrating that can be. On the other end of the spectrum, there is USB, where a typical bare-bones driver on the embedded system will be at least 2kB of code. But USB is perhaps the epitome of plug-and-play, and computer users today expect nothing less than to plug in any USB device to any USB socket and immediately have a perfect operation.


With that in mind, we dive in to explore one way of writing a reasonably robust, reasonably small footprint, reasonably expandable, and reasonably multi-tasking embedded system. All based on the needs, resources, and knowledge at the time of writing. The ARM Cortex family of processors happens to be my current favorite, and for a variety of reasons, I designed the EiE development boards using Atmel's SAM3U2 Cortex-M3 microcontroller. I use IAR Systems IDE as I have used this for a long time across many different processors. For the record, I have never worked for ARM, Atmel/ Microchip, IAR or any of the other vendors whose names you will see on the EiE hardware – every design decision I have made is based on my own experience with our research into the part or package that best meets the need of the design. Over time I have invited these companies into the EiE network for mutual benefit. For those that know me, you know that I am genuinely all about win-win.


You can use this book in one of four ways:

1. Read it from start to finish developing all the code by yourself and comparing it to the solutions provided. This is by far the most comprehensive approach. Since the goal of the EiE program is to jump-start young engineers through the first two years of working as an embedded developer, that is truly the way to gain this experience.
2. Skim through and highlight what is important for you to know, or what might be important for upcoming projects. Reference the solutions provided and use the book's explanations to complement the code documentation.
3. Work backward from the application level and reference this book when you need or want to know the theory or decisions behind the code.
4. Choose specific chapters or topics of interest a la carte and use the book as a reference.

In every chapter you will see three icons that flag certain text:

 Information. Use this icon to indicate some extra information relevant to the topic being discussed.

 Do some work. Use this icon to indicate the reader should perform a task at this point.

 This is massively important. Do not move past this point in the notes if you do not understand this yet.

It is my sincere hope that you get a lot out of this book and that it helps you gain skill as an embedded designer. If you love this system and adopt it (or parts of it) in your designs, that is great. If you hate this system and decide it is the best example of what never to do, that is great, too. There is as much value – or possibly more – in seeing what NOT to do as there is to seeing what to do.

In a world full of pre-developed solutions that people stitch together and call it “designed” I hope the story of this code shows how much thought and effort goes into making a system. Yes, we can patch together code and make use of work others have done, but when we design products for industry, how will these systems behave?

Will you understand your code next week, next month, or next year? Will your colleagues ever understand your code? How many bugs will your customers find? Do you understand the system you built and the tools you have to find and fix those bugs?

One thing I can say for sure is that after 18 years working with and teaching embedded systems, I know it is our processes that are most critical in being great developers, and if nothing else, this book will demonstrate the processes I rely on. To my knowledge, there are no other authors and/or engineers that have taken the time to go through and describe an entire development of a system as I will do here. Sharing this is a little like getting undressed in the middle of a crowded room... I hope you like what you see and find great value here!

Jason Long





## Chapter 1 • Getting started

In 1999, the world was poised for what was thought was going to be the most significant disaster to hit the modern world: the “Y2K” computer bug. People everywhere were desperately trying to upgrade systems and test their computers to see if they would suffer from the bug. Yet even if every personal computer was safely protected, there was a great concern for all the embedded processors that were virtually everywhere and could not be reached in time.



Figure 1-1 "Y2K" magazine cover

Fortunately, the new millennium started with barely a flicker of the lights. Though some may argue that the problem was completely blown out of proportion, no one can contest that many of us were clued into the incredible impact that computers and embedded systems have on our lives. There are literally billions of microprocessors at work in our electronic devices. Everything from coffee makers, garage door openers, sprinkler systems and pop machines etc. have microprocessors these days – a high-end automobile has dozens of them and with the advent of self-driving cars, the computing power in a vehicle will increase massively. Estimates on the global value of embedded systems and the supporting industry are in excess of a trillion dollars.

Engineers working in this space know that technologies have grown linearly, but there has been exponential growth in awareness and application of embedded systems. The mainstream adoption of the term “Internet of Things” has brought embedded computing to a whole new level, just as the term “Cyberspace” ushered in the internet to the average person. The world is truly embedded in embedded systems!

This chapter takes you through some of the fundamental knowledge in embedded systems.

### 1.1 • A Micro what?

So, what are microprocessors and microcontrollers? Both are digital devices packed full of transistors that make up logic gates and data paths. By turning on or off combinations of those transistors, data can be accessed, processed and distributed throughout a device and to other devices that are connected in a circuit. This data can come from places such as memory, input/output lines, and from the processor core itself. All this is managed by the program (firmware) that is written and executed by the processor.

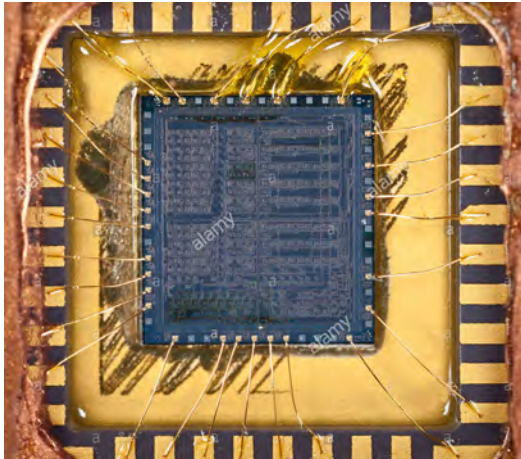


Figure 1-2 Example of a microprocessor

A microprocessor is the “core” where the instruction set of the processor lives and performs calculations. A core needs to be attached to external memory and other peripherals if it is going to be useful. It talks to RAM where programs and data are stored, communicates to the display to show you what is going on, processes the bytes that play music and sounds, and exchanges information with data storage devices. It is the main brain of an electronic device, just like the processor in a desktop PC or smartphone.

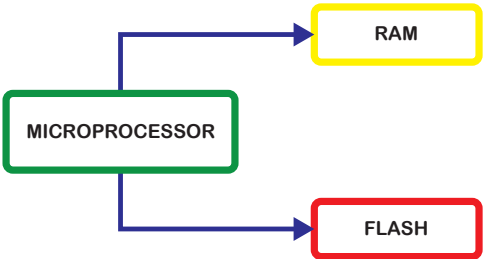


Figure 1-3 Microprocessor communicates with external memory

A microcontroller is an integrated circuit that contains a microprocessor, but also may have onboard memory, timers, communications peripherals, analog to digital converters, and many others. The microcontroller can usually run on its own with barely any external devices connected to it, though it still needs to be in a circuit to control. Literally billions of electronics in the world are microcontroller-based embedded systems.

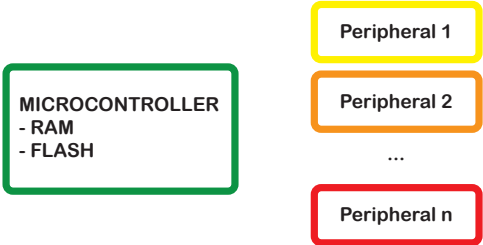


Figure 1-4 Microcontroller-based embedded system

[illegible]

**Figure 1-5 Comparison of features and options of Microchip processors**

Since everything is integrated on a single chip, a microcontroller-based system is more resource-limited than a microprocessor-based system that could have practically unlimited volatile and non-volatile memory attached. Even high-end microcontrollers max out around 2MB of non-volatile flash and 1MB of RAM and have clock speeds topping out at 200MHz. That might not sound like a lot compared to microprocessor systems with capabilities and resources order of magnitudes higher, and it's also a great way to date this book since advances are always being made. However, what you can do with even an 8-bit microcontroller with 1kB of flash and 100 bytes of RAM is still amazing. Microcontrollers tend to be simpler to work with, easier for hardware design, and are inherently less expensive.

Embedded in Embedded uses the SAM3U2 Cortex-M3 microcontroller from Atmel (which is now Microchip after being acquired in 2016). Like any vendor who provides an ARM-based microcontroller, Microchip licensed the ARM core and added peripherals to build its device. The SAM3U2 is just one of several devices in the SAM3U family, which in turn is one of the many Cortex-M3 processors available from Microchip. A Cortex-M3 core provides plenty of power to run our development board and for most of the time, it will be busy sleeping doing nothing at all. The specific device does not really matter for EiE since our goal is to teach the process of design.

So how do you navigate all the processor options to decide on the right one to use in a design? There are many factors and there is no simple formula to put in some parameters and crank out the answer. Likely the biggest deciding factor will be your own experience. The investment you make in learning how to use a particular family of microcontroller might be the most valuable part of the equation, though being open to change is good if it makes sense. If you are new to embedded design or faced with a design where your "toolbox" of available knowledge does not provide any solutions, then you are limited to searching on your own or consulting a colleague or reputable distributor to get advice. As you gain experience, you can more quickly identify what you may or may not need for features and capabilities.

Learning a new microcontroller is exciting, but even industry experts will experience a learning curve if they change platforms to something entirely new. "Time to market" is one of the timeless mantras in engineering, so it is inherently a good idea to minimize new learning in a product and maximize reusability. Easier said than done and you will find an almost infinite number of variables to consider. Cost, power consumption, and available peripherals are very important. A strategy that we have found successful for specifying a micro for a new product is to aim for around 50% total resource usage for the part you choose and ensure there is at least one part in the family that you could scale to further double the resources. For example, if you think you need 32kB of flash and 8kB of RAM for your product, we would suggest starting with a processor that has 64kB of flash and 16kB of RAM available. Within the family, there should be drop-in parts with 128kB of flash and 32kB of RAM.

The physical size of the chip is another factor. If you prepare a block diagram and decide you need 30 input/output pins, you probably want to have at least 10 spares. If you are upgrading an existing design, the margins can be much tighter. No doubt it is difficult to predict what you might need 2-3 years down the road, but a good brainstorming session of "what features will Marketing think of" is a good way to think outside of the box to anticipate reasonable growth of the product.



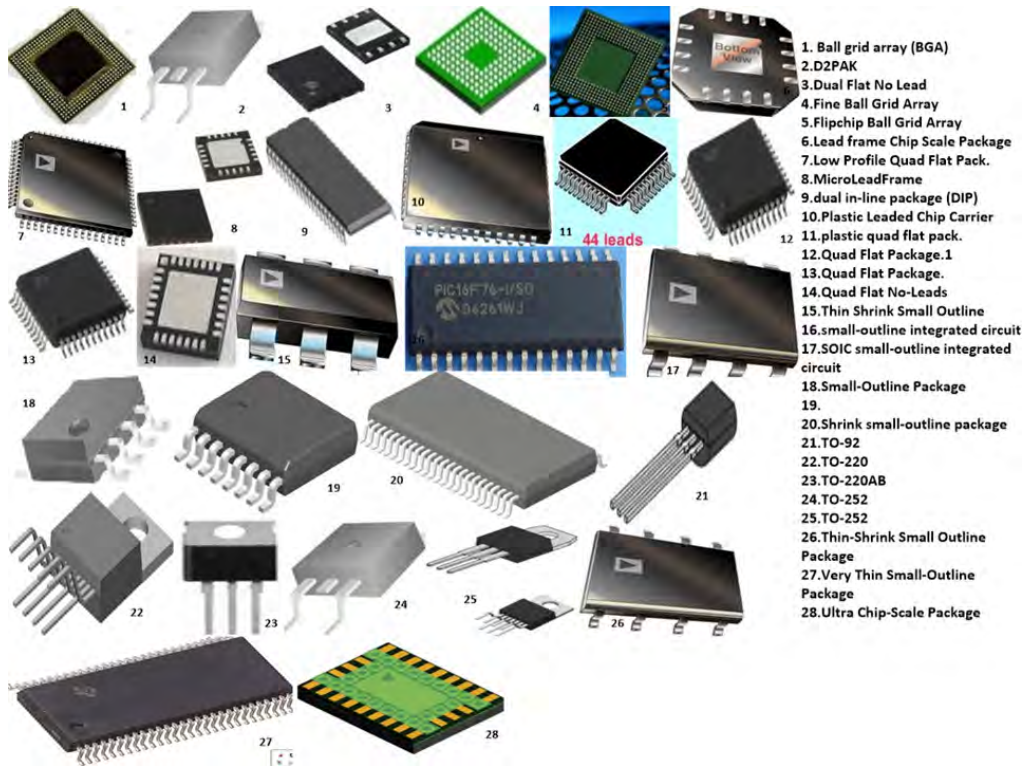


Figure 1-6 Examples of various component packages

One of the nice things about using an ARM-based microcontroller is that if you run out of power, you can migrate to a more powerful core like the Cortex-M4 or all the way up to an applications processor. If you need less power, then you can downgrade within the portfolio to the Cortex-M0. Since all the processors are built with very similar architecture, you can change quite freely and the underlying concepts plus all your development tools and knowledge remain applicable. While we do love ARM processors, that does not mean we use them exclusively. There are plenty of applications where we will look at an 8-bit or 16-bit solution if the requirements make sense to do so.

The Cortex family also attempts to make porting between different vendors as easy as possible. ARM developed a standard for Cortex devices called CMSIS – Cortex Microcontroller Software Interface Standard. It is growing in complexity but the portion relevant to us is the CMSIS-CORE. This part of the standard provides an interface to the ARM core and the core peripherals. Vendors also use the CMSIS pointer-to-struct style for accessing the microcontroller peripherals that they add. The exact register names and organization is different between vendors, but at least the main organization is the same.



Figure 1-7 Cortex microcontroller software interface standard

For example, ST has a nice family of Cortex-M3 and Cortex-M4 processors not to mention some very low-cost development boards, so you might be interested in taking some of the code from EiE to an ST micro. Porting high-level code is very simple since it has

been designed with a high degree of hardware abstraction, but porting peripheral drivers will require a lot of peripheral-specific re-write. In fact, the EiE firmware system started as a product-specific version of that code on an ST Cortex-M3 way back in 2008. This was ported to an LPC2148 ARM7 and then an LPC1752 Cortex-M3 where the code was further developed. The SAM3U2 was chosen in 2013 for final development because it had the perfect hardware peripherals and pin count that we needed for EiE and was also compatible with Segger programmer firmware.

The ST to NXP move inspired us to better layer the code and abstract each task, and when we later ported to the SAM3U2 it was a lot easier, though still took many weeks. We re-used a lot of the code structure and in some cases just had to determine the names of registers that were different. That being said, writing a firmware system with portability to other vendors is probably not an appropriate level of effort to expend, though it all depends on what you want to accomplish.

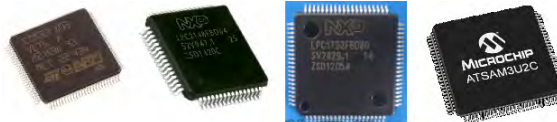


Figure 1-8 Cortex-M3 processors

## 1.2 • The 32-bit Processor

We have already mentioned that a microcontroller has many onboard peripherals that it uses to operate. The microcontroller attaches to the physical world through its IO pins, many of which are connected to the internal peripheral blocks that provide other capabilities to the processor. The figure below is the block diagram of the SAM3U2. Note all the different peripherals, memory, buses, and connections.

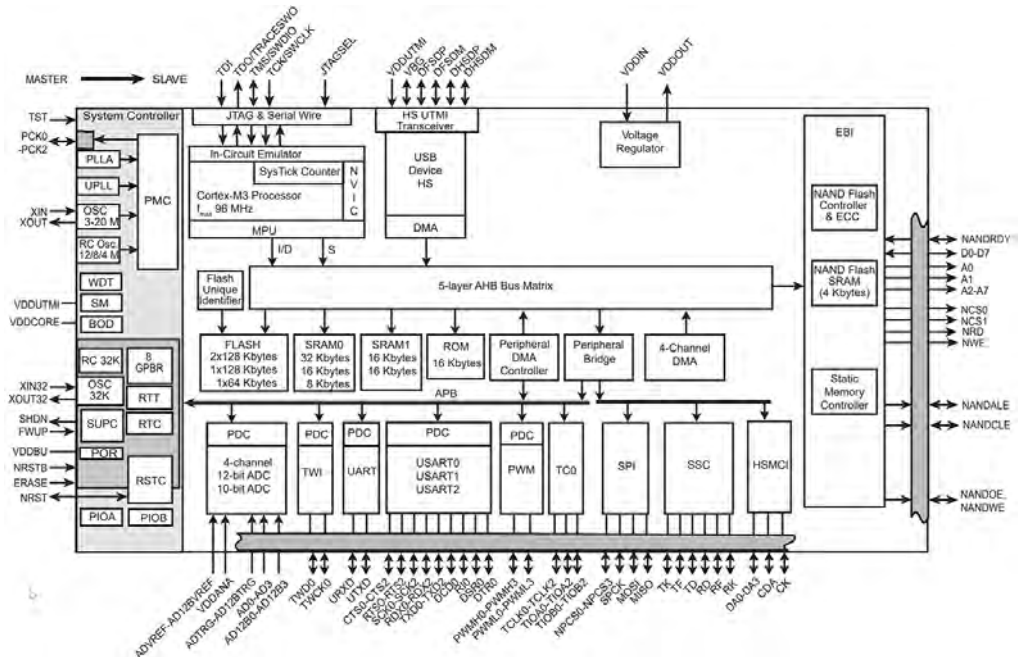
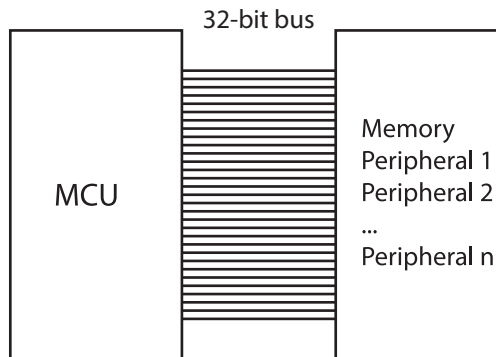


Figure 1-9 100-pin SAM3U4/2/1C block diagram



Even though these peripherals are built-in, they still require a physical connection over which they exchange data. This communication is done over a set of connections called a “bus.” In the diagram above, you can see two buses: AHB (Advanced High-speed Bus) and APB (Advanced Peripheral Bus). Depending on the processor architecture, there may be multiple buses for different purposes such as for instructions, data, and peripheral access. There are historically two different architectures: Harvard (single bus for instructions and data) and Von Neumann (separate data and instruction buses). The Cortex-M3 family happens to be Harvard architecture. A designer practically never worries about the type of architecture of a processor – the implementation ends up being transparent unless the requirements of the device are very specific for some reason. The Wikipedia page on ARM Cortex-M provides a wealth of information if you are interested ([https://en.wikipedia.org/wiki/ARM\\_Cortex-M](https://en.wikipedia.org/wiki/ARM_Cortex-M)).

The main bus on a SAM3U2 is 32 bits wide. In other words, there are 32 wires that connect the area where the program runs with the other parts of the system. This also means that the main registers and memory locations can be 32-bits wide. The native data size is typically called “word” size. Microcontrollers also come in 8 and 16-bit versions. Desktop PCs use 64-bit processors these days. The transition from 32-bit PCs started around the Pentium-4 processor. At the time of writing, there are not any well-known 64-bit microcontrollers, though there are a growing number of dual-core embedded micros entering the market that feature a higher end core like a Cortex-M4 paired with a lower power Cortex-M0.



**Figure 1-10 32-bit bus and addressing**

An 8-bit processor can do many of the same things as a 32-bit processor, but any time an operation requires numbers larger than 255, an 8-bit processor must complete tasks in pieces. Data transfer is also limited by the smaller bus size. If you want to add two 16-bit numbers on an 8-bit processor, you must take several steps to add the low bytes, watch for a carry bit, then add the high bytes and the carry bit (and watch for a carry bit there, as well). Another important consideration is that there is a limitation on the amount of memory that can be accessed directly. If the data bus is only 8-bits, then only 255 bytes of memory can be addressed directly, although there are lots of ways to add additional addressing bits. A 32-bit processor can inherently access over 4GB of memory.

### 1.3 • Microcontroller Programs

No matter what programming language is used to write a program (for example C or C++), the source code is compiled into the assembly language for the specific processor that will run the code. Programs may also be written directly in Assembler. Assembly language works with the collection of instructions that a processor knows how to process which is called the “instruction set.” All Cortex-M3 processors use the same instruction set

regardless of the vendor's microcontroller they are in.

The instructions and arguments in an assembly language program are converted into "opcodes" and saved to a hex (hexadecimal) file by the Assembler. Assembling is a much simpler process than Compiling and every developer's software Assembler should come up with the same hex file for a given program. The hex file is a bunch of addresses, data, and some error checking/checksum information. The 1's and 0's of the opcodes will be loaded into a processor's memory at the specified addresses and eventually be executed one-by-one when the core starts running.

There are various formats that this file can have, with one of the most common being "Intel Hex."

**Intel Hex file:**

:	num bytes	address	record type	data	checksum
:	100800	00042CFF	3FFF3FFF3F	8801762F8B1378308A	
:	100810	00210099	0087318E27	84318731AA278431BE	
:	100820	0087319D	2784318731B4	2784318B1700307D	
:	100830	0087315D	2784312000A0	002008D1005108B5	

Figure 1-11 Intel Hex file

The example below is a hex file for a program that turns on a light if a button is pressed on a certain microcontroller.

#### Listing 1-1

```
:020000040000FA
:100000000831601308600831206188610061C86149B
:020010000428C2
:000000001FF
```

The hex file needs to be programmed in a microcontroller's non-volatile memory. Most micros these days use "flash" memory that is both electrically writeable and readable hundreds of thousands of times. This makes learning and testing easy because the chip can be continually reprogrammed until the code works properly. Flash memory replaced EEPROM (Electrically Erasable Programmable Read Only Memory) in most microcontrollers in the late 1990s due to cost. EEPROM is nice because individual bytes can be erased, whereas flash memory is typically organized in large pages that must be erased. Both can write individual bytes so in most cases it's not a big deal. Some microcontrollers have a small EEPROM peripheral that is very handy for saving user information.

At the other end of the spectrum are processors that are one-time programmable (OTP) so you better do some serious debugging before you download your program as you get one chance only. Somewhere in the middle are EPROM's (Electrically Programmable Read Only Memory) that can be easily programmed but can only be erased by shining UV light through a small window on the chips for 15 – 20 minutes which tends not to be very convenient.

To write a program you code to a processor's non-volatile memory, you need some sort of programming tool. Many microcontrollers use a standardized interface commonly referred to as "JTAG" to move a program from the computer on which it is developed to the embedded system. MCUs can be both programmed and debugged through JTAG. JTAG devices are separate pieces of hardware that are usually called "programmers" or "debuggers." A common tool is called a "J-Link" from Segger.

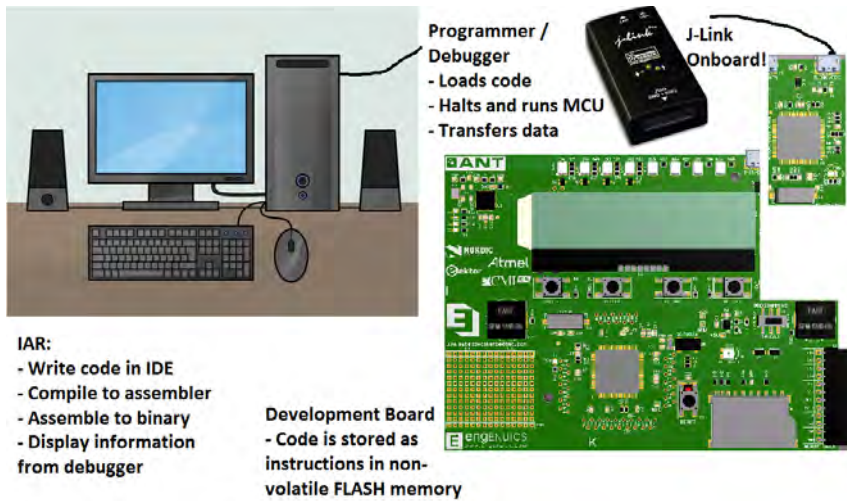


Figure 1-12 Programming tool chain

JTAG programmers/debuggers can be expensive. Serial programming devices are low-cost and can also be used to program but not debug a microcontroller. Serial programming requires the processor to support on-chip programming and have some type of “bootloader” program that runs outside of the programmed memory space. A processor vendor might provide this, or it can be designed into the system.

Regardless of how the connection is made, the act of programming is what physically drives the high and low voltage levels into the target microcontroller. Once the code is programmed in non-volatile memory, it can stay there for over 100 years!

Development boards typically have a JTAG device built-in since programming and debugging is the whole purpose of the board. The second SAM3U2 processor on the EiE main development board is the programmer called a J-Link Onboard (OB). The J-Link OB firmware license is purchased and loaded in the factory. The programmer is connected to a PC and the signal chain is complete.

It would be much too costly to add a programmer to every production device, so most products are designed with a connector to attach the programmer. This allows the developer to write and test code and gives the factory access to program the device during manufacturing. A programming header can consume a lot of space and add cost to each board. The standard JTAG header is a huge 2x10 box-connector. A much smaller standard connector was defined for Cortex parts, but it is quite expensive and from our own experience has not been widely adopted.



Figure 1-13 J-Link programmer and board connectors

Companies will often develop a proprietary connection that works well for their product line. One of our favorites is the “Tag-Connect” system that requires only a footprint on the PCB that a special cable securely attaches to.



Figure 1-14 Tag-Connect system

Cortex microcontrollers support a two-wire programming and debugging interface. With a bit of hacking, you can use a standard USB physical connection. This has nothing to do with USB data - it is just the physical connector. Be aware if you are designing consumer products, re-purposing a USB connector can lead to some confused customers. If the connector is used only for power by the consumer, then it might be ok.



Figure 1-15 USB pinouts

All custom solutions will still require an adapter to plug into a JTAG device. We designed a special J-Link adapter that can be used to connect to any of these options and our own open source solution that is a nice balance of size, cost, and can provide additional UART debugging access.

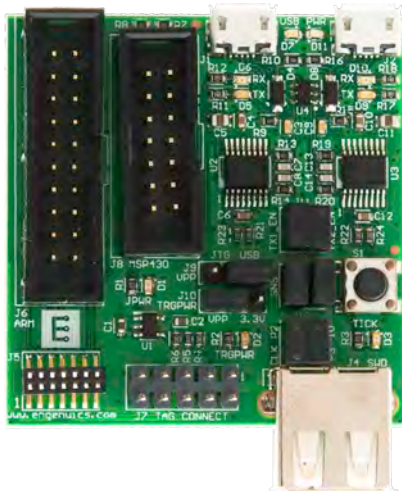
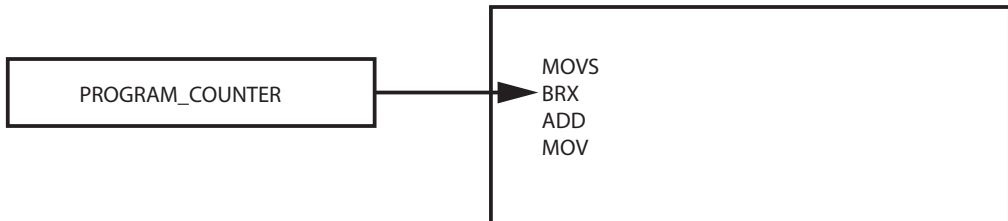


Figure 1-16 Engenuics programming interface adapter

Once the microcontroller is programmed, the code is ready to execute as soon as the power is turned on or the processor is reset to kick it out of programming mode and into execution mode. A program runs by executing each instruction sequentially via a special pointer called the Program Counter (PC) that indexes and reads out each instruction from memory to the bus. The PC always starts at 0 when the device powers on and increments after each instruction with every full clock cycle. Unless an instruction tells the PC to branch or jump, it will keep incrementing to the next address in memory.



**Figure 1-17 Special pointer "Program Counter"**

On a 32-bit processor, the program counter increments by 32 bits (0x04). Many instructions on 32-bit Cortex parts are only 16-bits (called the "Thumb-2" instruction set), so program memory storage can be up to 2x more efficient in some cases. In practice, many instructions end up taking two 16-bit locations, so the net result is less than 2x efficient.

Sometimes an instruction will cause the program counter to change by some number other than the next sequential amount, so it will jump to run instructions from a different part of the program memory. This is how function calls and branches are performed. When the program reaches the end of the programmed memory, you want to ensure that it returns to the beginning with a branch instruction. Otherwise, because of the way the address space in the ARM is set up, it will keep addressing memory and could start doing crazy things like putting the contents of your RAM memory onto the local bus to execute as opcodes. There is nothing stopping you from doing that on purpose and running code from RAM.

The basic operation of a processor and the corresponding flow of data can be understood if you remember a few key points:

1. There are 4 steps required to carry out an instruction: Fetch, Decode, Execute, and Writeback. Simple processors can often run one instruction step per clock cycle, and each instruction is pipelined so the fetch of instruction 2 happens on the same clock tick as the decode of instruction 1. In a processor like the Cortex-M3, most instructions require multiple clock cycles, but they still follow the general fetch, decode, execute, writeback flow.
2. Microcontrollers are digital systems: signals on the bus are merely logic level highs and lows that are momentarily present at discrete locations and are advanced through the system's logic gates with every cycle of the system clock. Signals propagate through basic logic gates as fast as the electrons can move. Eventually, they get stopped and held at circuits called flip-flops until the next clock signal. The maximum clock speed is determined by the worst-case delay through the logic, so the systems assure that the correct signals are waiting at the flip-flops before being advanced to the next set of logic.

## 1.4 • The Clock

The clock signal is fundamental to a digital system. It can be generated in a variety of different ways and most processors will support several different clock sources including some that are built-in and require no external components. In low-cost embedded systems, external clocks are typically a resistor-capacitor circuit which would cost less than a penny. Systems requiring greater precision can use other types of oscillators. Crystals are the most popular choice as they are very precise (typically 30-40ppm) and generally do not drift over power supply ranges, temperature or other operating conditions. At around 50 cents, they are very expensive compared to a simple RC circuit. Crystals also take up more board space and usually require two capacitors that must be carefully selected to make the crystal oscillate at the correct frequency.

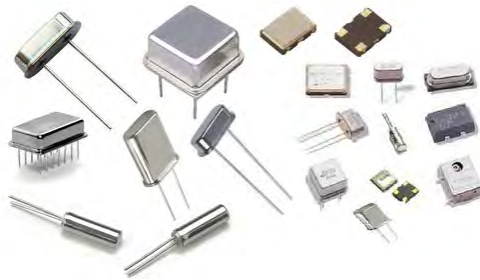


Figure 1-18 Crystal oscillators

During the final review time of this book, we came across some amazing little oscillators from a company called SiTime. These are tiny devices (1.54mm x 0.84mm) and don't require any external components. They advertise better stability and lower power than crystals, and magically cost roughly the same. Many different frequencies are available. We won't have a chance to test them before this goes to print, but if they're as good as advertised this will quickly become a big favorite in our designs.

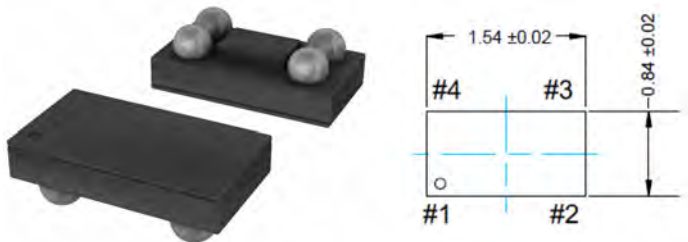


Figure 1-19 SiTime oscillator package

The clock frequency of a resistor and capacitor circuit is at the mercy of the component tolerances, temperature, and supply voltage. The speed of an RC oscillator is simply 1 over the RC time constant. For example, a 20 kΩ resistor with a 100 pF capacitor would yield a 500 kHz clock:

$$\tau = RC$$

$$\tau = 20 \cdot 10^3 \times 100 \cdot 10^{-12}$$

$$\tau = 2 \mu s$$

$$f_{osc} = \frac{1}{\tau} = \frac{1}{2 \cdot 10^{-6}} = 500 kHz$$



Processors that have built-in RC oscillators are specially “trimmed” in the factory so they are nominally quite good, but will still never compare to a crystal. If the processor supports the use of an external RC oscillator, then the range of frequencies and, to a degree, the precision and stability of the clock can be controlled by the designer. There is some internal hardware required for an RC oscillator which will only work to a maximum frequency on the order of 1MHz. Crystals usually start at 1MHz, except for watch crystals that have a very specific frequency of 32.768kHz and happen to be very common in embedded applications where low power and precise timing is required.

Processors often support clock selection and may have both a slow RC or crystal clock source and a high-speed crystal input. The slow clock is used for sleeping or other low-power, non-urgent functions, but the fast clock can be activated for high-speed processing or functions that require precise timing if the slow clock is an RC source or just too slow.

The SAM3U has a maximum clock speed of 96 MHz. The base clock is sourced from one of two crystal oscillator sources or an onboard RC. The internal RC is what the processor always starts up on. The micro offers a phase-locked loop (PLL) that can take whatever base clock signal is provided and up-convert it to a faster frequency with the same precision as the source crystal. On the EiE development board, the base clock signal is a 12.000 MHz crystal which feeds the PLL to provide a 48MHz main clock. 48MHz is chosen as it is the required speed for USB function.

Compared to today’s PC and smartphone processors that are running at GHz clock speeds, an embedded processor at less than 100MHz might not seem very fast at all. However, embedded systems where microcontrollers are used rarely require high-speeds especially if they do not have to support a graphics display or high-speed external communications like Ethernet or USB. In fact, many microcontrollers are deliberately set to run at much slower speeds because they simply do not need to go faster. The slower a micro runs, the less power it consumes, and designers are often very happy to trade speed for power consumption especially in battery-powered devices. The design of the processor plays an important role, as well. An intelligently designed instruction set like the ARM core can accomplish tasks faster by adding logic to the instruction that carries out operations as the instruction is executed such as incrementing or decrementing values or pointers, conditionally branching or executing, and toggling flags.

Even if you examined clock speed on its own, is a 48 MHz system really that slow? Consider a processor’s clock running *48 times slower* at 1 MHz and assume that an instruction can be executed at the end of each cycle. That means that 1 million instructions are being executed every second! Alternatively, you can look at it from the view of a single instruction period. Period T is the inverse of frequency f:

$$T = \frac{1}{f}$$

where T is in seconds and f is in Hz. So,

$$T = \frac{1}{1 \cdot 10^6} = 1\mu s$$

This means that the period of a 1 MHz clock tick is only 1 microsecond (1 millionth of a second). Considering a program can evaluate an input and make a decision in 8 to 10 clock cycles, only 10 microseconds are needed to do something useful. Granted, there are some things that can take hundreds or even thousands of instructions, but MHz speeds are fast enough to handle most embedded applications and the system will still spend most of its time in a low power sleep state.



## 1.5 • The Fundamentals

There are some important basics that you need to have on which you will build your embedded design skills. Some of this material you already know but make sure you review it. Other parts you think you know from the theory taught in school, but it can catch you once you apply it to a real circuit. Some of this material will be brand new. Whatever the case, a solid understanding of these fundamentals is essential to the embedded engineer.

### 1.5.1 • Number Systems

We are all used to the base-10 number system that we use in our everyday lives. A base-10 system makes calculations very simple. When working with embedded systems, it becomes convenient to make use of alternative number systems like binary, octal and hexadecimal – base-2 (binary), base-8 (octal) and base-16 (hexadecimal) systems, respectively. If you are fluent in converting between the different systems, you can improve your success or at least your speed in programming.

A simple way to describe a system base is the quantity you count before rolling over to the next digit. In decimal, we count to 9 in the “ones” position and roll over to 10 – the “tens” position becomes 1, and the “ones” position goes back to zero. Each digit position is a power of 10 (starting with  $10^0 = 1$ ). So the one’s position is  $10^0$ , tens position is  $10^1$ , hundreds position is  $10^2$ , and so on. A base-10 system has 10 possible digits in each position, from 0 to 9. The digit in the position multiplies the power of 10, and the sum of all the digits and their positions is the number itself. That is way more complicated to explain than it is, so check out this quick example to ensure you understand.

Consider the number 709. Break it down into the sum of the all digits and their powers of 10:

$$\begin{aligned} & (7 \times 10^2) + (0 \times 10^1) + (9 \times 10^0) \\ & = (7 \times 100) + (0 \times 10) + (9 \times 1) \\ & = 700 + 0 + 9 \\ & = 709 \end{aligned}$$

Working with a base other than 10 uses the same concepts but it’s a bit difficult because you are not used to it. For the base-2 binary system, you have only digits 0 and 1. The digits in a binary number are typically referred to as “bits”. The lowest value bit is bit 0 and is called the Least Significant Bit or LSB. The highest bit in a binary number is the Most Significant Bit or MSB. Of course, 8-bits make a byte and you might have heard that 4 bits are a nibble.

The EiE notation for binary numbers is  $b'nnnnnnn'$  where  $n$  is 0 or 1. On an 8-bit processor, we only have bit positions 0, 1, ..., 7. When you look at a binary number, the LSB is on the far right.

Name:	MSB							LSB
Bit number:	7	6	5	4	3	2	1	0
Example:	1	1	0	1	0	0	0	1

A number in binary is built using the same rules as building a base-10 number, except the multipliers are only ever 0 or 1, and the powers are those of 2. As a result, binary

numbers have more digits than decimal numbers when representing the same value. When converting binary numbers to decimal, the mechanics of the operation are the same.

Consider the number 145, which in binary is  $b'10010001'$ . Break it down into the sum of the digits and powers of 2:

$$\begin{aligned}
 & (1 \times 2^7) + (1 \times 2^4) + (1 \times 2^0) \\
 &= (1 \times 128) + (1 \times 16) + (1 \times 1) \\
 &= 128 + 16 + 1 \\
 &= 145
 \end{aligned}$$

Binary representation is commonly used when programming microcontrollers because so much functionality is based on individual bits. Even though there may be a group of bits stored in a memory location, the processor or program may only look at certain bits to make a decision or manage a process.

An example is with something we call a flag register which is a memory location that a programmer might allocate to use the individual bits to track events in a program. Set bit 1 if a user pressed a button. Set bit 2 if data is ready from a peripheral. Set bit 3 if a certain event occurred. Keeping track of these three items is done simply by toggling the corresponding bit. If the program is being debugged and the programmer wants to know what flags are set, the memory location is displayed in binary so it will be easy to see what flags are set. The meaning of each flag bit is entirely independent of the other bits in the byte, and totally arbitrary based on the programmer's defined (and hopefully documented) choice.

7	6	5	4	3	2	1	0	POSITION
1	1	0	0	1	0	1	1	FLAG VALUE
NEW DATA	TIME TO READ SENSOR	UNUSED	UNUSED	FAULT DETECTED	NO INPUT	SYSTEM ACTIVE	MOTOR ON	MEANING

Figure 1-20 Example flag register

A problem with working in binary is that large numbers are difficult to look at. Some compilers do not recognize binary numbers. Therefore, it makes a lot of sense to use hexadecimal numbers. The EIE notation for a hex number is "0x" followed by the value, like 0x42. Some people prefer a trailing 'h' like 42h. We almost always shorten "hexadecimal" to just "hex."

The base-16 hexadecimal system counts from 0 to 9 and then uses letters A through F to count another 6 values before rolling to the next digit. Since a larger quantity can be stored in each digit position, hexadecimal numbers have fewer digits than decimal numbers. Conversion to decimal is done in the same way, this time multiplying powers of 16 by coefficients 0 thru F, where A, B, C, D, E, F are 10, 11, 12, 13, 14, and 15, respectively.

Consider the number 499 which in hex is 0x1F3. Break it down into the sum of the digits and powers of 16:

$$\begin{aligned} & (1 \times 16^2) + (15 \times 16^1) + (3 \times 16^0) \\ &= (1 \times 256) + (15 \times 16) + (3 \times 1) \\ &= 256 + 240 + 3 \\ &= 499 \end{aligned}$$

Converting hex to decimal and vice-versa is not done very often, but converting between binary and hex is. Hex to binary conversion is performed one hex digit at a time by inspection into groups of 4 bits:

$$0x1F3 = b'0001\ 1111\ 0011'$$

Binary to hex is done by grouping 4 bits starting with the LSB and simply reading the result. Work right to left, and pad zeroes to the MSB to 4 binary digits. It is helpful to make the 4-bit groups easier to see by adding spaces:

$$b'1100100111110' = 0001\ 1001\ 0011\ 1110 = 0x193E$$

On an 8-bit system, the maximum hex value is 0xFF, which corresponds to 255 or b'11111111'. Base-16 numbers are the most-used number system in embedded programming – even more than decimal. The nature of working with an embedded system makes this a necessity. The whole art of making a system work really boils down to moving bits around, and the best way to show a bunch of bits is in hex. Doing so allows the programmer to see memory locations exactly the way they are stored on the actual hardware. Hex is obviously more compact than binary, and most often hex representation will show you what you need. Going back to the flag register example, that same byte displayed in hex is easy enough to translate in your head to determine what flags are set. For example, how quickly can you tell which of the flags are set if you read “0x5” in a memory location? You should see that bits 0 and 2 are set in this case.

You will find that you use decimal, binary and hex throughout an embedded programming design and you will see how they each play important roles. Octal (base-8) numbers rarely come up in our experience, so we will purposely neglect to discuss them. You will develop some preferences in your debugging for what representation you like most for certain situations. Make sure you configure the debugger to display the values in the form that makes the most sense when you look at different variables.

Variable	Value	Location	Type
bInputDetected	FALSE	R0	bool
u32Counter	159900	R1	u32
u8Flags	0b10100100	R2	u8
u32RegValue	0x7A30F1B4	R3	u32

**Figure 1-21 Debugger display numerical values**

The faster you can translate between number systems the better, as you often need to determine “what a number means” quickly. For example, you might have a counter that you can see in memory that is shown in hex. Rather than changing the display settings, it would be quicker to know by inspection that “0x7f” is 127 and thus the counter’s 6th bit is about to overflow. Doing quick translations like that in your head is handy. Practice being really fast with values from 0 to 15.

**Table 1-1** Decimal, binary and hexadecimal conversion

Decimal	Binary	Hex
0	b'0000'	0x0
1	b'0001'	0x1
2	b'0010'	0x2
3	b'0011'	0x3
4	b'0100'	0x4
5	b'0101'	0x5
6	b'0110'	0x6
7	b'0111'	0x7
8	b'1000'	0x8
9	b'1001'	0x9
10	b'1010'	0xA
11	b'1011'	0xB
12	b'1100'	0xC
13	b'1101'	0xD
14	b'1110'	0xE
15	b'1111'	0xF
16	b'10000'	0x10



Windows Calculator is a very useful tool for embedded systems programmers. It has a “Programmer” mode that lets you quickly make conversions. We’d argue the Windows 7 version (left) is nicer than the Windows 10 version (right).



**Figure 1-22** Windows 7 and Windows 10 calculator

### 1.5.2 • Ohm's Law

One of the first electronic equations you should learn or probably learned already is Ohm's law. Even if you never design hardware, keep Ohm's law in mind if you are working with a piece of hardware to prevent yourself from doing anything that might break the board. If you do not know it, then be sure to add it into your repertoire of electronics skills. Ohm showed that:

$$V = IR$$

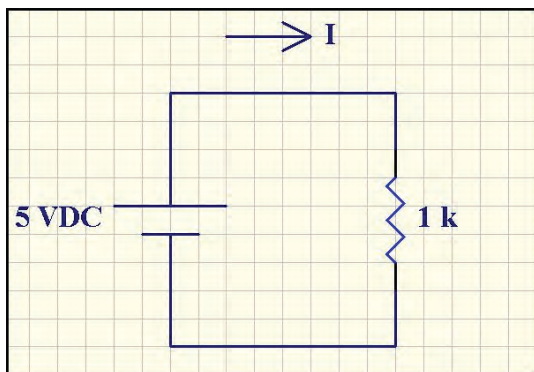
where,

V = voltage in volts

I = current in amperes

R = resistance in Ohms

A simple example of Ohm's law is shown:



Find current I in the circuit

$$V = IR$$

$$I = \frac{V}{R}$$

$$I = \frac{5V}{1000\Omega} = 5mA$$

**Figure 1-23 Finding the current in a circuit**

In most embedded applications that a firmware designer is working with, voltage and current are DC, so no time-varying signals have to be considered. Ohm's Law is linear and presents no challenges when using it to analyze a circuit. However, experience tends to show that everyone knows Ohm's law until they get into the lab and build a real circuit. Remember that even a piece of wire has a finite resistance, so Ohm's law still holds true when you accidentally short your power supply.

To avoid any human-based experiments, consider what happens if you take a wire and put it across a 9V battery. What would the voltage at the terminals measure? Or perhaps the better question, could you measure the voltage before the wire melts? The answer is that the battery itself has an internal resistance that is in series with the tiny resistance of the wire. The voltage of the battery will drop low enough to ensure Ohm's Law is holding true for the amount of current that the battery can deliver through the battery resistance and the load. This is amps of current, so it is not a good idea to try. Likewise, it is not a good idea to carry a 9V battery in your pocket with your keys.

This concept is one of the things that gets people new to electronics most often when adding a load to a power supply. As with the battery example, the load that you are attempting to power is fixed and has some resistance. Therefore, the current in the circuit is going to be set based on the power supply and load. There will always be some variation in the measured power supply output depending on the load current and how

the supply adjusts – therefore we have transient responses to changing loads.

Power supplies that limit current can do so in one of two ways: increase their impedance or lower their voltage. A shorted battery drops its terminal voltage if the load is trying to draw more current than the supply can source (internally, the voltage is dropping across the resistance of the battery cell). A good lab power supply will increase its resistance to limit the current to a safe level you set. In either case, the voltage you measure at the terminals of the supply will not be what you might expect because of Ohm's Law. In any case, even outside of the extremes, Ohm's Law is at work and the power supply is changing in internal resistance to maintain current. Even a "current source" supply (it is hard to come across a current source – most power supplies look like voltage sources) will operate following Ohm's Law and will crank up the voltage to maintain a constant current. LED drivers are a great example of this.

Keep these points in mind when you are building circuits:

1. Make sure you know all the V's and R's in your circuit before you turn the power on. Do you have any low-impedance paths to ground? LEDs without current-limiting resistors are good examples.
2. ALWAYS limit the power supply current to your device. Rarely does a microcontroller circuit require more than 100 mA. If you limit your supply to this amount, the chance of accidentally damaging something in your circuit or in the supply itself is minimal even if you have dead shorts to ground. If you do not know how to current limit a power supply, ask!
3. Remember your load "draws" current from the supply. If you try to draw too much current, the power supply voltage will likely drop as Ohm's law prevails!

## 1.6 • Switches

Ohm's law is a nice segue into some fundamental hardware parts. Again, even if you never design embedded hardware, you will benefit from this rudimentary understanding if you program drivers and interfaces. Being able to see the code you write affect change in the real world could arguably be one of the most gratifying parts of embedded systems development. If you are not careful, you can damage the hardware.

The first way to blow things up is with switches that you hook up to a circuit. Why? Because Ohm and his law are forgotten by people who want to quickly get their circuits going! Well, it is time to be engineers and think through everything in a design before turning the power on.

Switches are essential in circuits to supply digital inputs to the processor. One very common switch is the toggle switch, another is the momentary switch (which is more like a button) and a third is the slider switch. Typical schematic symbols are shown for each.



Figure 1-24 Typical schematic symbols for switches

Toggle and momentary switches change a circuit from an open to a short. Toggle switches will stay put when you switch them, whereas momentary switches will close the circuit only while you are pressing it (there are momentary switches that are normally closed, and open when pressed). The slider switch selects one of two or more inputs and routes it to a common terminal or vice-versa – kind of like a mechanical multiplexer.

So how can you possibly blow up your circuit with just a switch? As mentioned, switches are commonly used to provide a digital 1 or 0 to a processor input. Many people have

proceeded to hook up one of these two circuits:



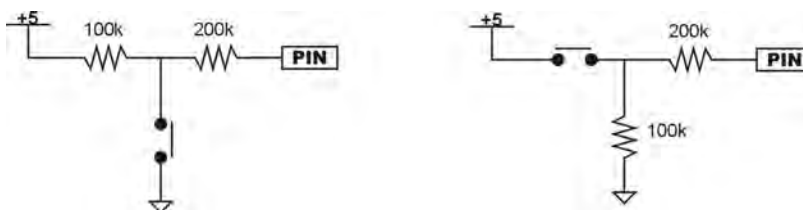
**Figure 1-25 Incorrect circuits for button input**

The principle of operation of the circuit on the left is simple – when the switch is open, no power is applied to the pin, so it is logic 0. Press the button, and the PIN is brought to logic 1. Though this might make sense, what you have is an undefined state when the switch is open. This is because the input to the processor pin is floating, so because of stray capacitance in the circuit and a variety of noise sources, the pin may be high, low, or somewhere in-between. You would model this by showing a capacitor to ground from the pin. The input to the pin is very high impedance (on the order of gigaohms), so the capacitor cannot drain its charge. When the switch is pressed, it is okay, since the PIN is tied high. When released, the parasitic capacitor would maintain the high signal. It also happens that the nature of CMOS circuits (complementary metal oxide semiconductors – the type of transistors the processor is made of), tend to float to logic high. The bottom line is that if you tried to use this circuit, you would get many erroneous high signals. The solution: always ensure your digital inputs are tied high or low.

So how do you hook up a switch to solve this problem? Could you satisfy the “tie high or low” requirement by hooking up the circuit shown on the right in Figure 1-25? Now the PIN is tied low when the switch is open and is pulled high when the switch is closed, right? If you hooked this up in the lab, you would probably be quite satisfied with your work, but is there a problem? Ohm’s law! Assume your switch has an ON resistance of 0.1 Ohms and your supply voltage is 5V. You now have 50 amps flowing through your switch. If your power supply could source that, your finger will be rather hot, and things will melt. Hopefully, you are using a power supply with current limiting, but what if you were trying to build a car alarm and were using your car battery as a power source? Ouch!

Of course, there is a simple solution: use a “pull-down” resistor. This keeps the pin tied low when the switch is open, so it does not float and supplies a high-impedance current path to the circuit when the switch is closed. Similarly, if you wanted the switch circuit to be normally high (i.e. high when the switch is open, low when closed), you can move things around and run the switch with a “pull-up” resistor.

The schematic shows the ideal normally high and normally low type switch circuits with 100k pull-up and pull-down resistors. Ignore the 200k resistors for just a moment and remember that the input impedance of PIN is very high.



**Figure 1-26 Normal high and low type switch circuits**

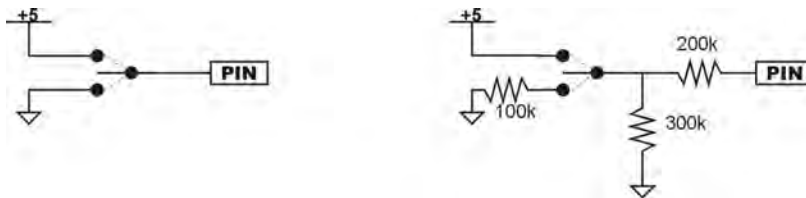
For the normally high circuit on the left, no current is flowing anywhere when the switch is open, so PIN sees logic high. We say it is “pulled up” or “tied high” by the 100k resistor, hence the term, pull-up resistor. When the switch is closed, PIN is pulled low. Now,

however, instead of having a short to ground, the power supply current runs through the 100k resistor. A high valued R is used to minimize the current flow and save power. 10k to 1M is a good range for a pull-up/down resistor. Anything smaller than 10k is usually a waste of power. If you go too small you are back to the same problem as the shorted path! Anything above 1M is too resistive to effectively drain the parasitic capacitance that is holding a charge and your signal will switch slowly or not at all if the input is fast.

Similarly, the normally low circuit on the right uses 100k to pull PIN low when the switch is open. Close the switch, and PIN sees 5V and a low-current path exists through 100k to ground.

The 200k resistors are there as it is also a good idea to put a resistor in series with the PIN. 200k is chosen arbitrarily and could be anywhere in the range 1k – 1M to accomplish its mission in this case. Even though the PIN has high input impedance, you could change the pin with your firmware to an output that might be high or low and therefore run the chance of shorting to ground. The worst part is that this current would be through PIN and thus through your processor which could be damaged. In the application above with the high-impedance input pin, no current flows in the 200k resistor, thus no voltage is dropped on the 200k resistor, the digital signals to PIN are preserved and you are protected in the event of an accidental input/output change. Designing in a resistor like this would only be done in critical systems. It doesn't cost much, but the board space and routing the PCB would be significant.

As for the slider switch, you might think that you could use it safely with power and ground attached directly to each of the selection points. This depends on the switch – some switches may short both inputs together for a moment when the switch is changed. Or, both inputs may be disconnected temporarily during the switch, thus your pin will be floating for a short period of time. This short period of time may only be a few milliseconds, but if your processor is running at MHz speeds, you could have literally hundreds of false switches when you change the input. The recommended circuit when using a selection switch to choose between power and ground is shown.



**Figure 1-27** Recommend circuit when using a selection switch

In the recommended circuit, 100k prevents power from shorting to ground during a switch; 300k prevents the switch from ever floating; 200k prevents an output problem if the PIN ever was an output high or low. Again, these values are arbitrary and chosen so we can reference them easily. In a real design, all three would be the same value. If using the switch to select signals from other circuits, be sure to consider the hardware being attached.

### 1.7 • Light Emitting Diodes (LEDs)

Practically every consumer electronic device these days has at least one LED in it to show power or some other status. LEDs are becoming more and more popular in all areas of our lives, and new technologies continue to make them brighter, more efficient and available in more colors. LEDs are making their way into vehicles, and are increasingly used for commercial and residential lighting applications.



LEDs also happen to be a very important element in the life of the experimenting microcontroller fanatic. They are the perfect way to quickly check the state of an output pin, and you can put them to use in hundreds of different applications. You can also do small projects that look really cool.

What is an LED and how does it work? A light emitting diode is just like a regular diode except that it happens to emit photons in the visible light spectrum when current flows through it. To understand how a diode works, you must get into a lot of semiconductor theory, which is way beyond the scope of this course and the bounds of practical engineering. What you need to know about LEDs and normal diodes is that they behave like a current valve – current can flow one way but is blocked in the other direction. Of course, Ohm's Law still applies.

The schematic symbol is shown with some terminology related to the diode. A diode is a two-terminal device. The positive terminal is called the "anode" and the negative terminal is the "cathode." Current can only flow from the anode to the cathode. For some reason, diode datasheets and schematics typically use "A" and "K" to denote anode and cathode. You would be correct to point out that the relationship of anode and cathode to positive and negative is opposite to a battery where the positive terminal is the cathode.

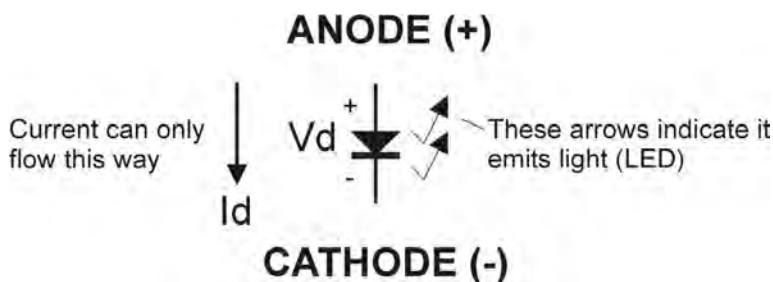


Figure 1-28 Function of the Light Emitting Diode (LED)

A diode begins conducting current when it is "forward biased" – that is, when a positive voltage,  $V_d$ , is applied across it. Every diode requires a certain amount of bias before it begins conducting and in steady state. When it is fully on, the diode will have a fixed voltage drop across it, called the "forward voltage" or "diode voltage" typically denoted  $V_d$  or  $V_{fwd}$ . For example, a common rectifying diode like a 1N4001 has a forward voltage of 0.7V. A red silicon LED voltage is usually around 1.7V, and many blue or white LEDs are around 3.6V. These varying forward voltages are all functions of the different materials that the diodes are made from. Note that most LEDs have a maximum rated constant current of 20 mA – if you run more than this through it, it will fry. The forward voltage and maximum current rating are always specified on the diode's datasheet, so check that out during the design phase.

Once enough bias has been applied to the diode to turn it on, the diode will generally hold its forward drop regardless of current,  $I_d$ . This might sound contrary to Ohm's law, but the diode's internal resistance continues to change with current. A typical voltage-current curve for a diode is shown. You can calculate the diode resistance at any given point directly from this graph by taking the inverse of the slope (Ohm's Law at a given voltage!).

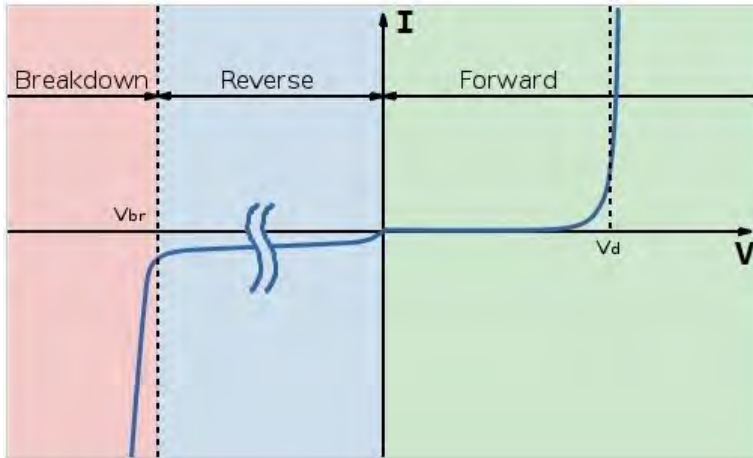


Figure 1-29 Typical voltage-current curve for a diode

Obviously, diodes are non-linear devices. The range of operation of a diode goes between “breakdown” on the far left where too much reverse bias is applied, through an “off” state when not enough forward or reverse bias is provided to get any appreciable current flowing, and then into an “on” state where the diode is forward biased and conducting current. The voltage asymptote is the maximum forward voltage that the diode will drop no matter how much current is flowing through (until it melts) and is what embedded designers usually care about.

Understanding a diode’s properties in the region of operation that we are concerned about allows us to model a diode that is *fully* on like a voltage source in a circuit for analysis. Not that the diode is able to source electrons like a battery, but its voltage drop behaves like a DC supply in analysis. You can improve the model by including the diode resistance, but it is easiest and usually sufficient to use only the diode’s forward voltage for a digital circuit.

Mostly we will use single LEDs set up to be on when you want them to be lit up. All you must do is pick the correct resistor to limit current. To analyze a diode circuit to pick that current limiting resistor, use the basic diode model and Ohm’s law.

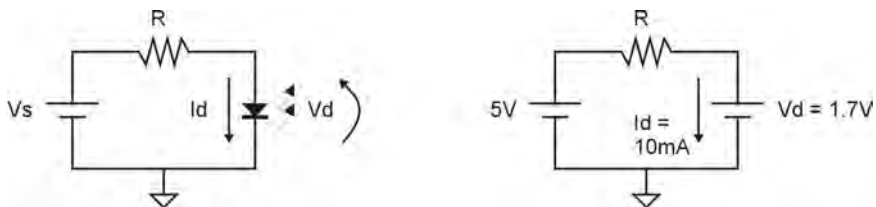


Figure 1-30 Single LEDs set up

We want to find  $R$  to set  $I_d = 10\text{mA}$ , which is a good amount of current for an LED unless the datasheet says otherwise. The diode in the circuit is a red LED with  $V_d = 1.7\text{V}$ , and the voltage source is  $V_s = 5\text{V}$ . The right side of the Figure shows the diode replaced with its model for analysis. By Ohm’s law:

$$R = \frac{V}{I} = \frac{V_s - V_d}{I_d} = \frac{(5 - 1.7)}{10e^{-3}} = 330\Omega$$

Note that if  $R = 0$ , our model would have  $I_d = \text{infinity}$  (theoretically). The diode has some resistance so the current would be finite but high enough on a strong enough 5V supply to cook your LED. Exciting as it sounds, what really happens is that the light goes out and sometimes it makes one small clicking sound. Either way, you have one less LED to use.

Many microcontrollers are limited in how much current each pin can source. The limit is often below 1mA, so you need to be careful about what circuit you design. A milliamp of current is ok for an indicator LED, but for a brighter light, you need more current and thus require an additional driver circuit.

## 1.8 • Transistors

Like diodes, transistors are semiconductors and have non-linear voltage-current characteristics. Also, like diodes, there are some complex equations to describe transistor behavior if you want to quantify everything that they do. Fortunately, transistors used in embedded systems often fill a few simple roles: digital switches and high-current drivers.

There are two types of transistors we need to worry about: Bipolar Junction Transistors (BJT) and Metal Oxide Semiconductor Field Effect Transistors (MOSFETs or simply “FETs” for short). BJTs come in “NPN” and “PNP” which refer to how the semiconductor material is put together and results in different behavior. MOSFETs come in NMOS or PMOS, which also refers to the way they are built with the semiconductor. The “N” and “P” in the two types of BJT and MOS are referring to the same construction characteristic of the material from which they are formed. If you would like more information, Wikipedia has a good discussion on both transistor types.

Transistors are three-terminal, non-linear circuit elements. Bipolar transistor terminals are the base, collector, and emitter (B, C, E). MOSFETs have a gate, source, and drain (G, S, D). It is reasonably safe to say that the base of a BJT and the gate of a MOSFET are analogous, likewise for emitter/source and collector/drain. The schematic symbols for the four types are shown.

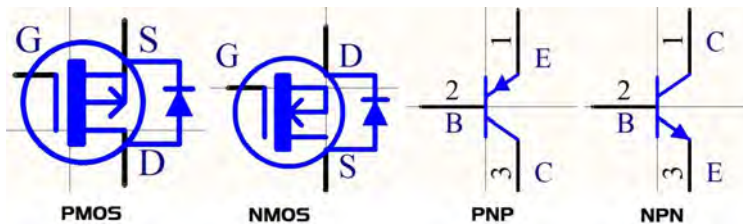


Figure 1-31 Schematic symbols for MOSFETs (left) and BJTs (right)

Transistors are good when it comes to switching high current loads or dealing with voltages in circuits that your microcontroller cannot switch directly. The micro can be used to control the transistor to tell it to be on or off but is isolated from the current that the transistor is switching.

An NPN BJT is used as an active-high, “low side” switch. In other words, if you want to drive a point in your circuit to ground or otherwise have it floating, you can use an NPN BJT. The voltage at the point you are switching can be higher than your processor voltage because the higher voltage will not feed through to your driving voltage. So, this circuit is perfect for switching high-voltage LEDs for example. You need the transistor and a resistor to do this, set up as shown.

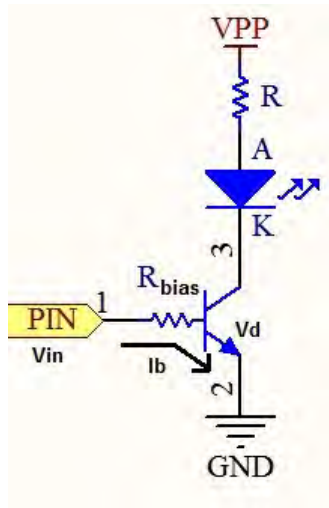


Figure 1-32 Switching high-voltage LEDs

This circuit will draw current through the resistor and the diode to ground. The amount of current from the processor pin is found simply with Ohms law on the path from the pin through the diode portion of the transistor to ground:

$$I_b = \frac{V_{in} - V_d}{R_{bias}}$$

The diode voltage in a BJT can be assumed to be 0.7V and you can pick a high-value resistor like 100k for the base bias resistor, so you do not waste current through the base-emitter diode. The current between the collector and emitter is a function of the base current and the “gain” of the transistor. For a quick approximation, you can assume the gain is about 100. If you want a bright LED, then you still need some amount of base current. If you do not put a resistor in, then our friend Ohm will try to set  $I_b$  as close to infinity as possible which will probably blow up the output driver in your processor. Many BJTs have a built-in base resistor so you do not need to add your own. These are commonly referred to as “digital NPNs” and they are purpose-built for this application.

A PNP BJT works essentially the opposite and is used as an active-low, “high-side” switch. This time, the base voltage must be lower than the supply minus the diode drop to turn the transistor on and the processor will sink current as shown.

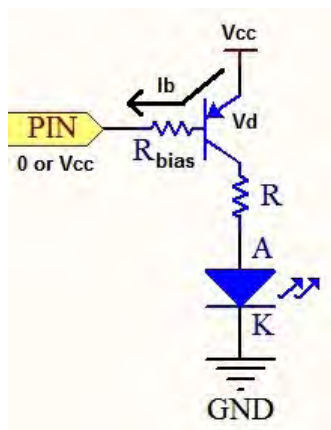


Figure 1-33 Active low switching circuit

You must be careful that the transistor  $V_{cc}$  voltage is not higher than the processor supply because this application does not isolate the switched voltage from the processor pin. To turn the PNP switch off, the processor voltage and voltage at the emitter should be the same ( $V_{cc}$  in this case). Like the NPN, there will be a small current flowing (assuming you have a large bias resistor) when the switch is turned on (i.e. the PIN voltage is low) and the emitter-collector current is a function of the transistor gain. With enough base current, the collector and emitter will be essentially shorted which switches in the supply voltage to the circuit.

$$I_b = \frac{(V_{cc} - V_d) - \text{Pin voltage}}{R_{bias}}$$

Remember, the base current will flow into the pin when the pin is driven low. If you raise the pin voltage to logic high ( $V_{cc}$ ), the base voltage is at the same level as the emitter voltage, so the diode will be off ( $V_d$  is 0) and the transistor is off. If you incorrectly design a PNP high-side switch when the transistor voltage is higher than the processor voltage  $V_{cc}$ , you will be unable to turn the switch off because the base-emitter diode will always be in its active mode which means the transistor is on.

Implementing MOSFET switches is similar. An NMOS is a low-side switch and a PMOS is a high-side switch. An interesting thing about a MOSFET is that it is almost a symmetrical device: you can swap the source and drain connections in the circuit and it will almost behave the same. Almost. The asymmetry extends from the construction of the transistor, which results in a parasitic diode between the source and drain. This is not a separate diode built onto the device on purpose. It is just there because of how a MOSFET is built. The parasitic diode should always be shown in schematic symbols and it is extremely important to orientate the transistor correctly in your circuit. Every time you design a FET into a circuit, be sure that you are conscious of the orientation of the parasitic diode. Current will flow through the diode when you do not want it to if it is backward, and the circuit that you are trying to switch will be powered when you think it is off.

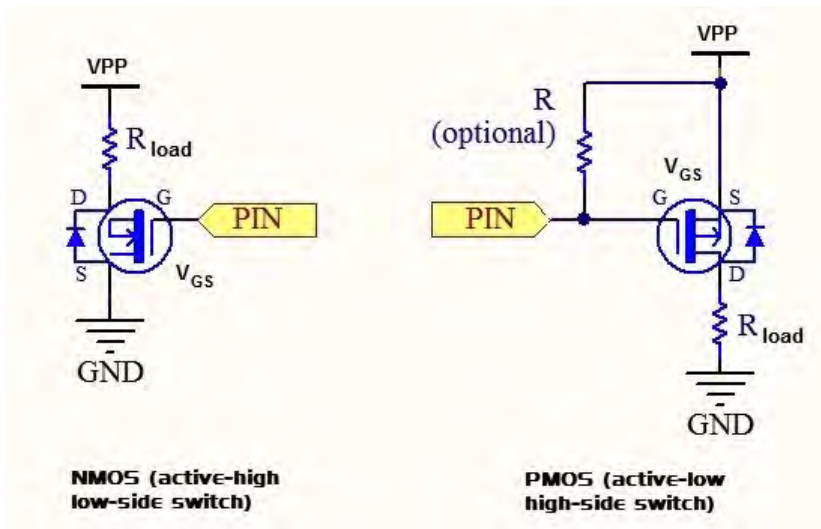


Figure 1-34 NMOS switching and PMOS switching

Another big difference between a FET switch and a BJT switch is that the terminal used by a processor to control the FET (the gate) is high-impedance, so unlike the BJT, there is no current flowing when the switch is active. This is true for both the NMOS and PMOS FETs. Since the gate is high-impedance, you do not need a bias resistor to limit current.

To turn the FET fully on, the voltage between the gate and source terminals ( $V_{GS}$ ), must reach a certain potential difference. This is critical to the operation of the FET and will be a specified parameter on the part's datasheet. It is not something you typically need to worry about when using FETs in a digital system as  $V_{GS}$  achieved from processor logic levels will always be plenty to activate the FET.

What you do need to worry about is that you can get  $V_{GS}=0$  to turn the switch off (regardless of PMOS or NMOS). The NMOS case is not usually a problem because you set the pin to common ground. Most processors can do this but not all. The PMOS case may require a pull-up resistor because it is often difficult to get a processor output all the way to the supply rail. So instead of driving the switch with logic high you use the processor pin's high-impedance state to remove the processor's drive to the gate and let the pull-up resistor do the work. Ohm's Law will show you why that works – remember there is no current flowing into the gate and no current flowing in or out of the processor when it is in high-impedance mode, therefore there is no drop across the resistor. The gate voltage will be exactly the source voltage, and by now you know that when  $V_{GS}=0$  the transistor is off!

The only thing you must worry about when using a pull-up is that the voltage to which the gate gets pulled up to is not higher than the input voltage rating on the processor pin. This lack of voltage isolation could activate the ESD protection diodes that live in most processor pin drivers which would result in some unwanted current flow through the pull-up resistor. This, in turn, would cause a small voltage to drop so  $V_{GS}$  would no longer be zero and the switch would turn on (at least partially).

The characteristic to look for on the processor's datasheet to determine if you need pull-up or pull-down resistors will be "rail-to-rail" output drivers. Or just find the parameter for  $V_{out\_low}$  ( $V_{OL}$ ) and  $V_{out\_high}$  ( $V_{OH}$ ) to check if they are the supply rails. If you are unsure, you can always add pull resistors in the circuit with barely any cost or space consequence. As a bonus, having the resistors ensures known states to the FETs even

when the processor is powered off or when the system is just starting up since practically every microcontroller will start up with all its GPIOs in the high-impedance input state.

In either case, the pull-down resistor value is chosen quite high to avoid high leakage currents. A common value is 100k but very low power designs will use 1M or higher. If you go too high, the switching time of the FET becomes too slow because, just like with any MOSFET device, there will be parasitic capacitances that need to be charged or discharged before things start changing state. There will always be some leakage currents at the processor and at the transistor gate, so too weak a pull-up will result in the switch being on slightly.

If you have been following all this transistor talk, you might notice that there is no solution that allows a high-side switch to be implemented if the voltage you are switching is higher than your microcontroller supply without worrying about activating ESD protection diodes. Fear not - you can make use of both an NPN BJT and a PMOS together to accomplish this task. This is a very useful circuit.

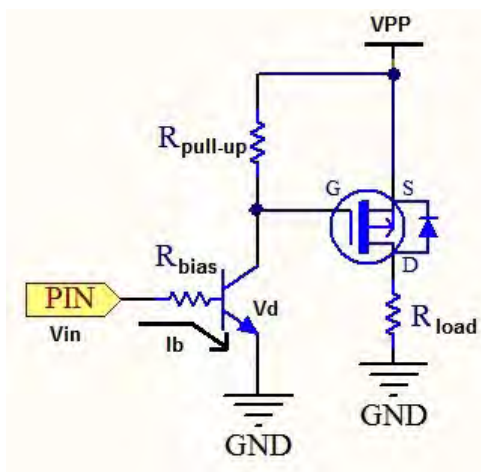


Figure 1-35 NPN BJT and PMOS circuit

Here, the processor drives the BJT so it does not see the high voltage  $V_{PP}$  in the circuit. In turn, the BJT drives the FET's gate. When the processor pin is logic high, the BJT is on which grounds the PMOS gate and turns the FET on, so the high-side switch is closed. When the processor drives low, the BJT collector-emitter channel is open so the gate is floating, except the pull-up resistor pulls it high which turns the PMOS off. This circuit is an active-high high-side switch. This circuit is commonly used in designs to get around situations where switching is required but supply rails are at different levels.

## 1.9 • Power Supplies

One more critical element that you need to have to run your embedded system is a power supply. Though a resistor or diode can run on any voltage, integrated circuits are usually designed to operate only over a specific range. Many microcontrollers will only operate from a supply voltage of 4.5 to 5.5 VDC, though most these days for embedded applications run at 3.0V – 3.6 VDC and some are now down in the 1.8V range. Throughout the operation of the device, the supply cannot extend outside of these ranges or you risk damaging the processor or at least cause it to run improperly.

The magic +3.3VDC that is shown everywhere in your schematics must come from



somewhere. You need a source of electricity, but of course, you cannot just plug your development board into a wall socket. Even a trusty 9V battery will not help you out here on its own. You need a 3.3VDC source and you need that source to be regulated to maintain the voltage regardless of the load.

It is nice to have a bench-top DC supply that is adjustable and stable, but these can cost literally thousands of dollars. They are not very portable, and you would not want to build a product that requires 10 pounds of power supply and a very long extension cord to go with it. So where else can we get some well-behaved electrons?

Wall transformers (AC/DCs) and batteries are the hobbyist's primary sources of DC power because they are inexpensive and readily available. USB chargers for phones are everywhere, and they are regulated to 5V. You probably have at least one lying around, so that's a great option. Be careful of old transformer-based "wall-warts" that do not have regulated outputs.

In most electronic systems, engineers devote a lot of time to developing stable and reliable power supplies for their circuits. These are systems that can take power from something like a battery or transformer, and output a fixed voltage on which the application circuit will operate. They must maintain the output voltages regardless of the load they are driving – Ohm's Law, Ohm's Law, Ohm's Law. There are now many integrated circuits called voltage regulators that can take a variety of input voltages and output a consistent voltage like 3.3 volts. All we must do is read a datasheet, add a few supporting passive components, and we have the power we need!

You can get away with some fantastically cheap ways to set the voltage in your circuit, but your choice must consider several things. First, think about the load itself. Is it going to draw a lot of current or very low current? Does the load vary in time as the device runs? Second, consider the quality of regulation you need. Can the circuit tolerate a slight ripple in the output voltage, or must it be ultra clean and quiet? Third, consider your input source. Is the input voltage close to the output voltage? Do you require high efficiency, or can you afford to waste some power? Are you reducing the input voltage, or raising it up to a level you need? Is it AC or DC? Lastly, consider the cost of the power supply.

There are many parameters to consider when designing a power supply – whole courses are devoted to this. We will touch on the very basics to get you started.

### 1.9.1 • Linear Regulators

Linear regulators are the most common type of voltage regulator. They behave like voltage-controlled variable resistors. Inside they contain a transistor and reference voltage. The output voltage is fed back to the regulator, compared against the reference, and the resistance of the transistor is adjusted to minimize the output error. Because there is feedback, linear regulator outputs always have a short transient response to changes in the load. Linear regulators do not introduce additional noise to the system, and many are able to reject noise from the input supply. They are low cost and do not require a lot of additional components. Their major downfall is that they simply burn off the power between the  $V \times I$  of the input and the  $V \times I$  of the output. The input current essentially equals the output current aside from a small operating current to run the IC itself.

There are thousands and thousands of different linear regulators available. An important parameter to consider when choosing a linear regulator is called the "drop-out voltage." This is the minimum difference between input and output voltages for the regulator to work properly. Old linear regulators had dropout voltages around 2V or higher. As system voltages got lower, this dropout voltage became very inconvenient. Industry responded by creating "low drop out" regulators that are now ubiquitous. Engineers call them



“LDOs.” LDOs can have dropout voltages as low as 50mV, so they are very useful for USB-powered or battery-powered applications with 3V or 3.3V Vcc requirements as they stay in regulation despite relatively low input voltages.

One of the most common regulators is the LM7805 voltage regulator so it makes a great example. It has been around forever and is now quite obsolete by modern standards but still widely used. Practically every IC manufacturer has their own version of the 7805, and most of them are exactly interchangeable. The 7805 can take any voltage from 7 to 25V as an input and will output a constant 5V independent of the load. The devices come in hefty packages with large heatsinks and are capable of sourcing 1A of current. They are also offered in a variety of voltage outputs including an adjustable output (which requires a slightly different connection schematic).

The 7805 is easy to use and almost as simple as taking it from its package and plugging it in. To use any regulator, simply follow the recommended circuit in the datasheet. The 7805 requires two external capacitors for stability and proper connections for input, output, and ground. Adding more output capacitance will improve transient response.

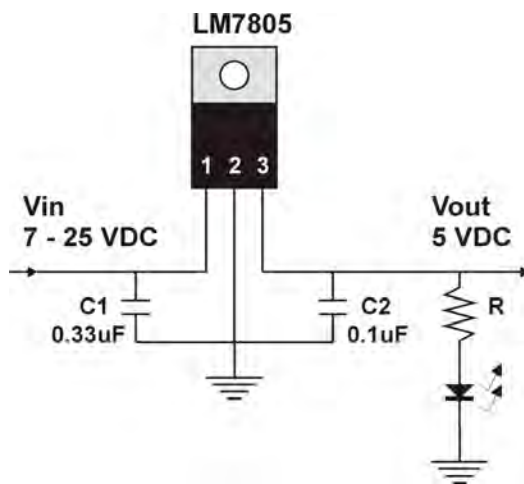


Figure 1-36 LM7805 voltage regulator

The input voltage can be a wall transformer or 9V battery – anything can be used if it is at least 7V. If your input voltage is less than 7V, the 7805 will “drop out” of regulation, hence your output voltage will not be 5V. Also note that the higher your input voltage, the more power the regulator must “get rid of” to maintain a 5V output. This is because the 7805 is still a linear regulator, and it changes its internal resistance to burn off energy that is not used by the load. The extra power is dissipated through heat loss:

$$P_{loss} = (V_{in} - V_{out}) \cdot I_{load} = (9 - 5) \cdot 5mA = 20mW$$

Since the 7805 is almost always dissipating power, it tends to get quite warm which is why it has a large heat sink built in. It may feel very hot in some cases, but if you are running the device within its specifications, you have nothing to worry about (just watch your fingers!).

For any design regardless of the regulator used, it is a good idea to hang an LED off the output to ensure that your regulator is working properly or to remind yourself that your circuit is “hot.” Do not forget the current limiting resistor for the LED! Pick R such that current is about 1mA, so you do not waste a lot of energy. If it is a battery-powered circuit, even 1mA is a lot of current in which case you would either omit it, crank up the

resistance, or maybe put a switch in.

With all those choices, how do you ever find one for your design? Engineers tend to have favorite manufacturers and project managers tend to like things that cost less, so most hardware designers probably have a tool-box of regulators they use ranging in capability, performance and price. If you find one that works well, you tend to stay with it until some reason to change comes up. The Texas Instruments TPS series of LDOs is a favorite high-quality part. The EiE development boards use an LDO from Microchip that is good quality, low cost, and several other vendors make a drop-in replacement. Having multiple sources on parts is ideal in case someone runs out. Output voltage can be set anywhere from 1.8V to 5V by varying the feedback resistors, which allows the flexibility of using the part in many different circuits.

### 1.9.2 • Switching Power Supplies

In addition to linear regulators, there are also regulators called switching power supplies or simply “switchers” or DC-DC converters. Unlike a linear regulator, switchers “sample” the input voltage to keep a capacitive or inductive “tank” full of electrons and thus maintain an output voltage. They literally switch a big transistor that connects the input voltage to the output at kHz or MHz speeds to make this work and only take the electrons they need from the input supply to refill their tank. The transistor is often built-in to the IC, though many come with leads to which you connect an external transistor – it depends on your needs. Because the on-resistance of the switching transistor is very low, switchers do not waste a lot of energy in heat like a linear regulator. For supplies that regulate down, the input current will be less than the output current. Really good devices can achieve efficiencies in the 90% range. They can often run from very high voltage input and some will not even care if the input is AC or DC – this is how the new style wall warts are implemented.

Switching power supplies that reduce high voltages to lower voltages are called “step-down” or “buck” converters. Some switching power supplies can provide output voltages higher than the input voltage by switching capacitors (“charge pumps”) or switching inductors (“boost converters”).

This schematic shows an LM25576 buck converter.

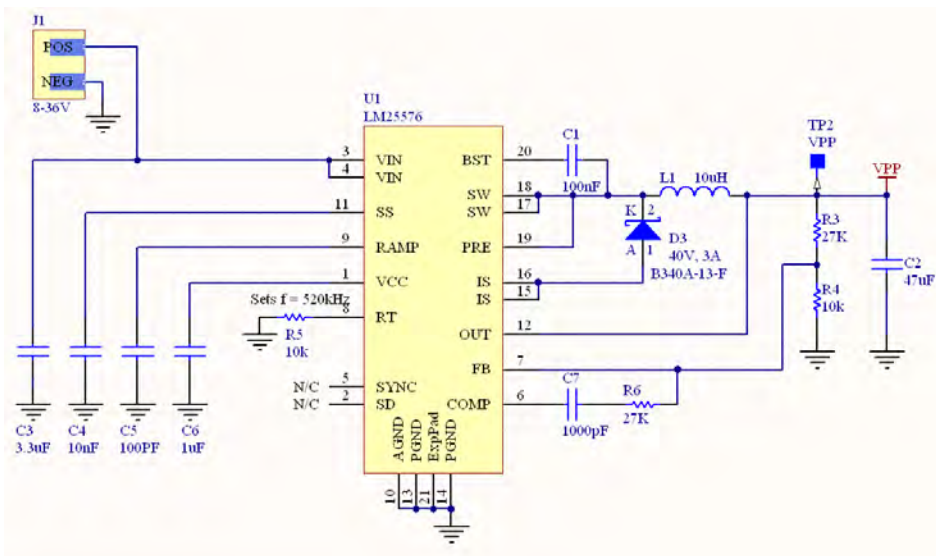


Figure 1-37 LM25576 buck converter

The PCB layout for a switching power supply is critical for a good system. Many switchers have reference designs in their datasheets that include layout recommendations and examples. Follow these carefully! Development boards of just the power supply circuit may be available from the vendor and it is highly recommended to order two: one for testing and modifying and one for baseline reference. Test it thoroughly with the components you want to use and the load you intend to power to see how it behaves. If you are happy with it, design it in and then repeat the tests with your layout and board to compare noise and response performance. A practical device tends to perform worse than the reference design that will be as close to ideal as possible.

The tradeoffs for using a switching supply come from the complexity of the integrated circuit which is passed on to you as extra cost along with quite a few supporting components that take up board space and add cost. Perhaps the worst characteristic of switching power supplies is the output voltage ripple and electrical noise that can be induced into or out of your circuit because of the switching action. It is not too difficult to get a switcher to work, but getting it to work well enough to pass FCC certification can be extremely difficult. It takes careful engineering to find and reduce noise in a circuit.

Many circuits will use a switching power supply to step down a high voltage to an intermediate level, then use an LDO to power noise-intolerant circuits. What type of regulator to use comes down to careful planning, a solid understanding of the system you are building, budget, board space, performance requirements, power consumption and in some cases, personal preference.

### 1.10 • Development Board Hardware

Now that you are armed with some fundamental knowledge of basic embedded systems hardware components, we can do a basic introduction of the development board schematics and tell you the must-know parts of the design. It is highly recommended that you print a hard copy of the schematics for your development board, as you will refer to them frequently as you write code in this program. All schematics are easily accessible on the program website – be sure to match your board version with the correct schematic file.

For every part symbol on a schematic page, there is a corresponding physical part on the development board. Parts are identified by “designator” which matches on the schematic and PCB. For example, look at the first connector on the first schematic page, its designator is “J1.” You can then find that part on the development board. A “bill of materials” or simply “BOM” provides all the details of that part such as who makes it and the specific part number you would use to order it.

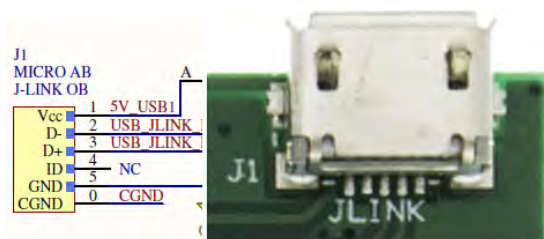


Figure 1-38 J-LINK pinouts

The numbers on the lines coming out of the part refer to the pin number. J1 has 5 output pins that you can see in the photo. “CGND” is marked as the 0th output – this is the chassis ground for the connector. Each of the outputs carries a specific signal. We know what these signals are because they are part of the standard for USB. In this

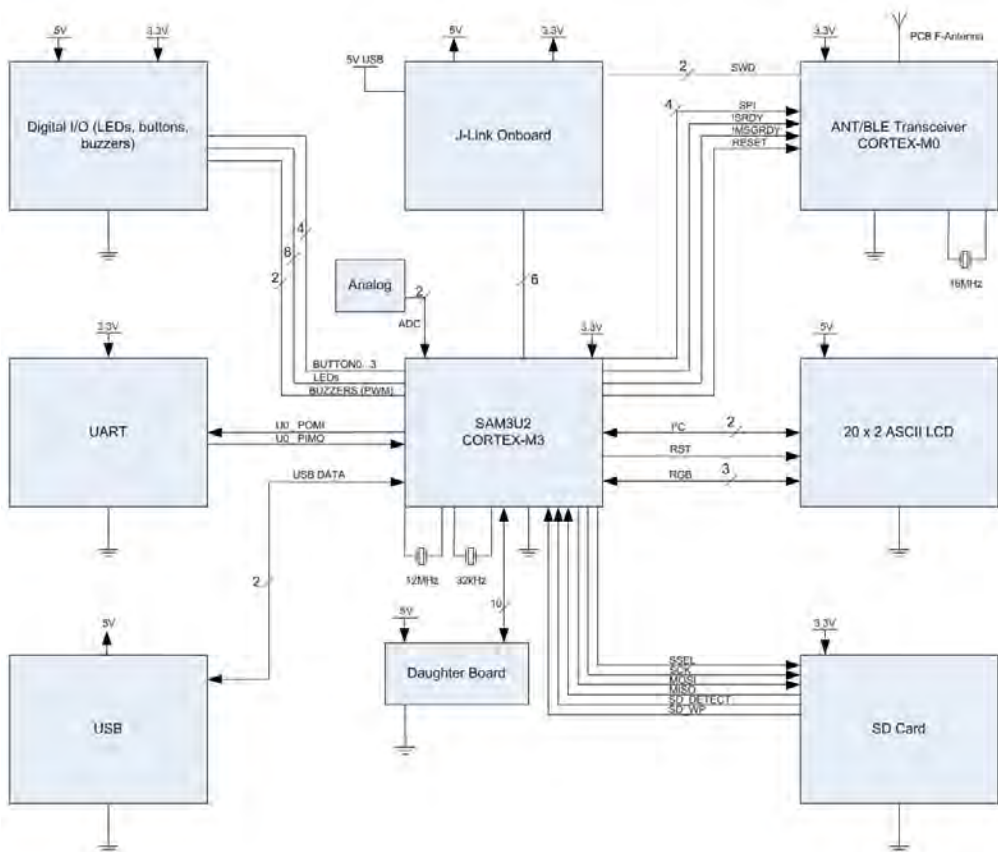
case, we have 5V power, two communication lines (D+ and D-) and a ground line. Pin 4 is an optional ID line that is not connected in this design and thus marked "NC" on the schematic. No matter what the part, there will be some sort of document or datasheet that explains how to use it, though it is up to the engineer to choose parts and implement them correctly.

### 1.11 • Block Diagram

Designing the hardware for an embedded system takes careful planning. Just like there are good processes and best practices to follow for firmware design, hardware design involves a sequence of steps to turn an idea into a real circuit.

The goal of the EiE ASCII development board was to provide a robust set of features that would enable developers to test a wide variety of circuits that are very typically found in embedded systems. We wanted the main board to be much more than just a processor breakout like an Arduino or ST Discovery board. Yes, those are fantastic platforms on which to build, but it would be difficult to meet the goals of the EiE program with something so basic. Still, not every feature can be included so having a means for expansion is essential. Our "Razor" development boards all features standard "Blade" daughter board connectors.

The final block diagram for the development board is shown. From here, each circuit was designed and captured in the project schematics and eventually laid out on the circuit board.



### 1.12 • Power Input

The first schematic page with components is simple and captures all the ways to power the development board. There are two Micro USB connectors. J1 is used to connect a PC to the J-Link Onboard programming chip so you can download and debug code through the integrated development environment. J2 directly connects to the SAM3U2 processor's USB peripheral. Use this port for the USB exercises or any USB functionality you code. Both J1 and J2 will power the board and it is safe to connect both at the same time with any 5V USB Micro B connection. If you need the 5V supply rail to truly be 5V, then connect 5V to J2. J1 will have a diode drop of about 0.3V.

The other thing to note from this page is that there are three power “rails” on the board: 5V (VPP), 3.3V (Vcc) that runs most of the systems on the board, and VBAT from the optional coin cell. You do not need to worry about VPP and Vcc unless you intend to build some new hardware in the prototyping space or via the daughter board connector. VBAT is purely for backup power when the system power is otherwise disconnected. It is connected to a special pin on the SAM3U2 that powers the “backup domain” which generally just provides voltage to retain RAM but not run the core. If you ever needed a device that could maintain memory if the power went out, then this is how you could do that. For example, if you built a fancy alarm clock application and wanted to make sure that the time would not reset if the power went out. By default, the development boards do not come with a coin cell or connector as we do not use it for anything in the program and coin cells are notorious for getting swallowed by (usually young) people. Please be careful if you decide to use these.

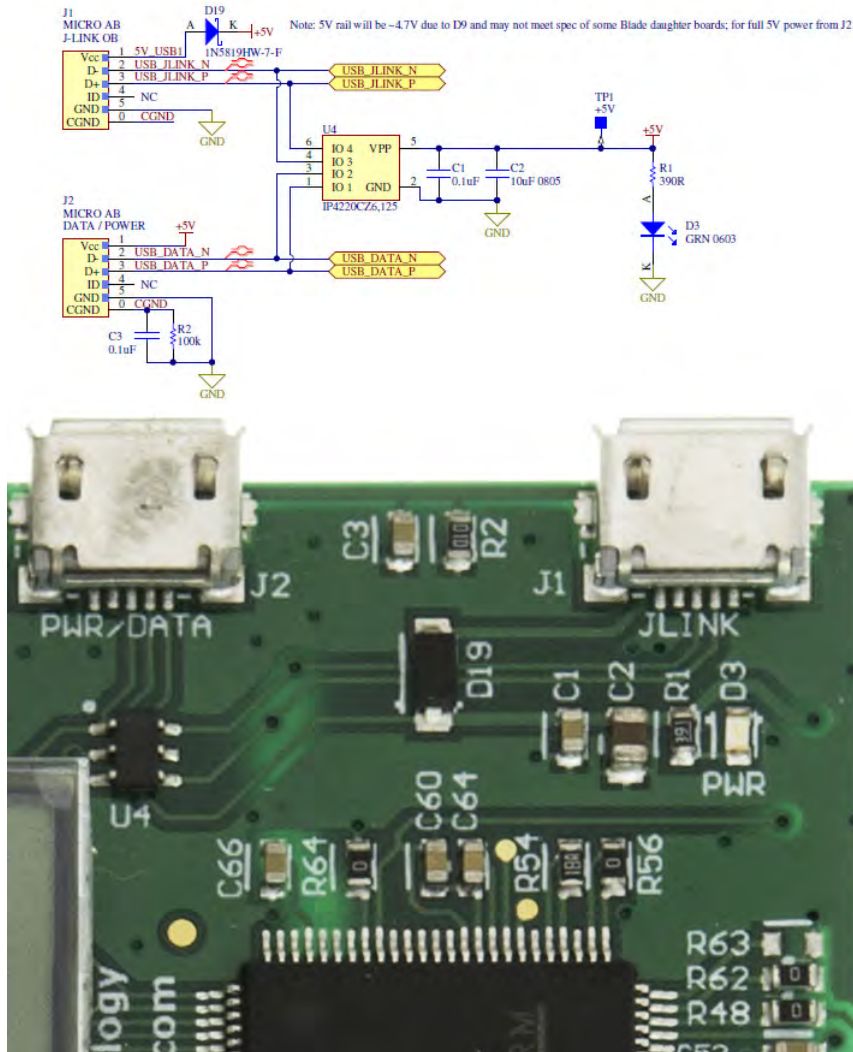


Figure 1-40 5V USB Input

### 1.13 • Main processor, JTAG, clocks, and connections

Next are the processor connections. While this may look like a complicated page, it is mostly just breaking out the 100 pins that all must be allocated to different parts of the development board. What you need to know is how the signals marked with yellow flags such as “LED\_RED” or “LCD\_RST” connect to the physical pins on the processor. LED\_RED connects to PB20 which is pin 65, LCD\_RST connects to PB29 which is pin 71. These represent the logical name of the signal in firmware, the logical peripheral name of the connection for the processor and the physical pin that they both correspond to.

Both LED\_RED and LCD\_RST are designed to be digital output pins, so in the firmware, we can write a logic 1 to a particular location in memory and a logic 1 voltage (Vcc) will appear on the corresponding pin. Writing a zero to that location will clear the voltage to logic zero (0V or GND).



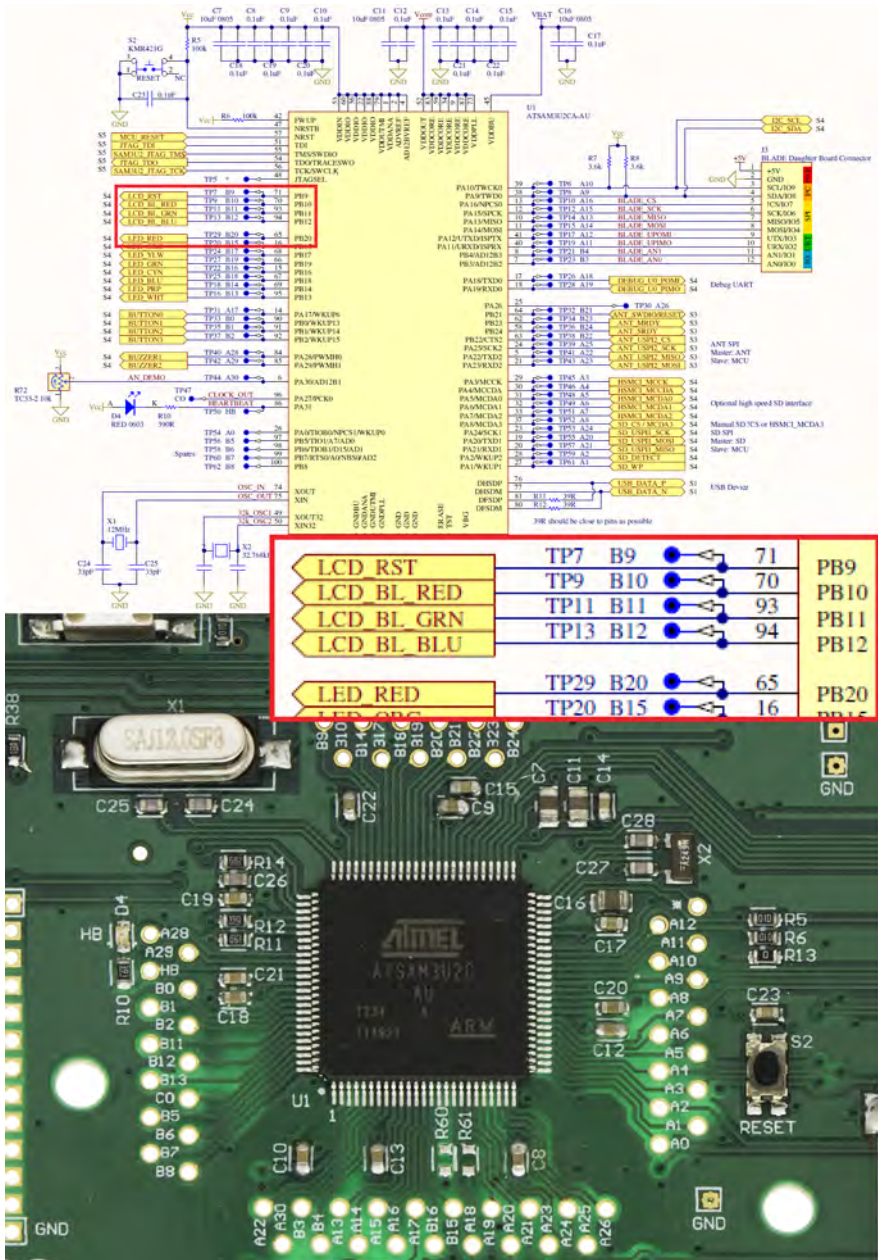


Figure 1-41 LCD and LED connections on development board

This page also has the two crystals that can be used to drive the processor. The main clock is intended to run on the 12MHz crystal. Low power applications can use the 32.768kHz crystal. You should also notice the reset button, trim pot for analog input (on the EiE ASCII development board), and Heartbeat LED.

## 1.14 • ANT 2.4GHz Transceiver

This schematic shows the second processor on the development board, the Nordic nRF51422. This chip has a Cortex-M0 that is also programmable like the SAM3U2. For EIE it is loaded with firmware that will be used for the duration of this text and we will focus more on using the radio functionality of the device.

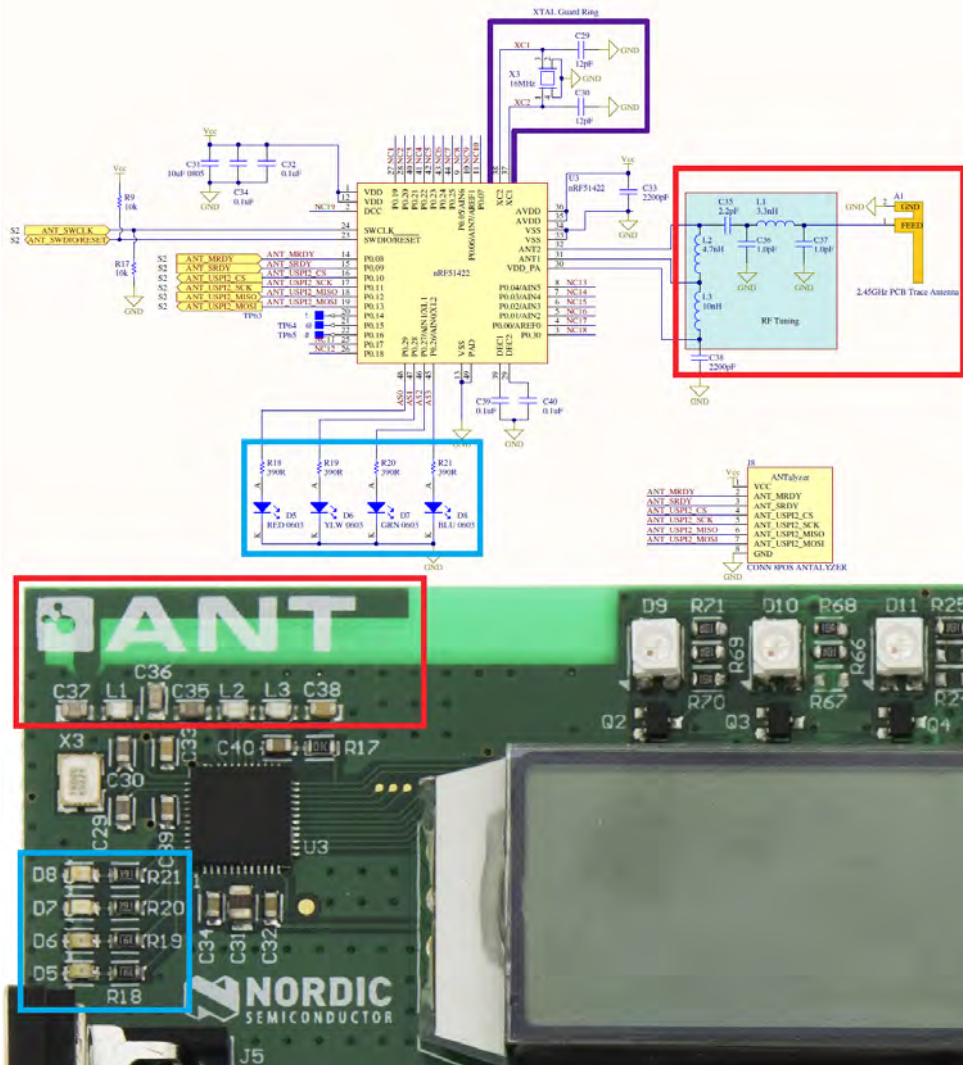


Figure 1-42 Second processor on development board

You can see that the signals are arranged just like on the SAM3U2, so you can equate the signals names to the logical names and physical pins. There are 4 LEDs attached to this processor, two programming lines, and six communication lines back to the SAM3U2. The antenna is built onto the circuit board as a simple trace in the top left corner of the board. The antenna must be spaced apart from other circuits and copper – you can see this



isolation as the lighter color of the circuit board around the antenna.

### 1.15 • User IO: LCD, LEDs, Buttons, Beeper

In Figure 1-43 on page 57, you should recognize at least some of the symbols from discussions at the start of this chapter. Here we are implementing devices as relatively simple circuits. This page makes up almost the entire user interface to the board. The LEDs, buttons, and buzzers are straightforward. The LCD has three communications lines and three lines to control the RGB backlight. J6 is the SD card, and U7 is an interface chip that allows a regular computer serial port to be connected into the development board. This is where the USB-to-serial device connects in. On the latest EiE development boards, the DB9 connector and driver was removed and serial data is directed through the J-Link Onboard processor that has a built-in Virtual COM port. Though this slightly reduces the flexibility of the board, it is a significant cost saving in making the board. Also, users do not have to provide a USB-to-serial converter to connect the development board to a PC.

### 1.16 • J-Link On-Board

The fifth and final circuit page in the schematic is the third processor on the board which is another SAM3U2. See Figure 1-44 on page 58. This processor is pre-loaded with a J-Link programmer firmware license and is what enables programming of the SAM3U2 and nRF51422 by you. Though the processor is physically identical to the other SAM3U2, the schematic symbol is drawn to only show the required pins. The circuit connections are mostly different.

In the bottom left corner, you can see S1 used to select which processor the J-Link is targeting. Be sure that you have the correct processor selected when programming and debugging. Nothing will break if you use the wrong one. The software tools will fail if you try to program the SAM3U2 with nRF51422 code or vice-versa. However, people have gone on long troubleshooting journeys believing their code or hardware was faulty because programming was not working, only to discover they had the switch in the wrong place.

The J-Link SAM3U2 is factory-programmed through a separate connection. If you try to re-program it, you can erase the J-Link code and turn your development board into a brick. It is, of course, possible to re-load the J-Link firmware, but each time a license is used it costs a licensing fee (not to mention shipping it back and forth for the service). There is also a hardware reset pin that can force an erase of the processor. If you probe around this processor, you could erase it, so it is best to stay away from it. We certainly want you to poke around your development board as you learn, but please avoid this area.

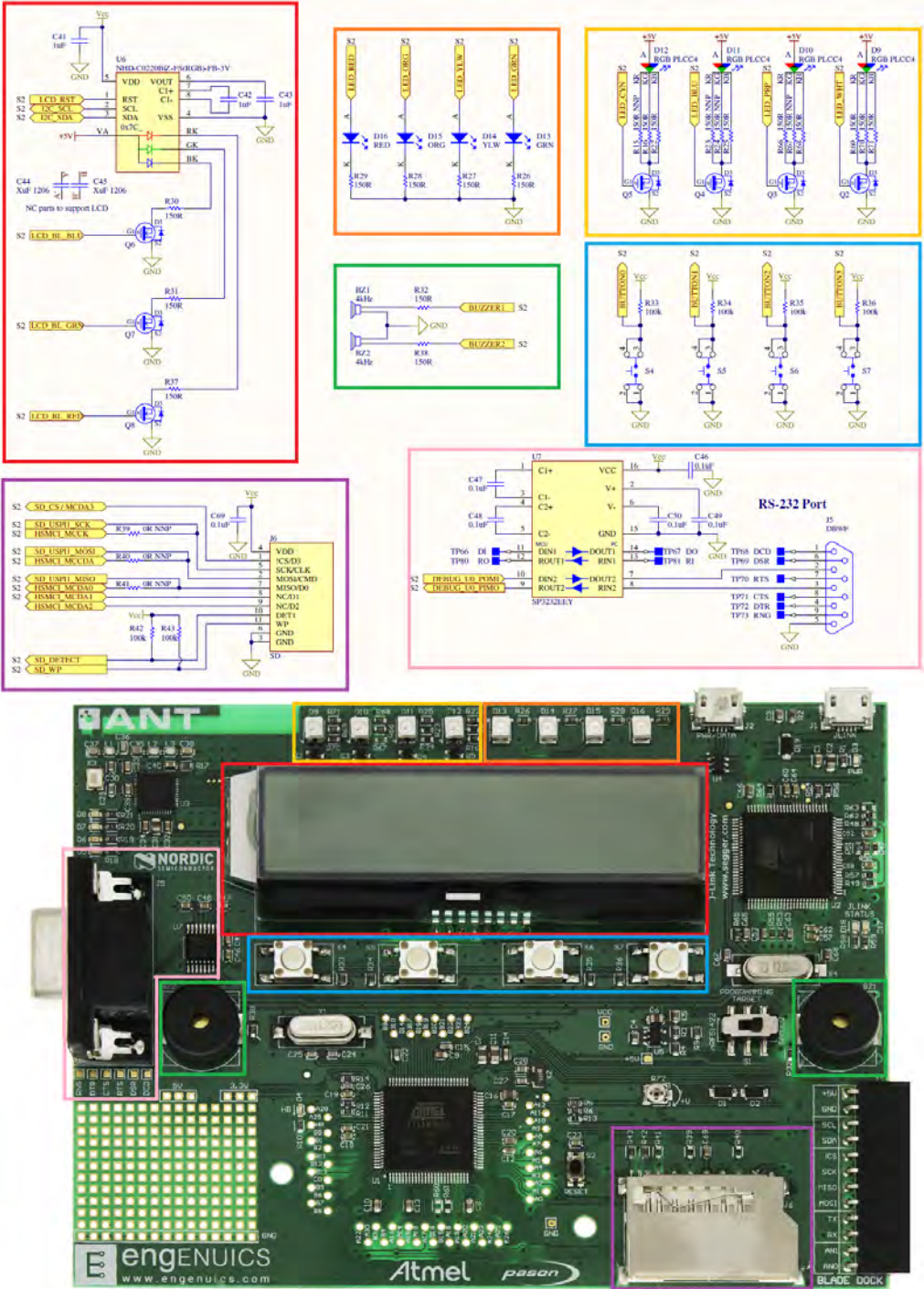


Figure 1-43 Inputs and outputs on the development board

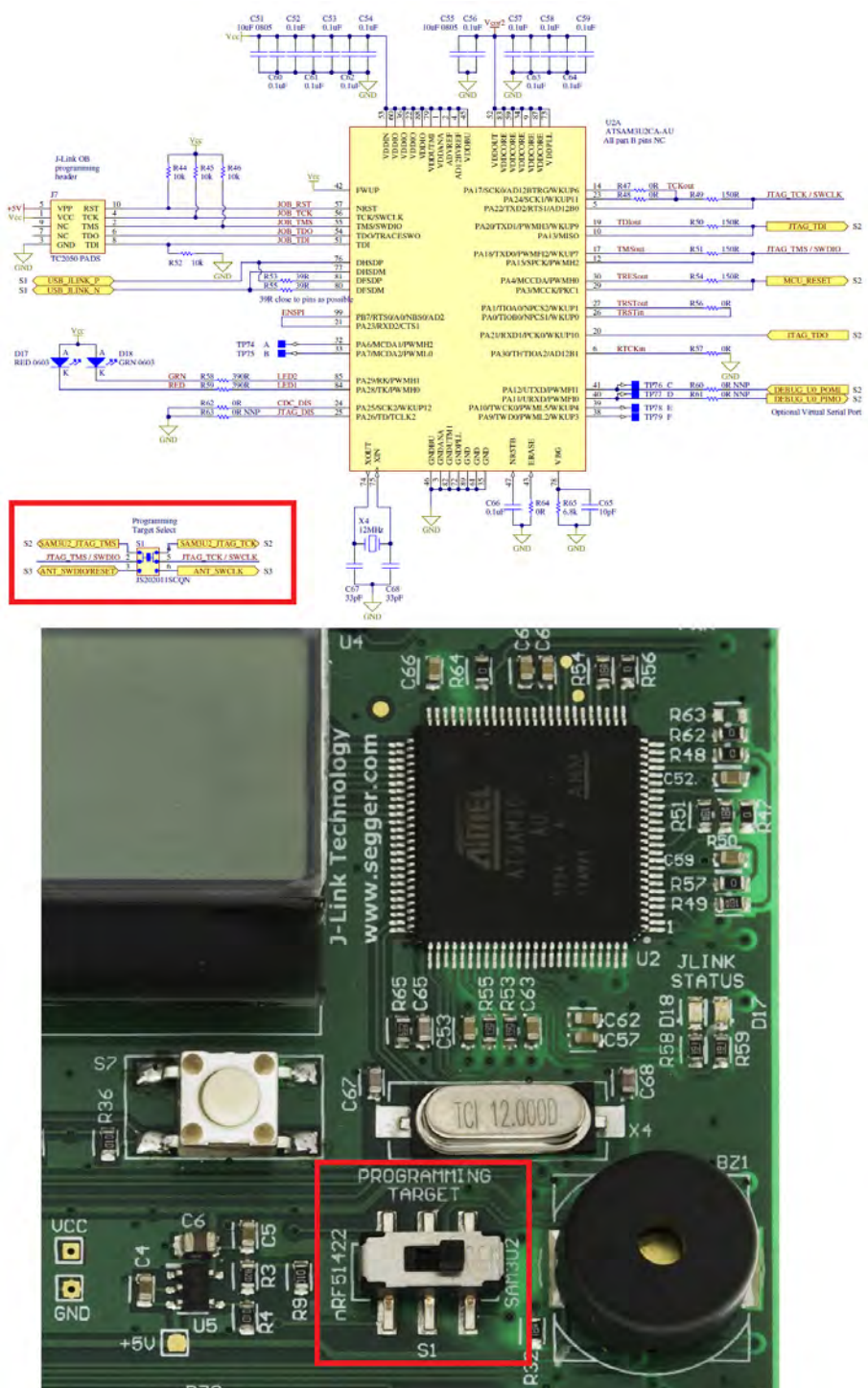


Figure 1-44 J-Link On-Board processor and target selection switch

### 1.17 • Summary

This chapter has given you an overview of the foundational knowledge you need to have if you are going to work with embedded systems hardware. Even if you are not doing hardware design, knowledge of the common circuits and terminology will help you to understand the design you are working with. If you are only doing firmware design, it is still highly recommended to get a copy of the schematics for the design. Review them to make sure the circuits are set up so you will be able to control the logic the way the hardware designer expects.

Hopefully, you see why Ohm's law is so important and where it can get you into trouble with even the simplest of circuits. It is always a good idea, no matter what you are designing and how long you have been designing for, to quickly do a mental check of your circuit to ensure you do not have any paths that are or could be, shorting your power supply to ground. Oh, and try to remember to keep your inputs tied high or low so they do not float, as you may induce wild and crazy signals when you are expecting steady state!



## Chapter 2 • Development Environment & Version Control

In the time you have been tinkering with embedded systems, you have probably noticed the abundance of acronyms floating around. It seems that every industry has a specific jargon associated with it and only the experts that belong to that industry can figure out what each other are saying. It may also seem like a barrier to anyone trying to break into the industry (and it probably is meant that way to some extent). But fear not, just as you have become experts in acronyms that are now so common in your own life that you do not even realize they are acronyms (like TV, LOL, RADAR, etc.), the jargon of embedded systems will soon be second nature.

IDE is a term in the programming world that stands for “Integrated Development Environment.” In our case, an IDE is a software package that provides the main tools to author embedded firmware. It is a container for all the files that will work together in your system. A text editor is provided that allows easy access to the files and does nice things like syntax highlighting and advanced searching. The most important part of the IDE is the compiler which does all the work to take your source code and build it into assembly language.

The compiler is generally what you are paying for when you buy an IDE. There are good compilers and not-so-good compilers, and they are always being improved. A good compiler builds excellent assembly language, where “excellent” means efficient use of instructions based on a solid understanding of the target processor. Compilers also have levels of optimization that can be set to build code that can be optimized for speed, space, or a combination thereof.

The next most important part of the IDE is the debugger which interfaces with your programmer/debugger hardware to run the code on your processor. Once the code is flashed, it can run on its own or under control of the debugger in real time or debug mode. Debug mode lets you run code line-by-line and see all the memory locations in the processor. This is essential for solving problems in your code, especially if you are writing very “low-level” firmware that is the foundation of any embedded device.

### 2.1 • IAR Integrated Development Environment

Our preferred IDE is called “IAR Embedded Workbench for ARM” or IAR EWARM for short. As far as we can tell, IAR does not stand for anything. There is another popular IDE for ARM processors called KEIL which is made by ARM itself and there is a GCC compiler that you can access through various front ends like Eclipse. Each has advantages and disadvantages. There are free full versions of IAR and KEIL that are time-limited for 30 days, or free code size limited versions that you can use as long as you want. The code size limit for Cortex-M3 firmware in IAR and KEIL is 32kB which is lots of space to build a fully functional system. This is all you need for EiE and thus the recommended install.

IAR and KEIL are similar, but we like IAR because versions exist for non-ARM processors that we use like the MSP-430. Having a common platform is helpful since there are always things to learn with new software. It is reasonably simple to port between different IDEs, but you must make sure various high-level settings are the same and any compiler-specific directives or special instructions match the compiler you are using.

### 2.2 • IAR Installation

The screenshots referenced in this book are all from IAR Version 8.10. The IAR version should not really matter, except there are a few things that do matter with the project files. We always keep separate project directories for each version of IAR we use, but they all reference the same source code files.



We haven't had any issues with our source code running in IAR version 7 or version 8, but there are issues if you go back to version 6. Version 7 feels very much like version 6. Version 8 is quite different and takes some getting used to. We have 7.20 and 8.10 archived and available on the EiE website. As this book ages you might want the latest version of IAR, so you should head over to [iar.com](http://iar.com) and click the links to get you to the latest IAR Embedded Workbench. These links always change, so it's left to the reader to poke around to find the download.

The installation process will vary over time, so please reference the EiE website for the latest instructions. There you'll find a step-by-step guide to install and register the software. A couple of suggestions that will likely hold true:

- If you already have an IAR installation on your computer, a window will appear to tell you. We suggest installing a new instance which is what we do since we have different projects that we are working on specific for each version. We like to choose where to install, so create a directory like EWARM\_8\_10\_1 that is specific to the version you are installing.
- There are many different drivers for programmers. You don't need them all and if you need one later it is easy to install the driver. For EiE, you just need the J-Link driver. If you miss installing it, you can install it later.
- You'll probably get a message about USB dongle drivers. This matters if you buy a real license and get a hardware USB dongle with the key.
- Depending on your system, you might see Visual C++ being installed.

When IAR launches, it presents a menu screen where you can access different resources. The User guides can be helpful and keep in mind the example projects that you might want to look at. If you close this, you can open it again under Help > Information Center.

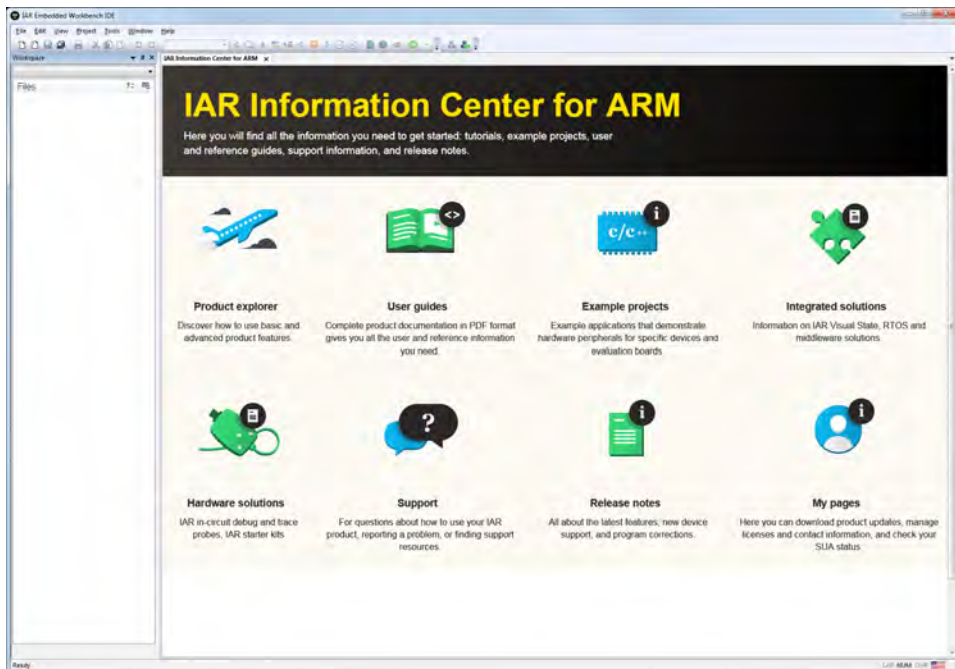


Figure 2-1 IAR information screen

If you are not prompted to get a license right away, find “License Manager” under the Help menu. It’s a very slow program but you only have to use it once. When License Manager is running, choose License > Get Evaluation License. Click the Register button to open the registration page. Make sure you choose “32kB Kickstart” edition. The most important field is your email address because the license will be sent there. We’d be delighted if you put “EiE” as your company and “embeddedinembedd.com” for the website. The processor we use is from Atmel but that does not impact the registration.

Once you submit the form and click the link in the email, you will get to a final webpage where you can copy and paste your license key. Confirm the details and then click next which will take some more time to generate the final keys that are specific to your computer. We have seen this fail a few times on a few machines, where repeating the registration form and getting another key solves the problem. On a couple of machines that still doesn’t work, so there is a way to do an offline registration and copy a key over. On exactly 3 machines in the entire history of EiE, no matter what we tried would not work. At that point, it’s beyond our ability to troubleshoot, so you would be best to contact IAR support for help. Support for free users is lower priority, but we have received responses from them on various issues over the years.

Once it is all finished, you get a confirmation and return to the main License Manager window. You should see a green square which indicates that your license has been successfully activated. You can close the License Manager and return to IAR.

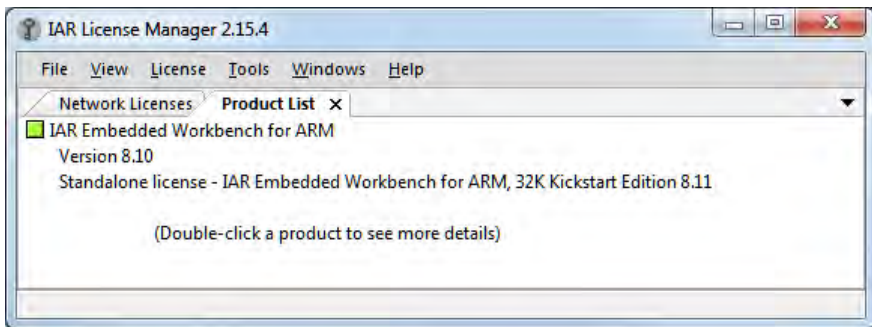


Figure 2-2 IAR license manager with active license

A newer license should allow you to use an older version of IAR. Each key can only be used on one machine. If you are setting up a classroom after you click “submit” you can click the back button in your browser and then immediately click “submit” again and another email with a different key will be sent. We have done this as many as 500 times in a row to generate a lot of licenses. IAR is cool with this the last time we checked.

### 2.3 • Setting Up a New Project

Designs within IAR are captured as “projects” in “workspaces” just as many other embedded tools might describe them. In general, a “project” is the collection of files that are built together to make a program. A “workspace” is the environment in which a project (or multiple projects) lives in the tools. The workspace is your screen setup, configuration settings, and any other personalized view, window, tool, option, etc. that you configure. Only one workspace can be active at one time, though multiple projects can be on the go in the workspace. Experience shows that it is best to have a unique workspace for every project with the same name as the project, rather than trying to use a generic workspace for multiple projects.



Since this is the first time you are launching IAR, you need to create a new project. This will be a very simple project to get you familiar with setting it up. You should always check your project settings to ensure your system is set up correctly, as errors with project settings can cause hours of troubleshooting that you think is code-related but is just incorrect configuration.



Choose Project > Create New Project which will bring up the window shown.

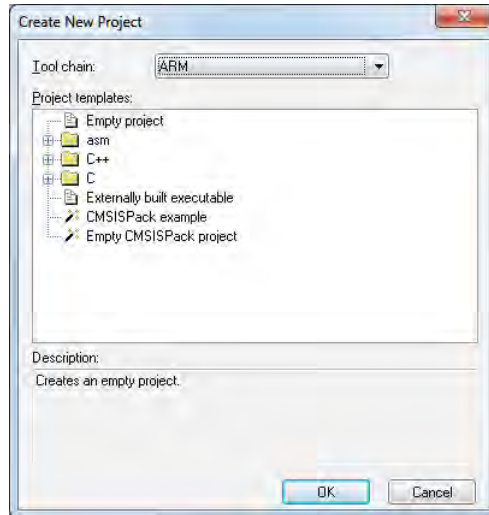


Figure 2-3 Create new project screen

Choose “Empty project” which will open a project without any files in it. If you choose a template, IAR will automatically add some startup files and a shell source file. This can help you get started quickly, but since you will probably have to customize the default files in some way, it can be more trouble than it is worth. You will be prompted to save the project – now is a good time to create an “EiE” folder in an easy place to find on your computer. Being organized is a very helpful skill to develop. We suggest creating the folder `\\EiE\Firmware\\ei_e_ide` for this project. Once that is complete, the blank project screen will be visible.

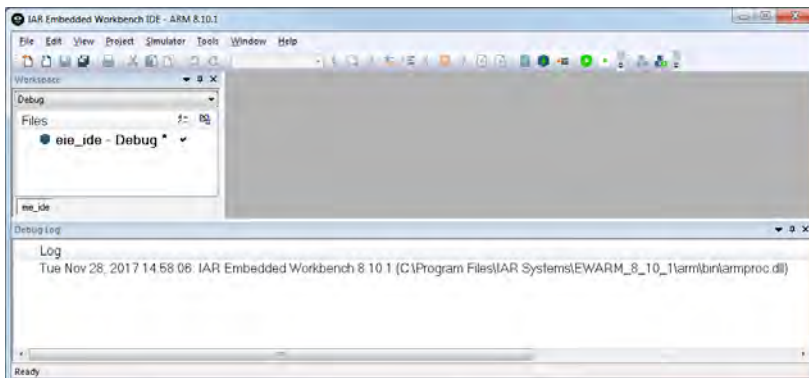


Figure 2-4 Main workspace screen

The main workspace is split into two windows. The left side is the Workspace window and will show you all the files involved in your current workspace. The right side is where your source code files will be edited. Along the bottom is an information window.

Before you do anything else, the project options must be configured correctly. Single-left-click on “eie\_ide – Debug\*” in the Workspace window to highlight it – you always need to have the top-level project name highlighted to edit the project options. The “\*” indicates the project has not yet been saved, but you do not have to save yet. Right-click the highlighted project and select “Options”

In the General Options Category within the Target tab, you should be able to set the “Processor variant” to “Device” and click the small box to choose the processor. Navigate to Atmel > SAM3 > ATSAM3U > Atmel ATSAM3U2C.

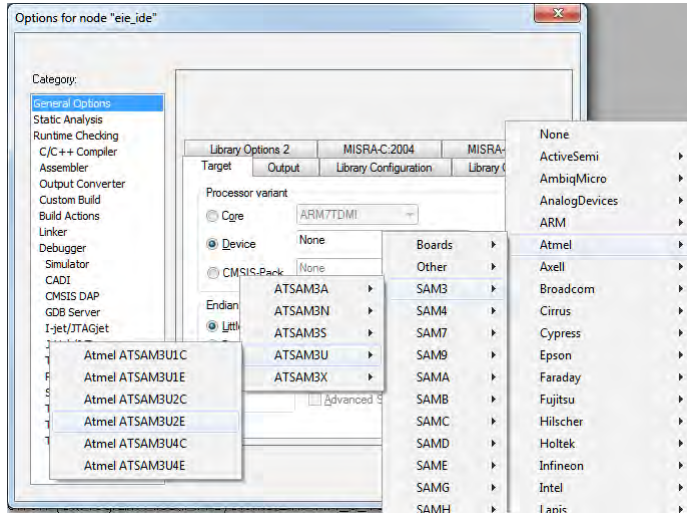


Figure 2-5 Choosing the ATSAM3U2C target processor

Configure the Library Configuration and Library Options 1 tabs as shown and leave everything else as default.

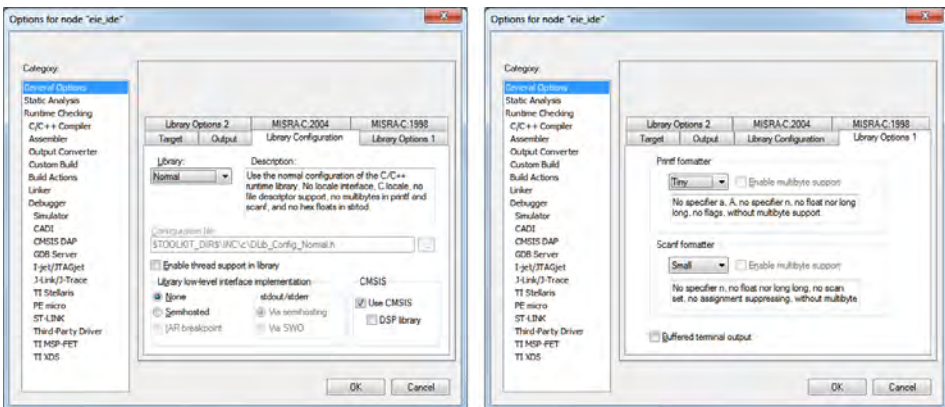


Figure 2-6 Library configuration

In the C/C++ Compiler category set the following values in each of the tabs. If not specified, leave as default.

**Language 1:** Language = C, C dialect = C11, check Require prototypes, Language conformance = Standard with IAR extensions.

**Language 2:** Plain 'char' is Unsigned, Floating-point semantics = Strict conformance.

**Optimizations:** Level = None. It is highly recommended to never develop with compiler optimizations turned on because optimizations do not always work as you expect. If you later turn optimizations on, you have to re-test every aspect of your code to ensure that those optimizations have not broken your code in some way.

**List:** Check "Output list file" and "Assembler mnemonics". If you have a full version of IAR you can check "Output assembler file" to get the assembly code that the compiler generates. The code size limited free version of IAR will not provide this file since it would be easy to compile different parts of the code and assemble them together (there is no limit to how much code can be assembled in the free IAR).

**Preprocessor:** In the "Additional include directories" space, enter \$PROJ\_DIR\$. Though this doesn't do anything right now, this was a good place to introduce the special codes that IAR uses. \$PROJ\_DIR\$ gives the relative path to the current project. The folder which the project files are in is automatically included when compiled, but if you create other folders that need to be included you can specify relative paths like this. Using relative paths allows you to move the project around and not break the folder connections.

**Diagnostics:** Enable remarks. And note that this tab can be used to disable certain remarks.

In the Assembler category, select the List tab and set all the options shown in Figure 2-7. The list file contains a great deal of information when the program is assembled. This includes your symbol table, memory statistics, actual code as substituted from macros or pseudo-instructions and other useful bits of info.

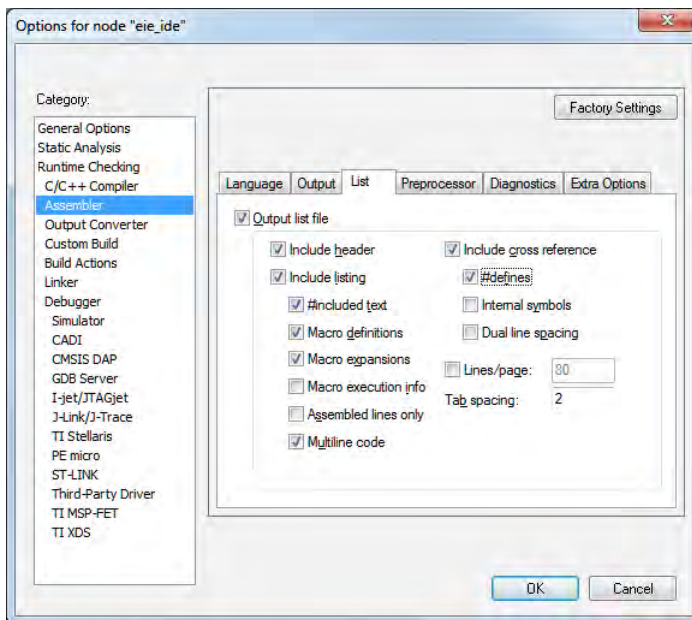
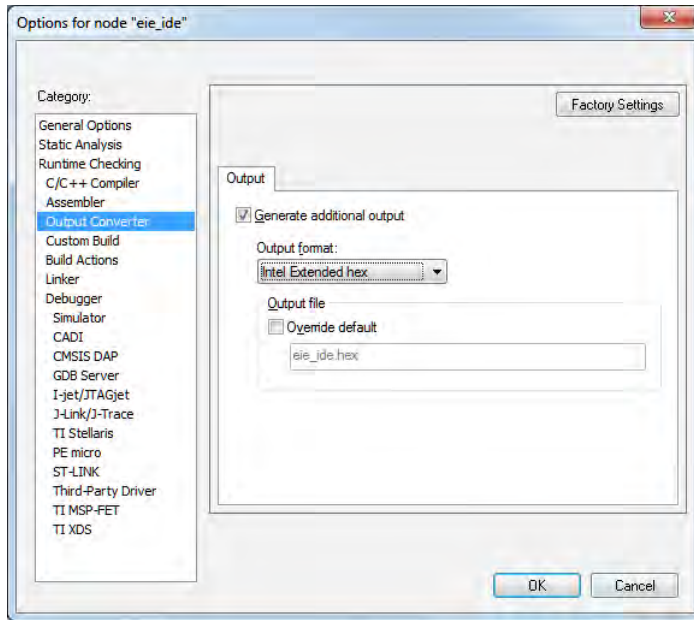


Figure 2-7 Assembler "List" options

**Preprocessor:** Add any paths required here relative to \$PROJ\_DIR\$

**Diagnostics:** Note that Warnings can be enabled and disabled here.

Select the Output Converter category and configure as shown.



**Figure 2-8 Output converter**

Now configure the Linker. Linker files (.icf in the IAR world) are a bit cryptic and thus can be one of the biggest pains in embedded design especially when working with a new processor where the memory addresses, conventions, and syntax are all new. The linker file is a key element in the system when your code is built, though there seems to be little documentation to support them and what is available makes many assumptions about what you know about how building and linking object files work.

Linker files allocate memory and map symbols and your assembly or C files must correctly correspond to the linker information for your code to build, debug, and program properly. If you spend some time to study the linker file, it will start to make sense aside from the syntax used. Eventually, you will have to edit linker files to make some interesting things happen.

You can write your own linker file or grab a default one from IAR. Linker files are found in the folder where you installed IAR in the \ARM\config\linker folder for the device you are using.

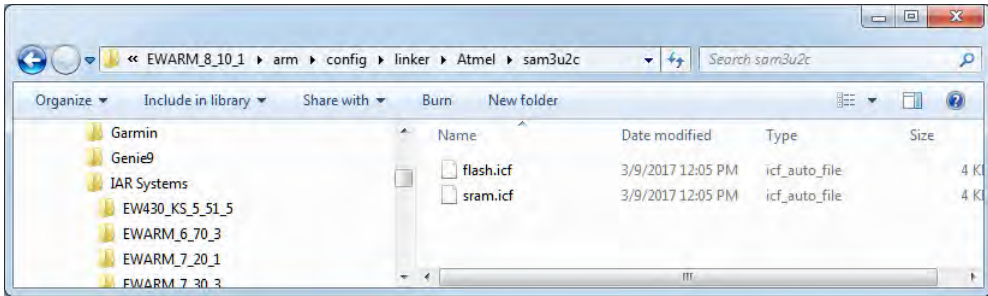



Figure 2-9 Default Linker files for SAM3U2

 Find the linker file and copy it into your project folder. In most cases, you want the “flash” Linker file which will direct the linker to place the program code in non-volatile program flash. Once it’s in your project folder, rename it to sam3u2c\_flash.icf.

Now set up the “Config” tab in the Linker category as shown in Figure 2-10. It is highly likely that you will customize the linker file for your project at some point, which is why we reference a local copy.

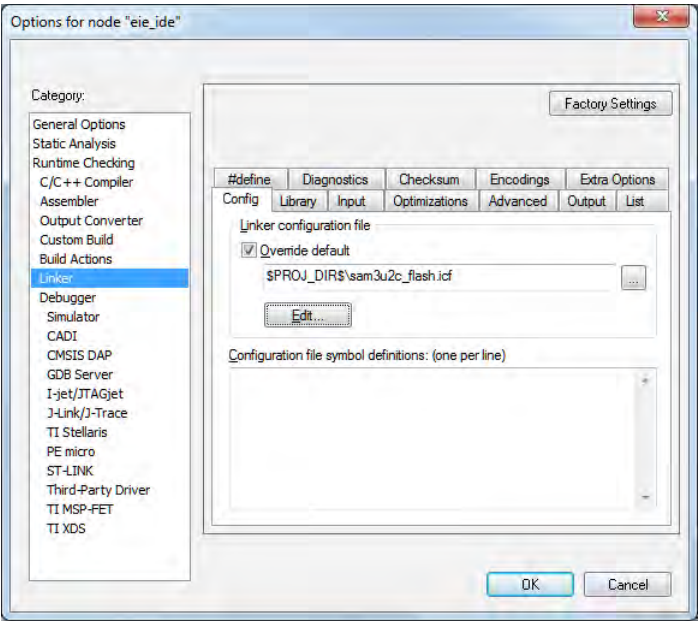


Figure 2-10 Linker configuration

Switch to the List tab and enable everything as shown. The map file is very useful, so make sure this option is set. Looking at the map file produced by the linker can provide important information when your project will not build due to a linker error and shows you how memory is allocated in the final executable when it does build. Also include the log file details as shown – again, the more help the better although we rarely use this log.

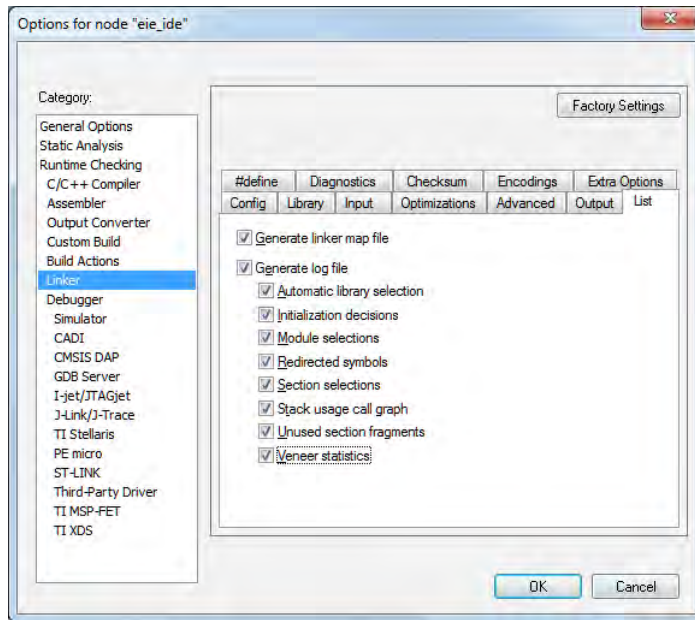
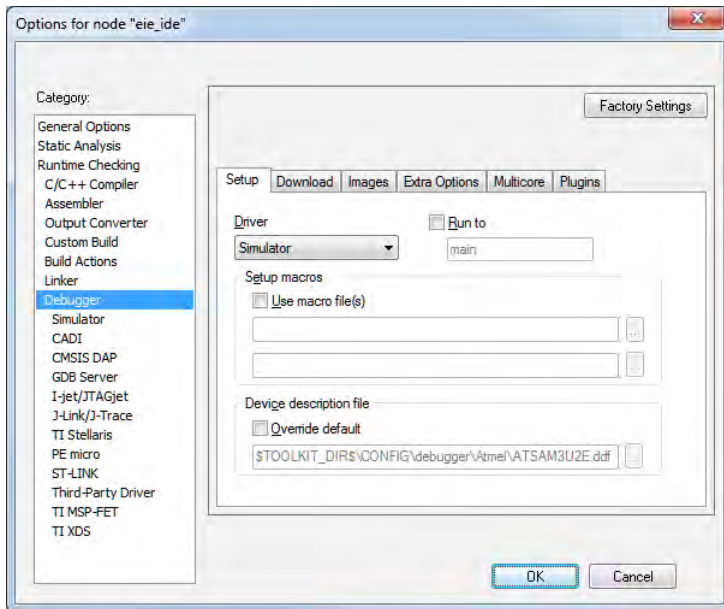


Figure 2-11 Linker list options

You can also change the output filename under the Output tab if you want to, and set up a checksum calculation under the Checksum tab. A checksum adds all the bytes up in the final build and saves the result. During production programming, you can verify that the code has been properly flashed by repeating the calculation and checking that it matches. Checksums aren't perfect because errors can cancel each other out, but it adds a layer of protection. During development, it is not necessary to use this.

Next, configure the Debugger. This is where you can choose between using the built-in processor simulator that IAR has, or connecting to a programmer/debugger like a J-Link. The simulator is quite limited, so in most cases, you will set this to the "J-Link/J-Trace" option. Note the "Run to" checkbox that allows you to automatically start at a certain position in the code when the debugger starts. Any code prior to the function you start is still executed – you just don't see it. In most cases, you want to start at main. For now, leave it unchecked.





**Figure 2-12 Debugger setup**

The only other tab you might use is the “Plugins” tab that enables support for real-time operating systems (RTOS). The EIE system is considered a “bare metal” system with no operating system so this does not apply. If you do use an RTOS, these plugins can supply some valuable debugging information.

That completes the project configuration for now. Click on OK to return to your development environment. Save the project and workspace so you do not lose all that work!



As a final configuration step in IAR, choose Tools > Options and select the “Editor” top-level options. Do the following:

- Set “Tab size” to 2
- Ensure “Tab Key Function” is set to “Indent with spaces”
- Check “Show line numbers”
- Optional: if you do not like the editor feature that allows expanding and hiding sections of code, uncheck the “Show fold margin” box. If you’re not sure, turn it on to see how it works.

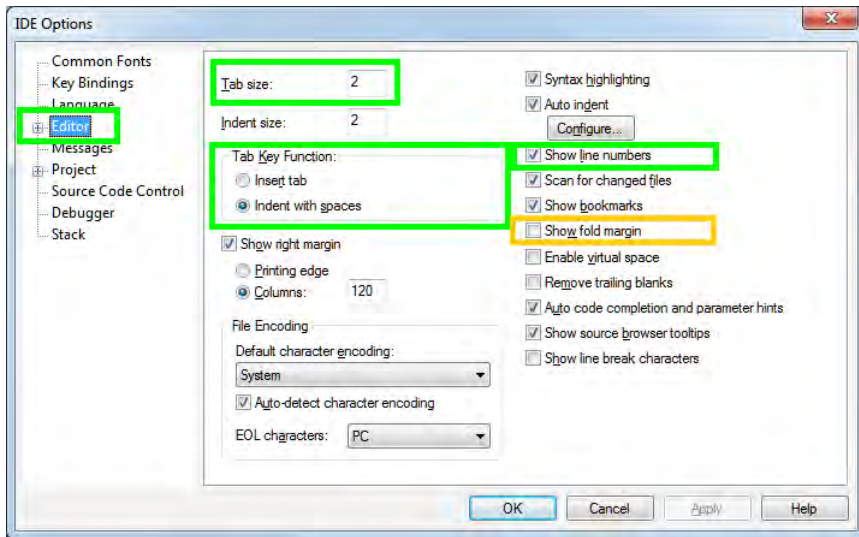


Figure 2-13 IDE options Editor

## 2.4 • Files in a New Project

Projects are built using various source files that will be compiled together to become your firmware. There are some rudimentary things that must be done with all microcontrollers so that they start up correctly. Warning: this is a little bit painful to go through, but if you don't know the process of setting up a project from scratch, you will find yourself stuck before you can even start.

ARM processors have something called a “vector table” that is a list of addresses accessed by hardware to handle certain events. We will learn more about those events later. For now, you need to know that this special list of addresses must appear at exactly the right location at the start of your program in flash. Furthermore, there are certain initializations that must occur when a program starts up including some global variable definitions and perhaps some processor-specific setup that is required before jumping to main.

This boils down to needing what is usually a vendor-provided startup file that might be written in assembler but might also be available as a .c file. The assembler versions are possibly easier to read since addresses and instructions can be written more explicitly, but as most people are more familiar with C that may be the better choice. Finding this file and making sure it is correct can be challenging. The file is IDE-specific since it has some low-level functionality that the IDE's compiler must correctly handle. Compounding this whole problem is that you do this so rarely, it's difficult to really define and remember a good process. Once you have the file, you tend to forget it's even there until you start working with a new processor and find yourself on the hunt again.

A good place to find the file is in an example project or in the supported development boards folders for the IDE. In the case of IAR version 8, you can get the file from a SAM3U development board.



In Windows Explorer, navigate to the IAR folder in Program Files then search for “board\_cstartup\_iar.c” A few options should come up – copy the option from the at91sam3u-ek folder.



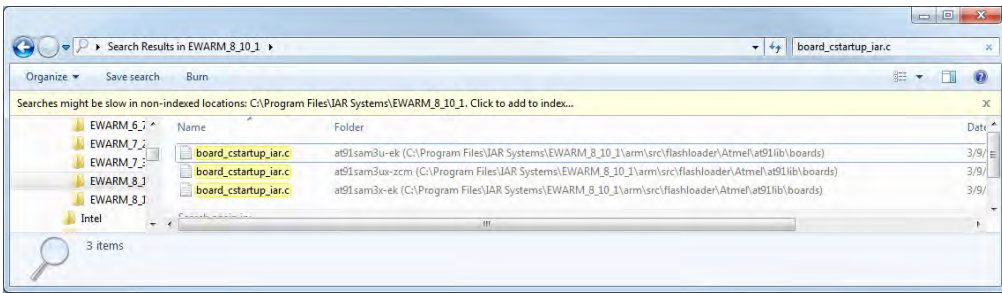


Figure 2-14 File Explorer search for IAR files

Return to IAR and select Project > Add Files... and add `board_cstartup_iar.c`. Once it's in your project, double-click to open it. Scroll around and notice all the "Handler" functions – this is the vector table. Go to the "Headers" section near the top. Your screen should look like this:

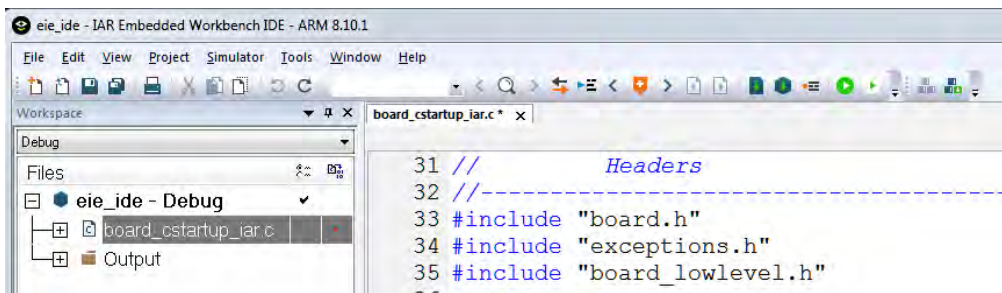


Figure 2-15 Board startup files

The startup file references three header files. Navigate to the `at91lib\boards\at91sam3u-ek` folder where the startup file came from to find these files. If you want to look at them, you can simply drag them into IAR and they will open for you to view. This does not add them to your project.

The files `"board.h"` and `"board_lowlevel.h"` are hardware-specific header files to set up the development board the example code was built for. You are going to learn how to do all of this for the EIE development board, so you do not need them and can close them.

Look at `"exceptions.h"` and you can see a lot of the same "Handler" function names that are in `board_cstartup_iar.c`. It turns out that `exceptions.h` and its source code file `exceptions.c` contains the code that `board_cstartup_iar.c` references. Therefore, copy those two files to your project directory, and add them to the project. Note that you do not have to add header files since their corresponding source files will `#include` them. We like to add them to the project explicitly because it is easier to find them.

We also add the linker `.icf` file to the project so it's easy to find and edit. Do that now, and note that you must select "All Files" to display in the file dialog box to see `.icf` files since adding files defaults to typical source code file types.

Since we don't need the board-specific header files, delete their `#includes` in `board_cstartup_iar.c`. Now your screen should look like this:

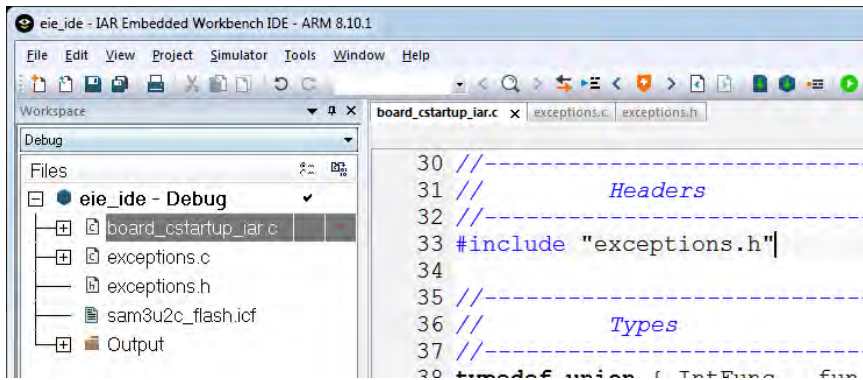


Figure 2-16 Delete #includes in header file

Click the “Make” button which is the one with the light green arrow inside the dark green hexagon (or press F7). This builds all the code and tells you if it works or not.

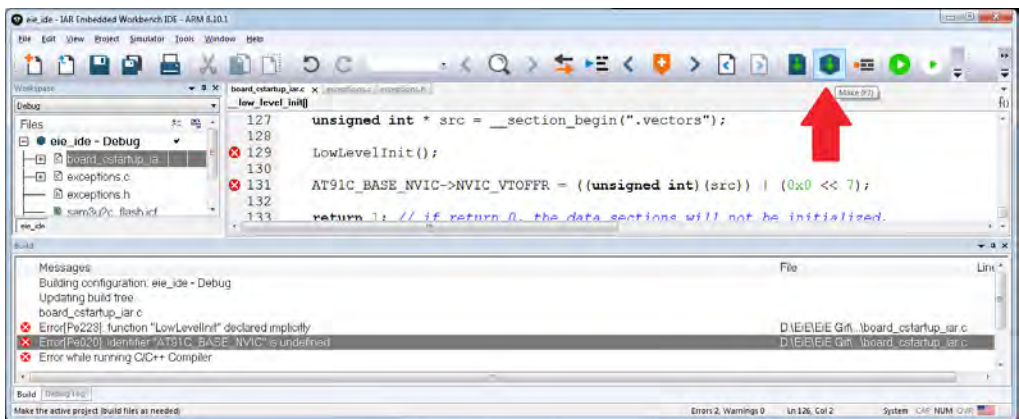


Figure 2-17 Make button

There should be two errors:

1. **Function “LowLevelInit” declared implicitly** – this usually means that you are trying to call a function that does not have a function prototype and/or definition.
2. **Identifier “AT91C\_BASE\_NVIC” is undefined** – this usually means that a symbol (which is just a word or name) you have typed (in this case AT91C\_BASE\_NVIC) cannot be found.

The first error is the result of stealing this code from another program that provided another function to set up more things on the processor. Since we are not using that development board, delete the LowLevelInit() function call.

The symbol that is trying to be referenced that results in the second error can be found in another header file that we need to include. This is a processor-specific definition file. Vendors provide these files which have every single bit name and register name that the processor needs. These names *should* match the names that are described in the processor’s user guide, so that you can read the guide and copy the names directly.

This file will be hiding somewhere and the example project we have already referenced is a good place to find it. The file is called AT91SAM3U4.h. If you were using a different processor, you would search for a file name that matches or resembles the part number you're using. Different vendors have different naming conventions. The file is massive – it is not something you would ever want to write yourself.

Once you find AT91SAM3U4.h, copy it into your project folder and add it into the project Workspace window. Then use `#include` to reference the file in the “Headers” section at the top of `board_startup_iar.c`.

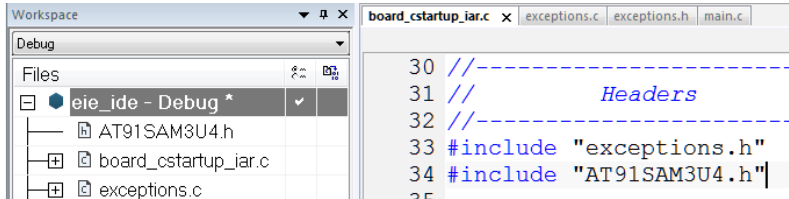


Figure 2-18 Headers section

Build the code and the two errors should be gone, but a new error “no definition for “main” from cmain.o” should appear. If you look through the source code we have, there is no call to main, so where is this coming from? The short answer to a long story is the reference to `__iar_program_start` at the top of the vector table.

```

67 {
68     { .__ptr = __sfe( "CSTACK" ) },
69     __iar_program_start,
70 }

```

Figure 2-19 Reference `__iar_program_start`

The first entry in the vector table is called the “reset” vector, which is where the code immediately goes when starting up from a power cycle or reset. In this case, it’s a function called `__iar_program_start`. This function is part of the C runtime library built-in to IAR that allows the compiler to add the super low-level code like variable initialization. The free version of IAR does not include the runtime library source code, but once you get the project to build you can step through the assembly language to see what the function does. As you’ll see, the function ends with a call to main.

To solve this last error, choose File > New File which will open up a blank text file. Save this as `main.c`. Add it into the project, then put in the classic function definition for main in an embedded system:

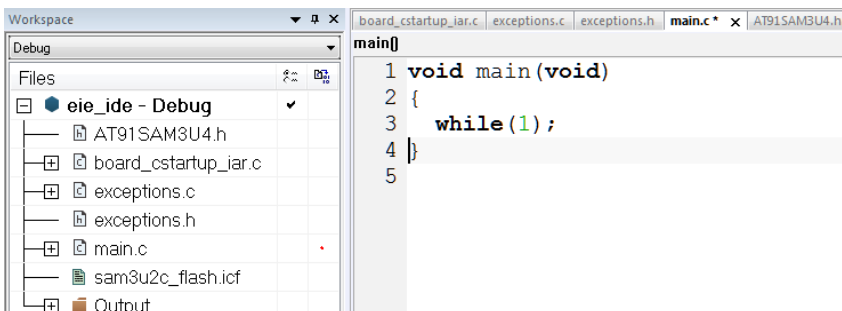


Figure 2-20 Adding `main.c`

Build the code again and celebrate the success of your embedded C project! Sure, it doesn't really do anything, but the mysteries you just learned about are critical to getting a new embedded system up and running. Complicated? Absolutely. Finding out all this information is extremely difficult as it is generally poorly documented, and many programmers have no idea about low-level startup functions. But now you do! The process is very similar for all ARM processors, and other cores as well.

## 2.5 • IAR Simulator and Debugger



To run the program in the simulator, press Ctrl-D to start the debugger. You can also click the green “Play” button. Your screen should look something like this:

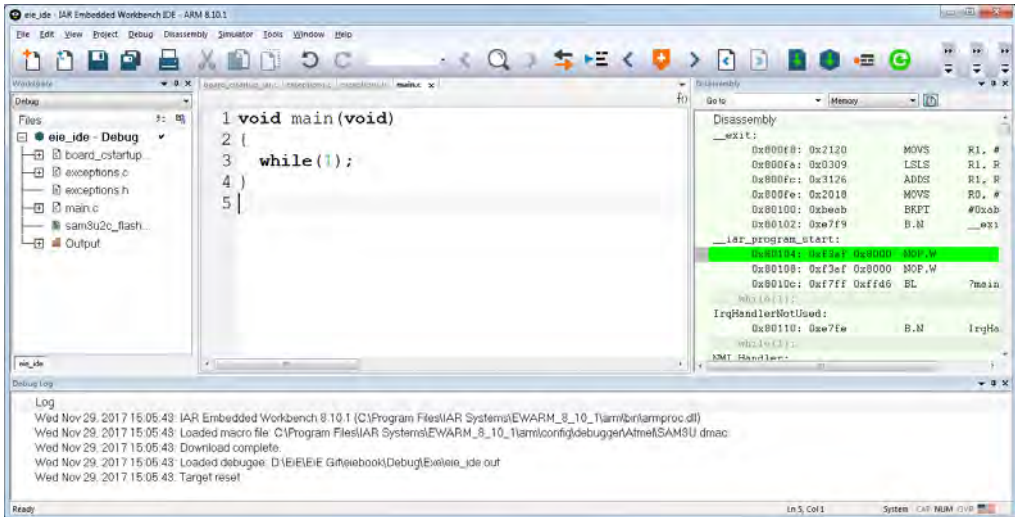


Figure 2-21 Simulator screen

If you do not own a 24" monitor, now is the time to buy one because it is very helpful to have a ton of window space available during development and debugging. In fact, a pair of 24" monitors is the ideal workstation for embedded development.

What you should be able to see is a green highlight bar in a window called “Disassembly.” See Figure 2-21. The green bar tells you that the debugger is active but halted. The green bar also represents the current address that the program counter is at and thus tells you what line of code is about to be executed. This is the first instruction at the reset vector address.

The “Disassembly” window is extremely important because it shows every instruction that was created from the code in your project (or from external files that end up getting linked in once you start including other files) along with each of those instructions’ addresses. It also shows the C source code from where the instruction originated from in light gray, where applicable.



Click inside the Disassembly window anywhere to activate it, then press “F11” a few times to get to the “return 1” line of code. If you are in the Disassembly window, F11 causes one instruction to be executed at a time. If you are in a C source file window, F11 causes one line of C code to be executed (which can be many lines of assembly).

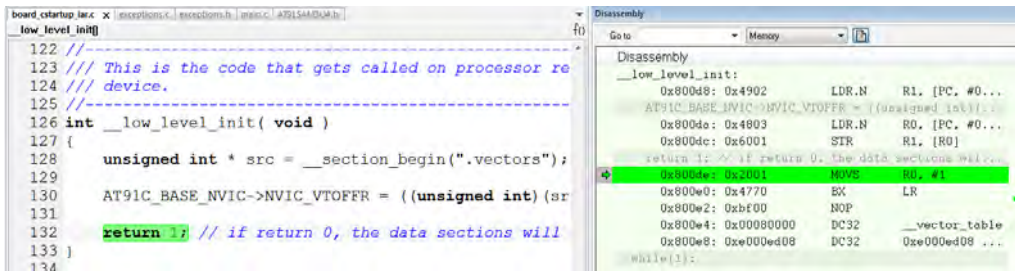


Figure 2-22 Disassembly window

In this example, the MOVs instruction is taking the number 1 and putting it into a memory location called R0. The BX is called a branch instruction and it is going to move the program counter back to an address that is stored in another memory location called LR. These two instructions are the result of the “return 1” line of C code.

Keep pressing F11 until you get to main(). If you continue to press F11, nothing seems to happen because the while(1) loop is just one instruction that branches to itself.

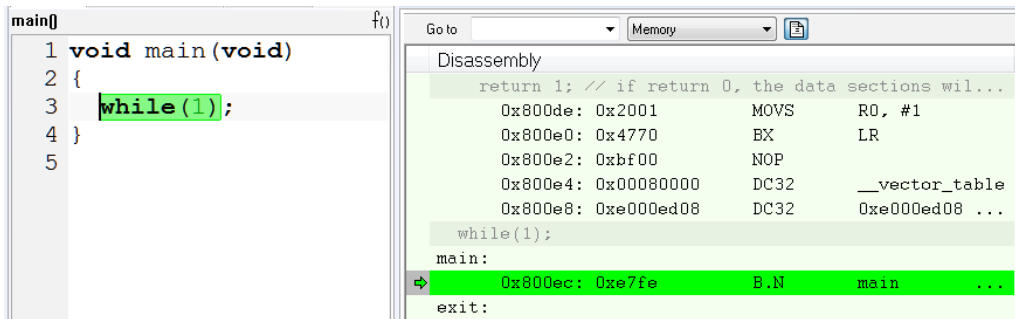


Figure 2-23 While loop

Before we do anything else, we need more information. Open the following windows:

1. View > Breakpoints
2. View > Watch > Watch 1 (repeat for 2, 3, and 4)
3. View > Locals
4. View > Memory > Memory 1
5. View > Stack > Stack 1
6. View > Symbolic Memory

The windows will open individually, and your workspace will appear quite crowded as shown.



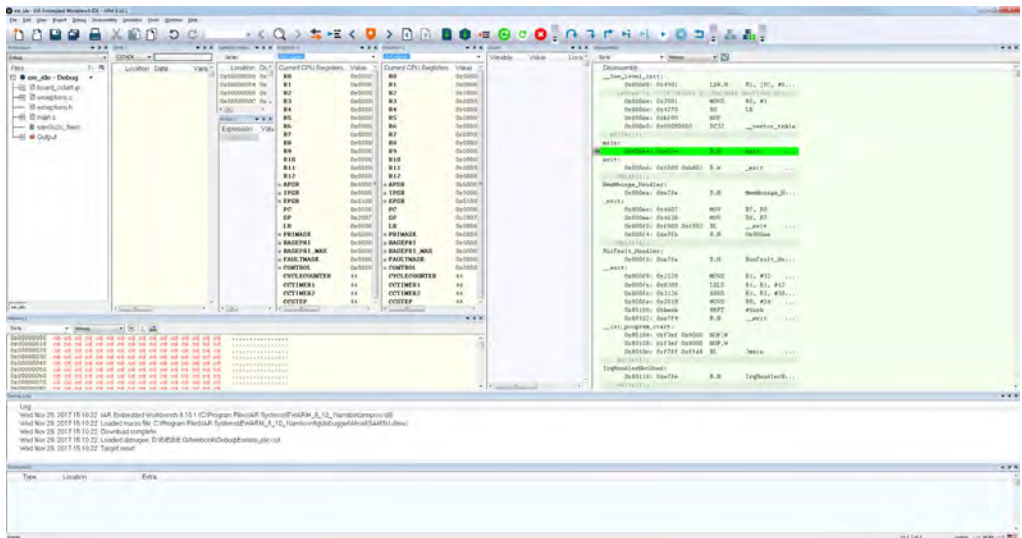


Figure 2-24 Workspace

You can group windows together by clicking and dragging them into each other and of course, you can resize and move them around as well. IAR 7 and IAR 8 work differently and can sometimes be frustrating. Experiment to see how resizing windows behaves. Though your own ideal window display and organization is something you will have to figure out, start with the following suggested setup:

- Group Debug Log, Breakpoints, and Build
- Group the Disassembly Window, Memory, and Symbolic Memory and put them at the bottom to the right of the Debug Log, Build and Breakpoints
- Group Registers 1, Watch 1, Watch 2, and Locals
- Group Registers 2, Watch 3, Watch 4, and Stack 1

In the end, your environment should resemble what is shown in Figure 2-25 on page 78. Save the Workspace at this point (with the debugger still running) to make sure the new configuration is preserved. We find this to be a very effective organization for all the debugging windows that you end up working with. Of course, different debugging scenarios may require different configurations so the most important takeaway is that you know what resources you have and how to organize them.

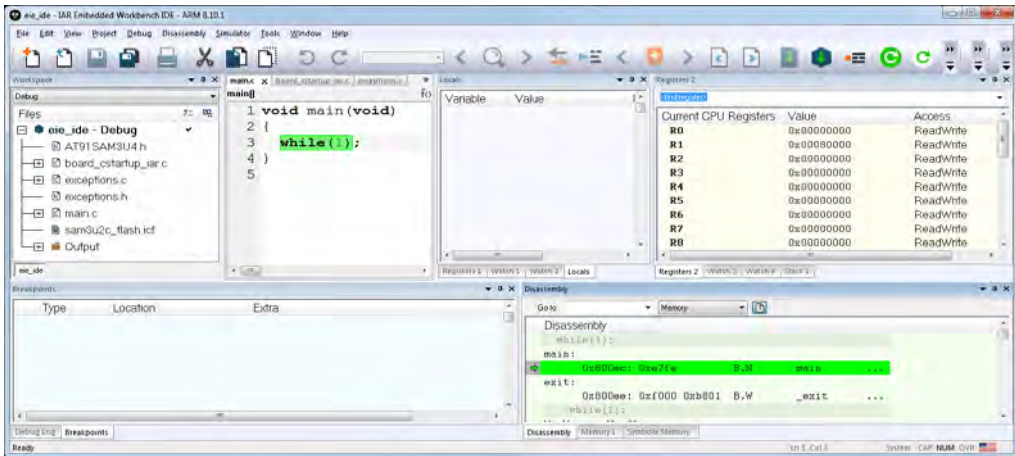


Figure 2-25 Debug environment

To navigate the code, the debugger provides a host of options which are available from the Debug menu, as shortcut keys, or on the Toolbar. Figure 2-26 shows there is a substantial difference in appearance between IAR 7 (top) and IAR 8 (bottom), but the functionality is identical. You can hover over the buttons to see their functions. The choices available depend on whether the code is halted (left) or if the code is running (right).



Figure 2-26 Debugger toolbars

**Reset:** Returns the program counter to address 0x0000 and initializes any register/memory location that has a defined reset state. All other memory locations will be left untouched, just like the real world. There is no default shortcut key.

**Step Over:** Executes a line of code and any code called by that line. If the line of code is a function call, the debugger will execute all code in the function including the function return. The shortcut key is F10.

**Step Into:** Executes only a single line of code. If the line of code is a function call, the program counter will advance to the first instruction in the function and wait. This is also called a Single-Step. The shortcut key is F11.

**Step Out:** Will run code to the end of the current function including the return, then will halt on the next instruction. The shortcut key is Shift-F11.

**Next Statement:** Runs to the next complete statement in the program. In assembly mode, this is simply the next instruction, but in C mode will step between complete statements. There is no default shortcut key.

**Run to Cursor:** Code is executed until the line indicated by the cursor. There is no default shortcut key, but you can right-click on a line of code after you place the cursor there and get "Run-to-Cursor" on the pop-up menu.

**Go:** Runs the program freely. Unless a breakpoint is hit, the code will continue to execute. While the code is running, nothing will appear to happen in the debug windows until the

code is halted at which point all the windows will update with the current information. This mode is mostly useful when debugging through a JTAG device since the target runs in real time. The shortcut key is F5.

**Break:** When the code is running, the Break command is available which stops the running program wherever the program counter currently is and forces an update of all the debug information windows. Also called “Halt.” There is no default shortcut key.

**Stop:** Stops the debugger and returns to the IDE. In most cases, the code will continue running on the processor.



Try clicking Stop to return to editor mode. While you’re here, use the project options to turn on the “Run to main” option in the debugger. Restart the debugger and the code should be on the while(1) line of C code.

Activate the Registers 1 debug window and note the CYCLECOUNTER value which should be 23. These means the processor has already executed 23 clock ticks to run the initial startup code. The cycle counter is a great tool to determine how many instruction cycles a given piece of code takes to execute which may be very important in timing-critical functions. Unfortunately, this feature only works in simulator mode and is not shown when debugging on a real target. If you need to know how many cycles a snippet of code takes in a big project, you might be able to rip it out and run it in a simulator on its own. Calculating instruction cycles manually is also possible, but difficult on an ARM processor.

Current CPU Registers	Value	Access
EPSR	0x01000000	ReadWrite
PC	0x000800E4	ReadWrite
SP	0x2007D000	ReadWrite
LR	0x000800D3	ReadWrite
PRIMASK	0x00000000	ReadWrite
BASEPRI	0x00000000	ReadWrite
BASEPRI_MAX	0x00000000	ReadWrite
FAULTMASK	0x00000000	ReadWrite
CONTROL	0x00000000	ReadWrite
CYCLECOUNTER	23	ReadOnly
CCTIMER1	23	ReadWrite
CCTIMER2	23	ReadWrite
CCSTEP	23	ReadOnly

Figure 2-27 Current CPU registers

Now press F11 once and look again at CYCLECOUNTER. The values in red indicate that they have changed since last time the code was halted (single stepping is just running one line of code and then halting again). The CCSTEP value is the number of cycles that just occurred. CYCLECOUNTER keeps the running total.

CYCLECOUNTER	26
CCTIMER1	26
CCTIMER2	26
CCSTEP	3

Figure 2-28 CYCLECOUNTER

Press F5 to run full speed for a few seconds, then press the Break button to halt. You can see that processor cycles accumulate very quickly!



<b>CYCLECOUNTER</b>	49668944
<b>CCTIMER1</b>	49668944
<b>CCTIMER2</b>	49668944
<b>CCSTEP</b>	49668918

Figure 2-29 Processor cycles accumulating

Halt the code and update the code in the main loop:

```
void main(void)
{
    unsigned long x = 0;

    while(1)
    {
        x++;
    }
}
```

Build the code and start the debugger again. Activate the “Locals” window and you should see the Variable ‘x’ there. Activate the Registers 2 window. Right-click in this window and select “View Group” and make sure it’s on “Current CPU Registers.” Adjust so you can see “PC” in the Current CPU Registers window. Take note of all the other register groups that are available for you to look at – we will use several of these in later chapters.

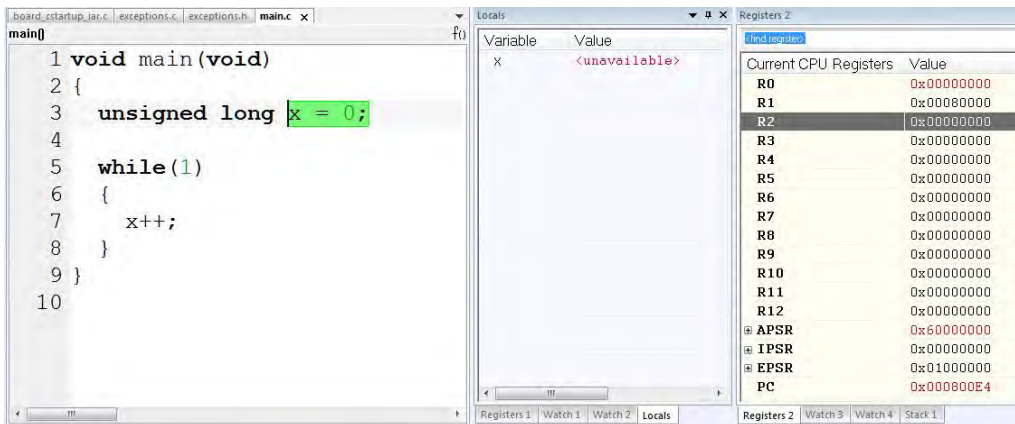


Figure 2-30 Debugger local windows

Use F11 to single step and watch the variable change as you go through the loop a few times. Also note what’s happening in the Registers and Disassembly windows. Test the difference of single stepping in the main.c window compared to the Disassembly window.

It does not matter if you have no idea yet what a CPU Register is, or why there are 13 of them from R0 to R12 plus some other strange things. All you need to recognize is the following:

1. Everything that changes with each step turns red in the active debug windows.
2. The memory location R0 is incrementing every loop and matches the variable x
3. The program counter (PC) is either 0x800e6 or 0x800e8 (it will change if you are stepping in the Disassembly window)
4. CYCLECOUNTER increments by 4 each loop



Press the Reset icon. What happened? The processor has been reset. Now you can repeat everything in case you missed something. Pressing Reset is quite different than stopping the debugger and restarting it. Be aware of this when working with your code.

Press Ctrl-Shift-D. Now you are back to the editing environment. Close this workspace now.

Everything that you have just done here is the basics of debugging. Your environment is set up in a way that will allow you to get a ton of useful information about what is happening on the processor as it executes code. You are starting to get a feel for the debugger and you have witnessed bits moving around because of instructions being executed on the processor. You could load this program onto a real micro using a JTAG interface and see the exact same results. Would anything happen on the development board? Nope. All you are doing is loading a couple of memory locations with some arbitrary values. Though these are core registers on the processor, nothing happens externally when you write numbers to them. All your processor would be doing is burning power as it looped the program counter with the branch instruction.

We cannot emphasize enough how important it is to use the debugger effectively. Knowing how to write good code is not even half the battle. The best programming experts are also experts at debugging and testing. Growing all three skills concurrently will be the fastest way to finding success in writing firmware.

## 2.6 • Other Development Tools

While you will spend most of your time writing and debugging firmware in the IDE, there are some important other programs that will be used throughout this book and during regular development. Archived versions of all this software are available through the EiE website, or you can search for the latest versions directly from the vendors online. These resources are all freely available.

### 2.6.1 • Tera Term

Tera Term is a terminal program for Windows. Terminal programs have been around forever and speak “RS-232” serial out the computer’s serial port. It’s rare to find a computer with an actual serial port these days, so there is a myriad of USB to Serial adapters available. Some are good and some not so good. At the time of writing, the ones that use “FTDI” chipsets seem to be the best. We even designed our own based on the FTDI chipset and added features that most commercial adapters lack because it is such an important development tool.

The RS-232 protocol is quite straightforward and does not require a lot of code on the embedded side to make it work. You’ll learn all about this later in the book. It is almost guaranteed that an embedded device will have a serial connection available for debugging or configuration purposes. If you design embedded hardware and don’t put in a serial port connection, we will call you crazy. To interface a PC to that connection, all that’s required is a level translation from the voltage on the embedded system to the +/-5V specification for RS-232. If you see a big DB9 connector on a development board or product, this likely means the level translator is built-in to the development board and you just need a USB to Serial adapter. If the embedded device only offers Tx (transmit), Rx (receive), and GND (Ground), this probably means you need to supply a level translator in addition to a serial adapter.

A terminal program on the PC interprets the data. Connections are made through “COM” ports and will appear in Windows Device Manager. The latest EiE development boards have hardware connections and firmware in the J-Link Onboards that offers a “Virtual COM port” over the same physical USB connection as the J-Link function. This eliminates

the need for a physical DB9 connector, level-conversion IC and external USB to RS-232 converter.

Whichever way you physically connect a PC to an embedded system for serial communication, a terminal program allows you to send and receive data over this connection. There are many choices, but Tera Term offers a nice blend of features and is easy to use. We use RealTerm when we need some more options.

### 2.6.2 • GitHub Desktop

Version control software is essential for efficient, intelligent development whether you are working by yourself or in a team. Version control has evolved through programs called CVS, SVN, and now Git which we will introduce in this chapter. These programs work using a local client application which connects to a server or cloud-based central hub. Companies may have internal servers, but there are also many paid cloud services and some free ones for individuals.

GitHub is a firmware community that is built on users contributing code. At the time of writing, there are reportedly close to 20 million registered GitHub users. You can be part of this community for free and have your code hosted in the GitHub cloud. This is what we do for EiE so our firmware base can be easily accessed globally.

GitHub offers a free graphical user interface (GUI) Git client called GitHub desktop. When you install GitHub desktop, you also get a shell-based program for command line access. The command line offers substantially more features but of course has a higher learning curve. The desktop application simplifies the process of interacting with the online repository so that is what we use primarily in EiE. Both the command line and the GUI will have the same net result of reading and writing to the GitHub cloud.

You are encouraged to create a GitHub account to ensure you can easily access the latest EiE firmware. By developing in GitHub, you also build a record of the work you have done which can be very helpful when it comes to getting a job or just trying to figure out what you've done over the last few years. If you keep your work in the cloud, it is also safe from getting lost. Some companies will ask to see your public GitHub account to check out how you write code and how much experience you really have. Start now if you haven't started already!

### 2.6.3 • ANTware II

ANTware II is a software application that enables testing the ANT radio on the EiE development board. It uses an ANT hardware device such as the ANT USB-m connected to a PC. This is essential if you are going to use the ANT functionality of the EiE system or do any ANT development.

Downloading ANTware is free, although a user account is required to access it. The website is [www.thisisant.com](http://www.thisisant.com). Sign-up is free. Downloading the archived software from the EiE website is faster. If you choose to log in and download it, find the Software Tools menu and load the current version of ANTware II and the USB drivers.

### 2.6.4 • nRFgo Studio

The nRF51422 processor on the EiE development boards is a special “system on chip” (SOC). The processor memory is divided into two sections. One section is for your own firmware that you can develop and debug normally through IAR. The other section is for the proprietary “soft device” firmware that must be loaded. Different soft devices exist for ANT, BLE, and ANT+BLE concurrent mode. The soft devices are updated often.

To write a new soft device, you need a freeware programmer provided by Nordic Semiconductor called nRFgo Studio. The EiE development boards are pre-programmed with the correct soft device and user firmware so if you do not plan to change it, then you don't need to install nRFgo Studio. If you do want to change it, please install the latest version.

### 2.6.5 • Windows Settings

While you might think this is a trivial step, good organization and having all your tools readily available and configured correctly will save you a ton of time.

Firstly, make sure “Show file extensions” is turned on in Windows. If not, IAR will not display file extensions and you will constantly be confused. This setting is found under “Folder and Search Options” in Windows Explorer.

Secondly, if you're going to be an embedded developer, you need to know about Windows Device Manager. You will use Device manager often, so create a shortcut to it. The easiest way to find it is by typing “device manager” in the search area. On Windows 7 you can type it under “Run”. When you see it, right-click and create a shortcut.

Creating the shortcut in Windows 10 always seems to be a problem, so just follow these steps:

1. Open-File Explorer
2. Go to \\Windows\\System32
3. Find devmgmt.msc
4. Right-click and drag out to your desktop
5. Select “Create shortcut”

Lastly, to facilitate easy access to all the installed EiE software, a program group called “EiE” should be created in the Windows Start Menu with links to each of the above applications. A link to the EiE home page ([www.embeddedinembedded.com](http://www.embeddedinembedded.com)) can also be added using the default browser. You can also create a shortcut to this folder on your desktop.

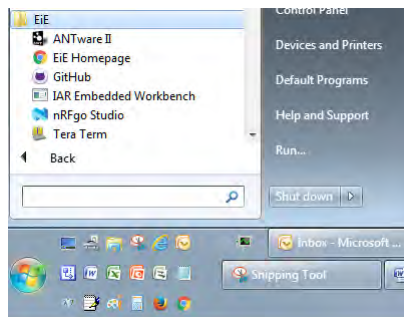


Figure 2-31 Windows start menu

The development board schematics and the processor User Guide are all available on the EiE webpage. You'll need these, so find them and download them so you're ready to go!

### 2.7 • Version control with Git

Version control is fundamentally important to developers. Whether you are an individual working on your own design, or part of a multi-person team scattered around the world

working on pieces of a large project, having a central repository and the ability to track and review changes is essential. Arguably the best version control software right now is Git.


All the EiE firmware is managed in GitHub. You could easily do a whole course on Git, especially if you use the command line access. There are many, many resources online if you want to become a power user. For EiE, the things you need to know are:

1. All code is contained in a “repository.” The repository consists of branches which are instances of the code base.
2. Firmware is organized in a tree-like structure where the “Master” code is at the top of the tree.
3. Branches are created under Master where new development is done or features are added. In this way, the stable, released code remains untouched and new development can proceed without affecting it.
4. Users duplicate the code to their local machines to make changes either explicitly by downloading it, or by checking out a repository through GitHub. This is called “Cloning.”
5. Updated code can be pushed back online or pulled up the tree – Git provides a good way to review changes and pull them into the main code.

There are many variations on how coding, testing, and releasing is done. Organizations will have their own strategy on how to successfully manage all the branches and merges that occur from developers. This is called a branching model. Branching models are well documented online for you to reference.

For this book, we decided to use an upside-down tree structure where we start with a virtually empty project and make a branch for each chapter that builds on the branch above. Obviously, we cannot update this book once it is released, so the eiebook repository is frozen and will never be updated so the code always matches what is shown here. The final version of the code from the book is forked to a different repository where it will continue to evolve with features, updates and bug fixes.

Let’s go through the steps to get the EiE repository and load up the code for the next chapter. Following through this short exercise will ensure you can access the online code and get a local copy on your computer. It is assumed that you have GitHub Desktop installed and can find “eiefirmware” on GitHub. We will use our company GitHub user, “Engenuics” for this example.

 Make sure you are logged into GitHub in GitHub desktop and online. Search for the “eiefirmware” user.

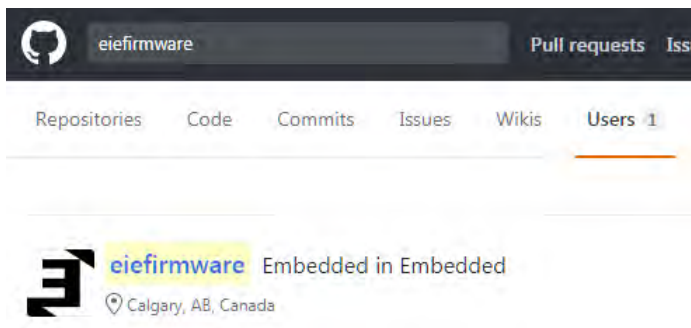
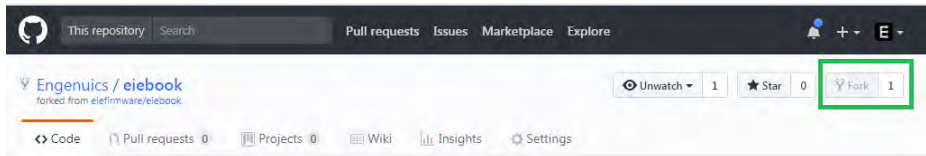


Figure 2-32 GitHub online

Be sure to “Follow” this user so you can be kept up to date. Choose the eiebook repository then click the “Fork” button to copy this into your own account so you will be able to make and track changes.



### Forking eiefirmware/eiebook

It should only take a few seconds.



Figure 2-33 Forking "eiebook" repository

Confirm you are in your account and can see that eiebook has been forked from eiefirmware.

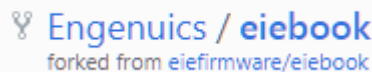


Figure 2-34 Confirmation that repository has been forked



Now Clone the eiebook repository by clicking “Clone or download” and select Open in Desktop. This should automatically open GitHub Desktop on your computer and ask you where to clone the eiebook repository. We recommend using \\EiE\EiE\_Git folder at the top of your data drive. Do not clone onto your desktop or in a Windows user folder as this tends to cause file permissions issues.

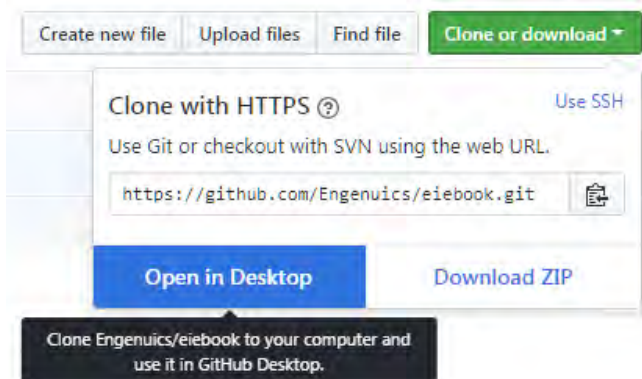


Figure 2-35 Clone eiebook to desktop

In GitHub Desktop, select the “Master” branch. This is our version of the project that was completed in the example above for this chapter.

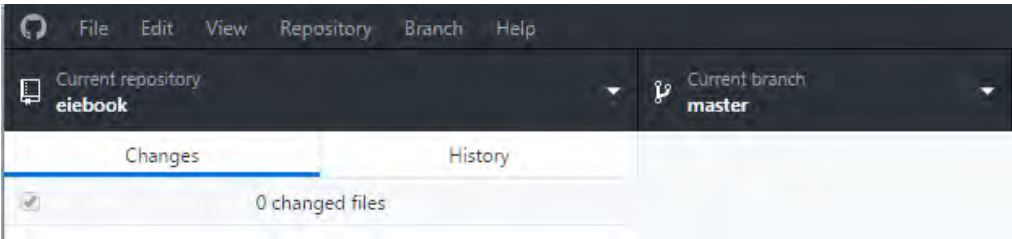



Figure 2-36 Select Master branch

A repository includes all the branches and all the history in making those branches. This is quite amazing if you think about it. That might seem like a lot of space to use, but since code is just text it can be compressed down significantly. The latest video you took of your cat chasing a laser pointer consumes a lot more space.

When you change the current branch, the files in the local Git directory you chose will be updated with the files from the selected branch that Git archives. Whatever branch was active before is saved into the local archive. You can only change branches if all the changes in your current branch have been committed, so you don't have to worry about accidentally losing a bunch of work by changing branches.

 Open the Workspace in IAR by clicking Repository > Show in Explorer. This opens Windows Explorer to the Git folder where the files for this repository are stored. This is a safe way to ensure you pick the right project to work with.

Double click the eie\_ide.eww file to open it. Do not change any files. Close the workspace immediately. Activate the GitHub Desktop Window and you will likely see that some changes have automatically occurred. IAR always changes a few project files whenever you open a Workspace — these are things like window position files and various history files. Since IAR can write project files, or sometimes you forget to save changes in the files you are working with, we highly recommend that you use GitHub Desktop and IAR independently. Always close the workspace in IAR if you are going to do something with Git.

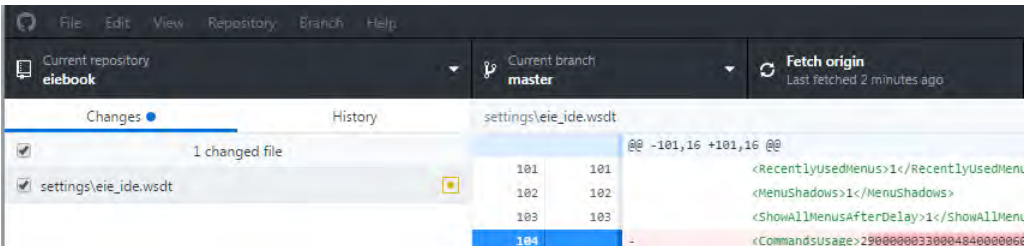


Figure 2-37 Git file changes

Try switching from “Master” to “assembler” branch and you should get an error. You cannot switch branches without doing something about the pending changes in the current branch. Changes must be committed or discarded. If it is just the project files that have changed, we suggest discarding the changes unless you are sure there is something about the project that you want to keep. If the file is plain-text, you can see the changes on the right side of the window. Lines that were changed are shown in red; the new lines



replacing them are shown in green. Right-click where it says “1 changed file” and choose “Discard all changes.” A dialog box will appear to confirm that you really want to delete the changes.



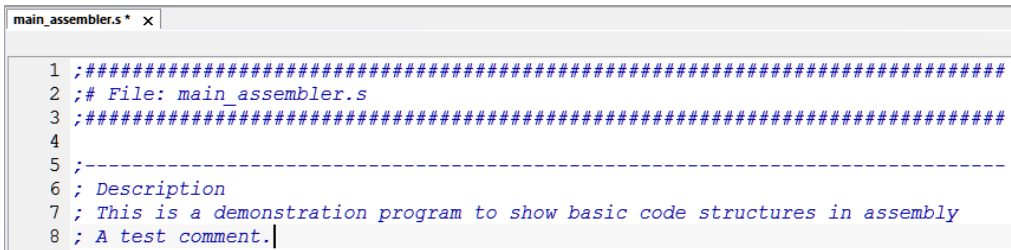
Whenever you discard changes, carefully check the list of files that have changes that will be deleted to make sure there are no files listed that you really want to keep. If you discard changes at this point, these changes are gone forever.

If your current branch has no changes, you can select a different branch from the drop-down. THIS WILL AUTOMATICALLY RETRIEVE THE NEW BRANCH FILES AND REPLACE THE FILES ON YOUR ACTIVE PROJECT. This should still be safe because to take this step, you had to commit your changes (or discard them).

Now you should be able to change the current branch to “assembler.” Choose Repository > Show in Explorer. Launch eie\_assembler.eww. Usually, it is easiest to open using IAR’s recent workspaces list, but this project has a different name so that won’t work. The project name in IAR and the branch name in Git are totally independent of each other even though we often end up naming them the same or something very close.



In IAR, open main\_assembler.s and add a short test comment somewhere in the code, then save the file and close the workspace.



```

1 ;#####
2 ;# File: main_assembler.s
3 ;#####
4
5 ;-----
6 ; Description
7 ; This is a demonstration program to show basic code structures in assembly
8 ; A test comment.

```

Figure 2-38 Add "A test comment" in main\_assembler.s



IAR and Github both have control over the same repository files, so it is easy to end up with complicated problems of files access. Therefore, we recommend always closing the IAR workspace before doing anything in Github.



Switch back to GitHub Desktop and notice three things:

1. main\_assembler.s is in the “changed files” list. Click on it and the window on the right will show you the changes
2. The project files changed again
3. The indicator in the top right now says, “Push origin”

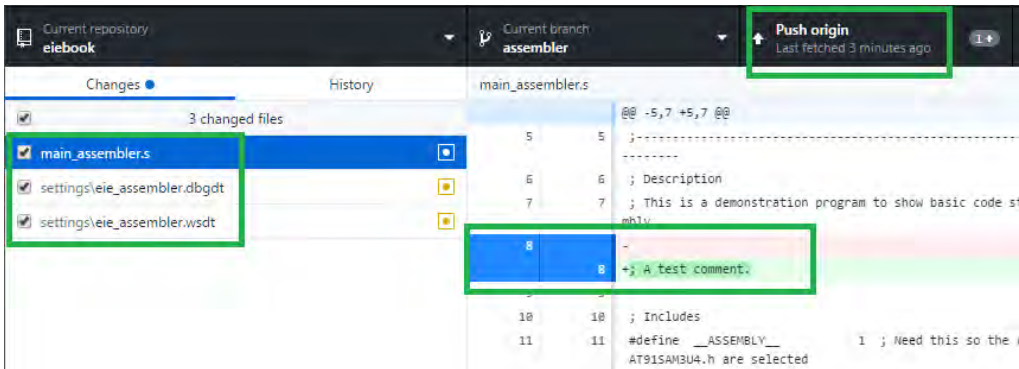



Figure 2-39 Switch back to GitHub

This is all happening locally. To synchronize the local repository with GitHub cloud, you must commit your changes and then push those changes online. This “Push” just means you are copying the changes from the local “assembler” branch into the cloud “assembler” branch. It does not affect the other branches.

 Uncheck the two project files but keep `main_assembler.s` checked. In the “Summary” line, enter “A small change test” then click “Commit to assembler.”

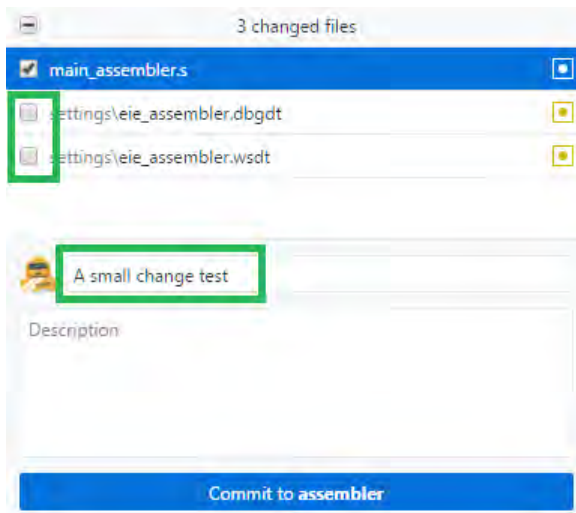

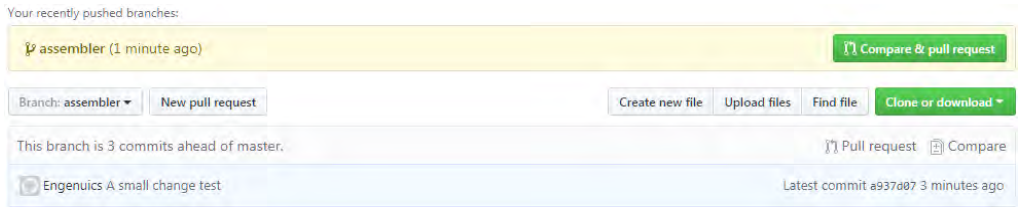


Figure 2-40 Uncheck the project files

 When the commit is done, `main_assembler.s` will no longer be in the change list. Discard the two project file changes. Click “Push origin” to update into the cloud then go to Github.com and refresh the page. You should see your commit information.



**Figure 2-41** Click "Push origin" to update

The change you made is now synchronized on your local machine and in the cloud, but the change only applies to the assembler branch. You can examine the change to see all the files and all the changes that took place in each commit. Different branches can also be compared to see what changes have occurred in each.

If this change was something that other branches should update to, you can do a “Pull request” to alert other branches about the change. You can even alert other users that have forked the same repository. It is up to the person who controls the other branches to review the pull request and decide if the changes should be accepted.

Git will indicate if the changes can be updated automatically, or if there are conflicts that must be resolved first. Conflicts happen when the same line of code has been changed independently in the two branches that are trying to merge. Conflicts are handled almost entirely manually and can be quite a pain. This is where a good process about work distribution and frequency of updates can really help. If you have many developers working on the same code base and no one pushes changes for weeks at a time, dealing with the resulting conflicts could be virtually impossible.

In this case, the Master and assembler branches are substantially different projects so attempting to merge the two makes no sense at all even though it can be done without conflicts. As we start working with the main EiE firmware in subsequent chapters, you will see how the merging process can work beautifully. You can read all the GitHub documentation about how merging works, how code can be reviewed and how conversations between developers can be had. The intricacies of Git take some getting used to, but it really is a fantastic way to control any code you write once you have a handle on it.

You are now ready to start writing some code!



## Chapter 3 • ARM Cortex-M3 Assembly Language

It is our firm belief that having at least a basic knowledge of the instruction set and assembly language of the processor you are using will allow you to be significantly better at writing and debugging firmware even if you never write in assembler directly. This intimate knowledge of how the processor works demystifies a lot about what happens when you click “compile” from a high-level language. There are many ways to write better C code based on your knowledge of the instruction set and processor resources. You can more effectively debug and understand your code if you understand the assembly language. There are some bugs that would be impossible to find without being able to work at the assembly level.

When a high-level language compiler processes source code, it generates the assembly language translation of all the high-level code into a processor’s specific set of instructions. The target processor’s “instruction set” is the set of capabilities that the processor knows how to execute. For example, it is safe to say that every processor has an “ADD” instruction that takes two numbers, sums them, and stores the result somewhere. The implementation might be very different between various micros, but the net result is the same.

Atmel includes a section in their user guide that explains the ARM instruction set in their own words. If you are really interested in the details of the ARM core look for the ARM Core Technical Reference Manual.

Everything we need to learn about Cortex-M3 assembly language is available in the SAM3U user guide in the ARM Cortex M3 Processor section. Basic knowledge about what instructions are available and how they are used is plenty to get you going in the right direction. As you build more complicated programs and want to learn more, you may have to review this section more carefully. For now, you need two things:

1. Core register map
2. Instruction set summary

### 3.1 • Core Registers

ARM cores have what is referred to as a “load-store architecture.” This means that if you want to work on any variable from RAM or peripheral memory, you must load it into the core registers, do what you want to it, then store it back to memory. The instruction set is designed for this architecture, and the core registers provide the memory locations for the core to do its work. This is all hidden if you are writing in C or C++ but is right in your face if you are writing code in assembler. Even though it is hidden by a high-level language, understanding it can be beneficial for writing better code and will certainly help you debugging. The good news is that it is actually really simple because so little occurs with each instruction. Other than the syntax of the instructions, you should have no problem writing some basic assembly code to make things happen on the processor.

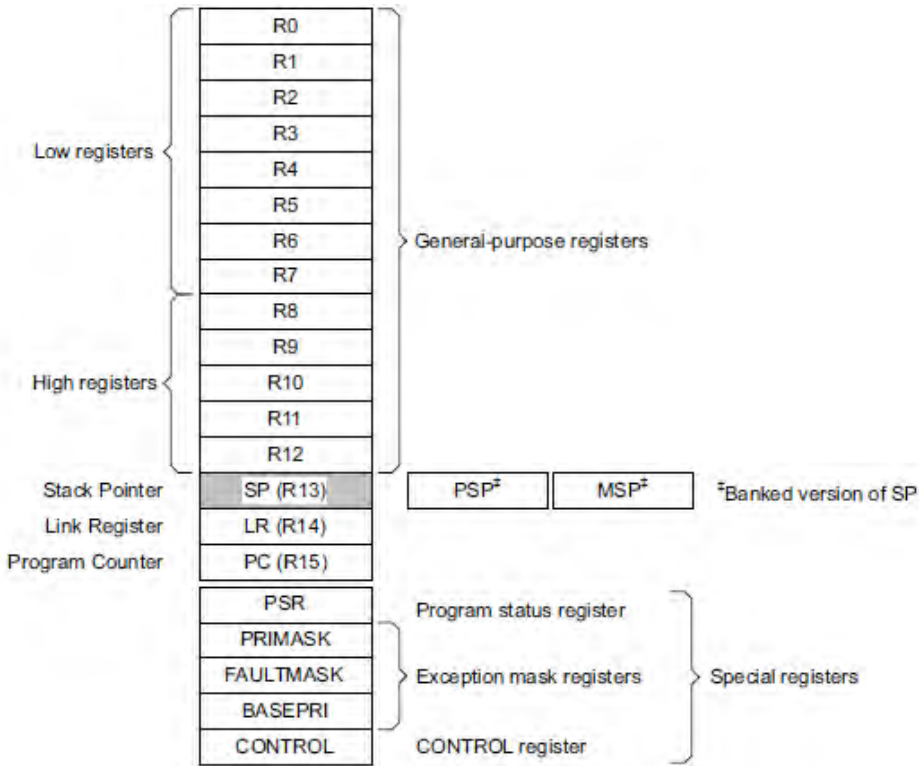


Figure 3-1 Cortex-M3 core registers

The ARM core is directly connected to the core registers which can, therefore, be accessed very quickly. The first 16 registers hold data or addresses that will be used with arithmetic or logical operations. Of these, the first 13 are generic and denoted R0 thru R12. Register R13 is reserved as a stack pointer and R14 is the “link” register that is used to hold the most recent return address for a function call. The Program Counter in R15 is the main pointer that addresses your program from memory. Whatever address is here will be the next instruction executed by the core. The program counter will continually update automatically from hardware every time the clock ticks.

There are a few other registers shown, but the only one we need to worry about is the Program Status Register (PSR). If you read the documentation on the PSR, you see it is actually a combination of three registers that all sort of overlap each other but are effectively mutually exclusive (we are not sure why ARM does it this way, but there is probably a good reason). What we care about are the top four MSBs of the Application Program Status Register (APSR).

31	30	29	28	27	26	25	24
N	Z	C	V	Q	Reserved		

Figure 3-2 Application program status register: most-significant bits of interest

These four bits are essentially how the processor makes decisions.



*In the context of looking at the individual functionality of bits in a register, the bits are often referred to as “flags” or “flag bits.” We say a flag is “set” when it is logic 1, and we say a flag is “clear” when it is logic 0.*

Depending on the instruction that is executing and the result of that instruction, some of these bits may change to indicate to the processor what has just happened as a result of the instruction executing. The bits are:

**N: Negative / less-than.** Set when the instruction result is negative. In comparison instructions, this is used to indicate less-than. The bit is cleared for all positive/greater-than or zero/equal-to results.

**Z: Zero.** Set when the instruction result was zero.

**C: Carry / borrow.** Set when the result over-extends the MSB.

**V: Overflow.** Set when bit 30 carries for use with signed values.

The processor logic can examine these flags and make simple binary decisions by branching based on their values. The easiest example is deciding if two numbers are equal. Instructions would be used to load two registers with the values of interest, and then a subtraction instruction would be used. If the numbers are the same, the result is 0 and the Z flag would be set. Therefore, after this instruction is complete, the processor can look at the Z bit in the APSR and make a branch (i.e. a decision) on what to do next. This is an if-then scenario in its most basic form.

Every processor that we have ever worked with has some form of status register like the APSR and has the same N, Z, C and V flags. If you are coding in assembler, you must understand how the status flags work and how to use instructions that can reference them.

### 3.2 • Instructions

Instructions are the binary machine language that makes the processor do everything it does. This is the essence of the logical-to-physical translation that occurs inside the processor. The bits that make up an instruction in flash memory are logic high and logic low voltage levels that appear on the bus when the instruction is being executed. These ones and zeroes cause the logic gates inside the processor to turn on or off depending on the ones and zeros. The net result is that the processor does something in hardware which results in more logic high and logic low voltages levels propagating through logic to create the result in whichever destination location was configured by the instruction. This happens through several cycles called fetch, decode, execute and write-back.

Instructions are almost always “pipelined” which means that while one instruction is at the write-back phase, the next instruction is already executing, the instruction after that is decoding, and the instruction after that is being fetched. So, after the first four instructions in a program have finished, we say the “pipeline is full” and the processor is running as fast as it can. There should be a big asterisk there because this depends on whether or not the pipeline can stay full and execution does not require other operations – but that goes beyond what we want to cover here.

For Cortex-M3 cores, the instruction set is a mix of 16-bit and 32-bit instructions that can be used simultaneously. This is actually quite amazing and has a long history. Another amazing thing is that there are only about 100 instructions in total. So somehow, some way, even devices as complicated as game-playing smartphones can break down all of the tasks they need to do into combinations of just a few different actions. Now would be a great time to quickly glance at the instruction set summary in the SAM3U2 user guide.



### 3.3 • Assembly Language Syntax

Just like in C, there are syntax rules for assembly and standard ways of formatting lines of code. In assembly language, there are fewer rules because there are a limited number of expressions that can be written. Each line of assembly code always has four fields: Label, Instruction Mnemonic, Operands, and Comments. The format looks like this:

Label	Instruction Mnemonic	Operands	; Comment
-------	----------------------	----------	-----------

The Label field is optional and can appear on the line above. Labels will help organize your code and allow you to branch/jump around in the program. Each line of assembly language translates into an instruction that will live at a program memory address. If a line is labeled, the label becomes a symbol to which the corresponding program memory address is assigned when the code is assembled. Therefore, you can specify labels in your code – you are just indicating a name for an address.

The Instruction Mnemonic is any one of the reserved words listed in the Instruction set (e.g. ADC, ADD, MOV). Most mnemonics are self-explanatory for what they do, but in case you cannot figure it out they are described in the instruction listing. What is not self-explanatory are the optional flags or conditions that you can set to each instruction. If you have looked at other instruction sets, you will quickly recognize how comparatively powerful the ARM instruction set is thanks to what a single instruction can do.

Operands refer to register locations, addresses, or constant values that an instruction uses. Some instructions do not require any operands, while others require 3 or more. For example, an “ADD” instruction needs two values to add and somewhere to store the result, so it needs 3 operands. ARM operands can also contain information that causes additional things to happen in the processor during the instruction cycle.

The comment field is optional other than the semi-colon. Comments are just text and are entirely ignored by the assembler just like in C. Assembly language is difficult to look at and understand what is supposed to be happening, so extensive commenting becomes essential for writing good code. It is not uncommon to comment every single line of assembly code written with an explanation or pseudo code of what is supposed to be happening.

As an example, look at a simple move instruction that has the mnemonic MOV. Reading the instruction set for MOV, you see the following detail:

Label	Mnemonic	Operands	Comment
move_eg	MOV[cond][s]	Rd, <Op2>	; Move syntax

Any text written in square brackets [ ] denotes that it is optional, and any text written in angle braces <> is required but has several options. Almost every instruction has a condition field [cond] that makes that line conditionally executable based on the APSR flags. If you specify a condition, the processor will determine if the instruction should be carried out or skipped. This happens before the instruction is fetched, and does not cost any cycles to evaluate which is actually quite incredible, especially if you have worked with other processors that have to use separate instructions for condition testing. To use the [cond] code in an instruction, the instruction is preceded by an “IT” (if-then) instruction that sets up the logic for the possible branch.

Label	Mnemonic	Operands	Comment
simple_move	MOV	<operands>	; Move version 1
setup_cond	IT	PL	; Setup for condition
cond_move	MOVPL	<operands>	; Move version 2

Looking at the code above, the first MOV instruction will move the values specified in the operands and regardless of the result in the destination register, the APSR flags will not be touched. The second MOV instruction is suffixed with “PL” which tells the processor to only execute the move if the last instruction that updated the APSR flags resulted in a non-negative (positive or zero) value. This instruction must have the IT instruction with PL as an argument to set up the logic. In this small code snippet, there is no way to tell what the APSR flag values are, so we cannot say for sure if the MOVPL instruction will execute.

To set the APSR flag values, most instructions can add the [s] modifier which tells the processor to update the APSR flags after the instruction has been executed. Being able to control the flag updates like this is another very powerful feature of the ARM architecture.

Label	Mnemonic	Operands	Comment
Logic	ANDS	<operands>	; AND with flag update
setup_cond	IT	PL	; Setup for condition
cond_move_s	MOVPLS	<operands>	; Move version 3

Now the MOVPLS instruction is conditionally executed based on the result of the ANDS instruction. The APSR flags will be updated again depending on the result stored in the destination register. This shows that one instruction can have at least 3 variations even before considering the operands. Note that order of the conditions and flags is indeed important – MOVPLS is valid whereas MOVSP is not.

The operands that must be used with each instruction type can be complicated – this is why you must have your instruction set readily available when writing code. Instructions that work with data typically specify a destination register and one or two source registers. The operands required for a MOV instruction, are listed as:

Rd, <Op2>

This states that a move instruction requires a specified destination register, Rd, and some sort of second operand, Op2, which must be the source of the data to move. You will see in other instructions registers denoted Rs (source register) and Rm and Rn (other register arguments).

Recalling the registers available, Rd can be anything from r0 to r15, though remember that r13, r14, and r15 are reserved and should not be used in most cases. This leaves r0 through r12 as options. Depending on what you intend to do with the data, you might strategically choose Rd to support the next instruction. For this standalone example, r0 will be used.



If you are mixing assembly code with C code source files, you must be aware of how the compiler will use registers.

The second operand has several optional forms and thus takes a bit longer to explain. Nearly all the Data Processing instructions involve <Op2> as an operand like this. Fortunately, once you learn all the variations of Op2 for the MOV instruction, you can apply that verbatim to the others.

Table 3-1 on page 96 summarizes what Op2 can be. It also describes the conditions that have to be imposed on some of the options. Notice that other than the first case where Op2 is an immediate value, Op2 has two components. Rm is the register to use, and this, in turn, can be used directly or first shifted in one of five ways. The content of Rm remains the same – only the value used in the instruction is shifted. Please see the user guide to understand exactly how the different shifts work.

**Table 3-1 Summary of Operand 2 functions**

Operand 2	
Immediate value	Allowed value
Register, logical shift left by number	Rm, LSL #
Register, logical shift right by number	Rm, LSR #
Register, arithmetic shift right by number	Rm, ASR #
Register, rotate right by number	Rm, ROR #
Register, rotate right one bit with extend	Rm, RRX
Optional shifts (opsh):	
No shift: Rm (same as Rm, LSL #0)	
Logical shift left: Rm, LSL #<shift> where allowed shifts are 0-31	
Logical shift right: Rm, LSR #<shift> where allowed shifts are 1-32	
Arithmetic shift right: Rm, ASR#<shift> where allowed shifts are 1-32	
Rotate right: Rm, ROR #<shift> where allowed shifts are 1-31	
Rotate right with extend: Rm, RRX	

An "immediate" is just a number, so why does the table show "allowed value" instead of just saying "any number?" well, to make this even more exciting there are conditions on the constant value because the number itself must be stored as part of the instruction while still leaving room for the other information that the instruction must contain. If the number was stored directly in the instruction, only a few bits could be used since the other bits in the instruction is the instruction itself. So instead, numbers are encoded and thus only certain numbers can be loaded directly. If the number cannot be encoded in the instruction, the full value must be stored somewhere else in flash which is often called a "literal pool". Yes, this is complicated to understand, but even if you do not get the "why" just remember the "what". The numbers that can be encoded in Op2 of an instruction are:

- any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word
- any constant of the form 0x00XY00XY
- any constant of the form 0xXY00XY00
- any constant of the form 0xXYXYXYXY.

For other instructions where a number is involved, the exact size of the number allowed depends on the instruction. It turns out that the numbers available are quite useful and there are not too many times during a regular program that the number you want cannot fit in the instruction. If you are writing assembly code and try to use a number that is not available, the assembler will flag an error. In this case, you must use multiple instructions and a memory location to store a constant and retrieve it for use. If you are writing code in C, the compiler will figure all of this out for you. It just happens that writing numbers that do not fit in the instruction end up costing more flash space and more instruction cycles to retrieve. By knowing this, you can save these resources.



The instructions can be explained, but it will be much more meaningful to enter and test all the upcoming examples in IAR. Select the "assembler" branch in Github and open the workspace in IAR. Open main\_assembler.s and enter all the example code in the "main" section.

If you wanted to load the value 10 into r0, you would write the instruction like this:

Label	Mnemonic	Operands	Comment
move_eg	MOV	r0, #10	; r0 = 10



Type this in and try it to see what happens. You should see 0xA load to R0. Note that you must single step on each instruction (F11).

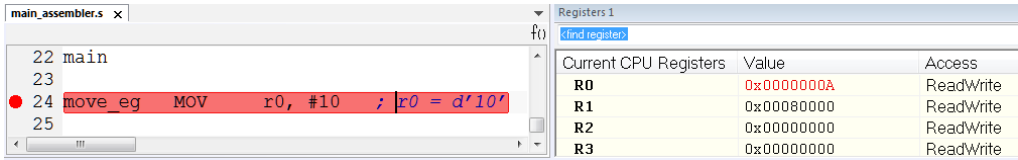


Figure 3-3 Main assembler example

Here are some examples assuming that r1 will be the Rm register for Op2. Every shift left is equivalent to multiplying by 2; every shift right is equivalent to dividing by two and throwing away the remainder.

Label	Mnemonic	Operands	Comment
	MOV	r0, #10	; r0 = 10
	MOV	r1, r0	; r1 = r0
	MOV	r0, r1, LSL #4	; r0 = r1 * 16
	MOV	r0, r1, ASR #1	; r0 = signed(r1 / 2)

You can also shift by a number held in a register. Examples are as follows:

Label	Mnemonic	Operands	Comment
	MOV	r2, #8	; r2 = 8
	MOV	r0, r1, LSL r2	; r0 = r1 x 2^r2
	MOV	r0, r1, ASR r2	; r0 = signed(r1 / 2^r2)

Right now it will not be clear as to why you would need so many similar options (and some that may seem pointless) for a simple move instruction. However, if you start thinking about some more complex operations or scenarios where you are moving numbers around and performing certain calculations, you might start to see how this can come in handy.

Let's look at a longer example. As each step is taken, the register contents will be shown in the notes, but you should follow along with the debugger to see it working. A big part of this exercise is getting more familiar with using the IAR debugger. Note we use two Watch windows plus a Register window so we can see the core register locations in hex, decimal, and binary. The project should debug on your development board, or you can change the project options and work on the simulator here.



*To be able to view a register value in multiple formats (e.g. to look at R0 in decimal and binary), you must use two different types of Watch windows (e.g. a Watch window and a Live Watch window). You cannot just specify the same variable twice in the same window, nor can you use different windows of the same type because IAR will automatically show the variables with the same representation. You might think you have it set up so it will work with the same type of window, but as soon as you step in the code it will change.*

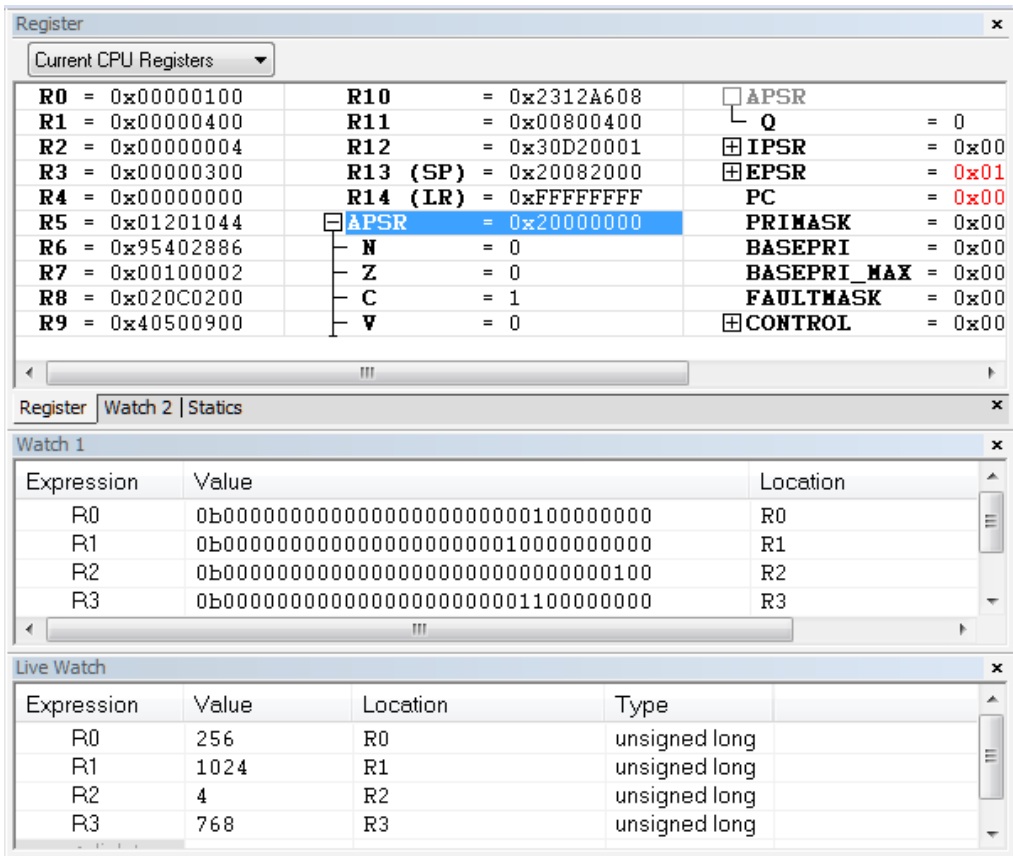


Figure 3-4 Setting up register views

We start after the simple example but let's assume we have no idea what any of the register values are yet, so we draw our memory like this:

r0	r1	r2	r3	NZCV
?	?	?	?	?



First, initialize registers r0, r1 and r2 like this:

```
MOV    r0, #0      ; r0 = 0
MOV    r1, #0      ; r1 = 0
MOV    r2, r1      ; r2 = r1
```

We are not using MOVs instructions, so we cannot be sure what the NZCV flags are doing (they will be set based on whatever last instruction set them, which we do not know). The memory that we know looks like this:

r0	r1	r2	r3	NZCV
0	0	0	?	?



Now zero r3 and use the “S” flag to force an update to the APSR flags:

```
MOVS      r3, #0                ; r3 = 0, update APSR
```

Now we have the registers of interest initialized to known values. This, of course, is exactly what you do in a high-level language – get your system in a known state. Now we can safely proceed with the example to use different instructions to change the memory locations and demonstrate the instruction set and processor operation.



Next, fill up r0, r1 and r2 with more interesting numbers. MOV instructions could be used, but we will make life more interesting with some ADDs. It is acceptable to use one of the source registers as the destination register as shown:

```
ADD      r0, r0, #256           ; r0 = r0 + 256
ADD      r1, r1, r0, LSL #2     ; r1 = r1 + (4 x r0)
MOV      r2, r0, LSR #6         ; r2 = r0 / 64
```

r0	r1	r2	r3	NZCV
256	1024	4	0	0110



Now try a subtract with the “S” flag:

```
SUBS      r3, r1, r0            ; r3 = r1 - r0, update APSR
```

r0	r1	r2	r3	NZCV
256	1024	4	768	x0xx

The other flags will update, but focus on the zero flag for now: the “x” represents don’t care.



The following instruction uses a condition code and will only be executed if the result of the previous instruction (that is, the previous instruction that updates APSR) was zero. Since it was not, this line of code will not actually be executed and r3 will remain 768. If you are stepping through with the debugger, the program counter will point to the MOVEQ line but when you step nothing happens except the program counter goes to the next line. The IT instruction is required to set up for this conditional execution.

```
IT      EQ                ; Get ready for a conditional
MOVEQ   r3, #1024         ; if {Z}, r3 = 1024
```


r0	r1	r2	r3	NZCV
256	1024	4	768	x0xx



Next, do a reverse subtraction with a left shift of r0 by 4 (which is the same as multiply by 16). “Reverse” just means that instead of doing operand1 – operand2, the processor will do operand2 – operand1 which might be important to you one day.


```
RSBS      r3, r1, r0, LSL #4    ; r3 = (16 x r0) - r1
```

r0	r1	r2	r3	NZCV
256	1024	4	3072	x0xx

 Now do a multiplication instruction conditionally executed as long as the RSB did not result in 0. Since the RSB resulted in a non-zero number, the MULNE instruction executes and r3 is updated.


IT	NE			
MULNE	r3, r2, r0			; Get ready for a conditional ; r3 = r2 x r0

r0	r1	r2	r3	NZCV
256	1024	4	1024	x0xx

 Try a comparison. Comparisons simply “look” at the values in the register to update flags, so no registers will change their value as a result of this instruction other than the APSR (even though the processor will actually subtract the two numbers). You do not have to specify the “S” flag as it is implied by the instruction that you want to update APSR.

CMP	r3, r1			; Set flags if r1 == r3
-----	--------	--	--	-------------------------

r0	r1	r2	r3	NZCV
256	1024	4	1024	0110

 For the last example case, use a logical bit-wise OR conditionally executed on the CMP being equal, and update the flags on the result.

IT	EQ			
ORREQS	r3, r0, r1, LSR #10			; Get ready for a conditional ; r3 = r0   (r1 / 1024)

r0	r1	r2	r3	NZCV
b'1 0000 0000'	b'0100 0000 0000'	4	b'1 0000 0001'	0000

The memory is shown in binary for r0 and r3 for clarity. The number 1024 in binary shifted by 10 bits to the right is just 1. So, the result of the instruction is what you see in r3.

This example has shown most of the different options available to code an instruction and given lots of samples of Data Processing instructions that operate only on the core registers. Until we are actually working towards a goal for a program, it will not be entirely clear how all of these instructions are going to come together to make something useful happen. In fact, Data Processing instructions by themselves will not suffice to write a program. We have to get into the memory access instructions to load and store memory to and from RAM, ROM, and peripheral registers.



### 3.4 • Load and Store Instructions

ARM is classified as a load-and-store architecture. The instruction set allows data processing to occur only on the core's registers, so there must be a way to get data in and out of the core from RAM or ROM. There are quite a few different load and store instructions but we will just look at the two simplest cases.

If you want to access a memory location, you must know where it is. The 32-bit address of the memory location you want must be loaded into a register, which requires an immediate to specify the address. What is not so obvious is how to load that number due to the limitations of what numbers can be stored in an instruction vs. numbers that must be stored as constants and accessed by reference.

LDR is “load register from memory” and STR is “store register to memory.” In their simplest forms, they use a destination register and a pointer to a memory location as arguments. For notation, a pointer is shown as the register in square brackets, like this: [r1].



Continue the same program you have already been working on. Adjust the debug environment so you can see a Register window and both the Memory and Symbolic Memory windows. In both memory windows, enter “0x20000000” in the “Go to” areas so that the windows show the memory addresses we want to look at.

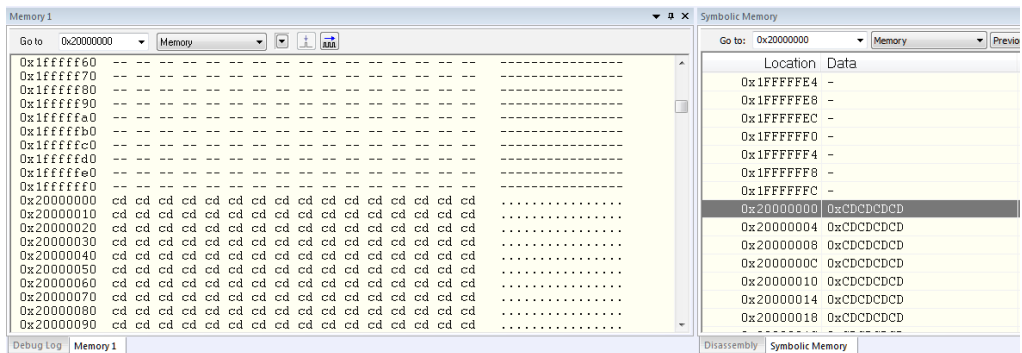


Figure 3-5 Memory and Symbolic Memory windows

The example will use registers r0 and r1, and RAM locations 0x2000 0000 and 0x2000 0004. We know where valid RAM locations are from the “Memory model” section of the user guide.

Assume our memory starts out like this:

r0	r1	0x2000 0000	0x2000 0004	NZCV
?	?	?	?	????

First, make r1 point to the RAM space by loading the RAM address:

```
MOV      r1, #0x20000000    ; r1 = the target address (this
                             ; is a valid immediate)
```

Note that there is nothing that tells r1 that the value it has been loaded with is a pointer – we just know that is what that number will be used for. Think of a child with a stick. To you it's a stick, to the child it could be a sword or a magic wand.

We want the number 5000 to be loaded into the RAM location 0x20000000. The first step is to get the number 5000 into a register, so can we do the following?

```
MOV    r0, #5000          ; r0 = 5000 ?
```

In this case, yes, because a move instruction can take as high as a 16-bit number. If a larger number is needed, an LDR instruction is used that will automatically do two things: store the number in a flash location and then update the instruction to access that location to get the value into a core register. The location where this number is stored is known as the “literal pool” where a bunch of constants hang out in flash memory (they are written there as data locations by the assembler when you build code that requires constants). The syntax is:

```
LDR    r0, =5000          ; r0 = 5000
```

Look at the Disassembly of the instruction above to see how it is actually encoded. See if you can “find” where the value 5000 is. Regardless, the memory is now:

r0	r1	0x2000 0000	0x2000 0004	NZCV
5000	0x2000 0000	?	?	0000

Dereference the pointer in r1 to store the value from r0 into RAM. The syntax requires square brackets around the r1 mnemonic to indicate you want to use its contents as an address pointer. The typical destination, source operand order is also reversed for an STR instruction:

```
STR    r0, [r1]           ; *r1 = r0
```

r0	r1	0x2000 0000	0x2000 0004	NZCV
5000	0x2000 0000	5000	?	0000

If you look at your memory windows (and have correctly set them to show you address 0x20000000), you should see the new value written. Notice two things:

1. 0x1388 is hex for 5000 (this is where number conversions start to get handy)
2. The value is 32-bits and stored Little Endian (lowest byte first in memory)

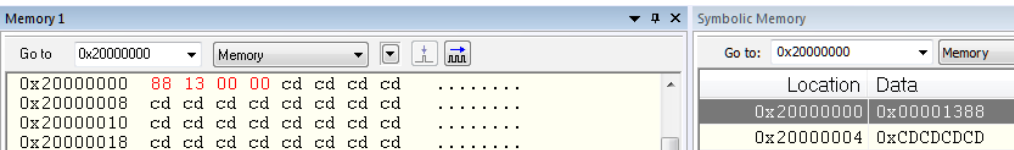


Figure 3-6 Memory and Symbolic Memory windows

Now do a bunch of other operations to ensure you are comfortable with loading and storing. See if you can predict the memory changes for each instruction before checking with the debugger.

Multiply the value in r0 by 4 using an LSL shift instruction. Update r1 to the address of the second RAM register we want to use by adding an offset (4 bytes) to the next 32-bit memory address. Store the multiplied value in r0 to the second RAM addressing by dereferencing r1.



```

LSL      r0, r0, #2          ; Multiply r0 by 4
ADD      r1, r1, #4          ; r1 += 4 (new RAM address)
STR      r0, [r1]            ; *r1 = r0

```

r0	r1	0x2000 0000	0x2000 0004	NZCV
20000	0x2000 0004	5000	20000	0000

Lastly, get the value stored in RAM at 0x20000004 and put it in r3. Stop your code from running wild by putting a “branch” instruction back to the start of main:



```

LDR      r3, [r1]            ; r3 = *r1
B        main                ; Repeat infinitely

```

LDR and STR in their simplest form are indeed easy using registers as pointers. Any value that is stored in RAM that is needed must be loaded to the core registers, used as required and then stored back.

That is enough information to give us what we need to be able to load peripheral registers in the next section, so we will leave it at that even though you could easily spend a whole course learning all the ARM instructions and figuring out how to use them effectively to write large programs.

### 3.5 • Hello World in Assembly

With the knowledge you now have of ARM assembler, you can complete the classic “Hello World” program that practically every embedded designer will do when starting a new design: a blinking LED. You can choose the LED to blink (we’ll use blue). We will also add code to turn on a different LED whenever BUTTON3 is pressed.

This will demonstrate some basic code concepts like IF statements and function calls using assembly language and should give you a bit more insight into what is happening as a program runs. It also provides a brief introduction to how logical signals in the processor translate to physical signals on the development board. That’s not the focus here, though. These concepts are thoroughly covered a few chapters from now, but peeking now will help get your mind ready for it. We will be brief, so you may want to revisit this chapter after learning more about input and output registers.

The first thing you need to know is that each pin on the microcontroller is controlled by certain registers. In some cases, the 0s and 1s in these registers will physically appear as high and low voltages on the pins, which is how a line of code that executes in the microcontroller can do something like turn on an LED. The registers for pin control are grouped together in what we call a “peripheral” which is a block of hardware on the processor responsible for the pins. The peripheral for controlling the pins we need is called “PIOB” which stands for “Peripheral Input Output (Port) B”. “Ports” are just groups of pins.

There are eight discrete LEDs on the development board that are controlled by this peripheral. BUTTON3 is the button on the far right of the board and also controlled by PIOB. In total, there are 9 signals, and each has a related bit position in the PIOB registers. We know these positions by looking at their pin connections on the schematic.

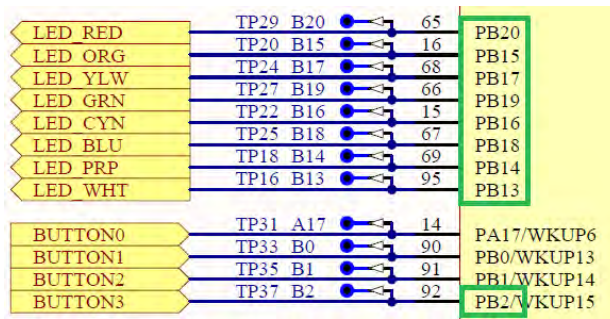



Figure 3-7 LEDs and buttons on the development board

 The LEDs are at bits 13 – 20 (PB13-PB20), and BUTTON3 is at bit 2 (PB2). To help make our code self-explanatory (the correct term is “self-documenting”), define some symbols for these bits that explain exactly what and where they are. Write these just after the “PUBLIC main” declaration in main\_assembler.s:


```
; “PUBLIC” declarations make labels from this file visible to others
PUBLIC main

;-----
; Constants
#define PB_20_LED_RED      0x00100000
#define PB_19_LED_GRN     0x00080000
#define PB_18_LED_BLU     0x00040000
#define PB_17_LED_YLW     0x00020000
#define PB_16_LED_CYN     0x00010000
#define PB_15_LED_ORG     0x00008000
#define PB_14_LED_PRP     0x00004000
#define PB_13_LED_WHT     0x00002000

#define PB_02_BUTTON3     0x00000004
```

For the peripheral to work properly, the registers that define its behavior must be set up. Don't worry about what the registers do yet, just know that they are just memory locations at specific addresses. All the registers are at addresses relative to the PIOB base address, which is a number that has the symbol AT91C\_PIOB\_PER. If you care, you can look up the actual address for that symbol in the AT91SAM3U4.h file and see that it is 0x400E0E00. Each register we need also has a symbol with its address in the header file. To get the offset which will be used when we write the assembly language, we take the absolute address of the register we want and subtract the absolute base address. Define all the addresses we need as offsets in main\_assembler.s just after the PIOB definitions above.

```
; Register address offsets
#define PER_OER_OFFSET     (AT91C_PIOB_OER - AT91C_PIOB_PER)
#define PER_OWER_OFFSET    (AT91C_PIOB_OWER - AT91C_PIOB_PER)
#define PER_SODR_OFFSET    (AT91C_PIOB_SODR - AT91C_PIOB_PER)
#define PER_CODR_OFFSET    (AT91C_PIOB_CODR - AT91C_PIOB_PER)
#define PER_ODSR_OFFSET    (AT91C_PIOB_ODSR - AT91C_PIOB_PER)
#define PER_PDSR_OFFSET    (AT91C_PIOB_PDSR - AT91C_PIOB_PER)
```

 The bits in each register control the PIOB peripheral. We must initialize the values in certain registers to tell the processor what hardware is connected to them. All of this is covered extensively in the GPIO and LED Driver chapter. For now, add the following after

the register offsets. Make sure you include the comments! It is critically important to document an assembly file well because they are inherently cryptic.

```

;-----
; Power Management Controller Configuration
; PMC enable register init (set to turn on power to Port B)
#define PMC_PCER_INIT          (1 << AT91C_ID_PIOB)

;-----
; Port B Configuration
; PIO enable register (set to make pin GPIO)
#define PIOB_PER_INIT          (PB_17_LED_YLW | PB_18_LED_BLU | PB_19_LED_GRN | \
                                PB_20_LED_RED | PB_16_LED_CYN | PB_15_LED_ORG | \
                                PB_14_LED_PRP | PB_13_LED_WHT | PB_02_BUTTON3)

; Output enable (data direction) registers (set to make pin an output)
#define PIOB_OER_INIT          (PB_17_LED_YLW | PB_18_LED_BLU | PB_19_LED_GRN | \
                                PB_20_LED_RED | PB_16_LED_CYN | PB_15_LED_ORG | \
                                PB_14_LED_PRP | PB_13_LED_WHT)

; Output write enable (set to allow output writes to ODSR)
#define PIOB_OWER_INIT         (PB_17_LED_YLW | PB_18_LED_BLU | \
                                PB_19_LED_GRN | PB_20_LED_RED)

; Output set registers (set to make output pins low)
#define PIOB_CODR_INIT         0xFFFFFFFF

; Watchdog timer control
#define WDTMR_INIT             0x00008000

```

You can read the comments and get a basic understanding of what is happening, though it is not yet supposed to make a lot of sense. For `PMC_PCER_INIT`, the number is created by taking '1' and shifting it to the correct position. The number of shifts is `AT91C_ID_PIOB` which is another value from the big header file that is the `PIOB`'s peripheral number relative to the other peripherals in the processor.

We have the values we need to write the code now. The structure we will follow is a structure that every embedded system will have. When the processor starts up, there are some things that must be done only once to configure the processor. Once the configuration is done, then a main loop is entered that carries out the main function of the program. This loop runs forever as it does not make sense for an embedded system to ever stop.

Use the `WDTMR_INIT` value that we defined to load a register called `AT91C_WDTC_WDMR`. This turns off a special circuit in the microcontroller called a Watchdog Timer. See the next chapter to learn more. To write the register with the value we want, the value is loaded in `r0`, the target register address is loaded in `r1`, and then the value from `r0` is stored into the address pointed to by `r1`.



```

; Hello World program
init
    LDR    r0, =WDTMR_INIT          ; Load r0 with the init constant
    LDR    r1, =AT91C_WDTC_WDMR     ; Load r1 WDTC_WDMR register address
    STR    r0, [r1]                 ; *r1 = r0;

```

Next, we will load a register called `AT91C_PMC_PCER` with `PMC_PCER_INIT` using the same core registers `r0` and `r1`. See if you can write this code without looking at the solution!



```
LDR    r0, =PMC_PCER_INIT      ; Load r0 with the init constant
LDR    r1, =AT91C_PMC_PCER     ; Load r1 with address of PMC_PCER
STR    r0, [r1]                ; *r1 = r0;
```

Now use r2 for the address of AT91C\_PIOB\_PER and load the PIOB\_PER\_INIT value. Again, do this without looking at the code below if you can.



```
LDR    r0, =PIOB_PER_INIT      ; Load R0 with the inti constant
LDR    r2, =AT91C_PIOB_PER     ; Load R2 with address of PIOB_PER
STR    r0, [r2]                ; *r2 = r0;
```

Three more PIOB registers still need to be initialized, but since we have the base address of PIOB in r2, we do not have to waste instruction cycles to keep loading it with new register values. Instead, we can encode the offsets we defined to reach the other PIOB registers we need. The PIOB\_OER\_INIT value is used for both the OER and OWER registers.



```
LDR    r0, =PIOB_OER_INIT      ; Load R0 with the init constant
STR    r0, [r2, #PER_OER_OFFSET] ; *(r2 + PER_OER_OFFSET) = r0
STR    r0, [r2, #PER_OWER_OFFSET] ; *(r2 + PER_OWER_OFFSET) = r0
```

Now load PIOB\_CODR\_INIT into r0 and store it to PER\_CODR using the offset.



```
LDR    r0, =PIOB_CODR_INIT      ; Load R0 with the init constant
STR    r0, [r2, #PER_CODR_OFFSET] ; *(r2 + PER_SODR_OFFSET) = r0
```

This completes the initialization needed. If you have been working in the simulator, attach your development board and change the Project Options (right-click on eie\_assembler) to set the Debugger Driver to “J-Link/J-Trace” and check “Run to” main.

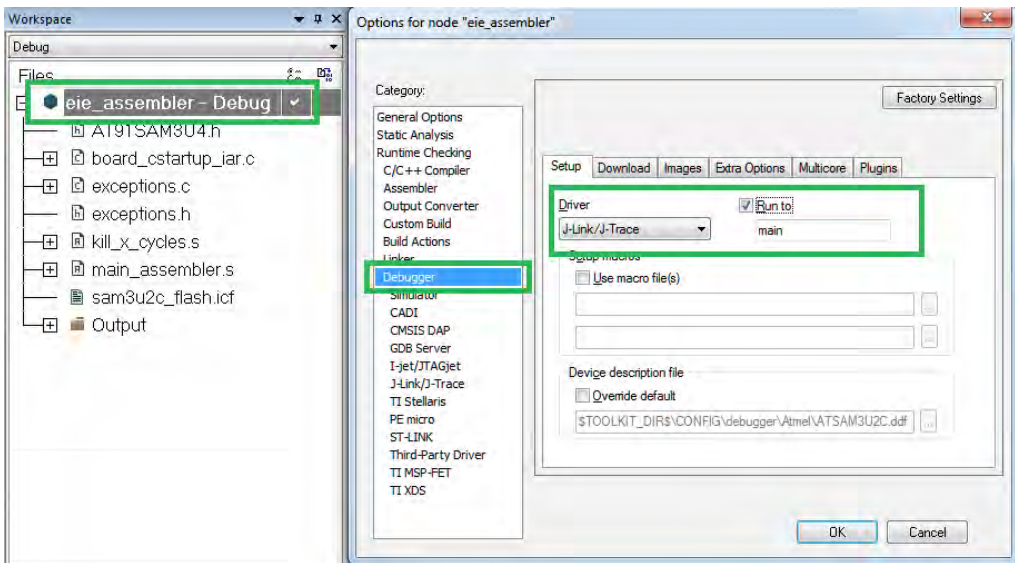


Figure 3-8 Setting Debugger to J-Link

Build the code and start the debugger. You are not going to see too much happening on the development board for most of the lines of code, but you can look at the changes in

the processor registers. Put a breakpoint on the LDR instruction after init and set up two Register view windows. The first should have the Current CPU registers and the second should show the PIOB group.

The screenshot shows the IAR Embedded Workbench IDE with the following components:

- Assembly Code Window (main\_assembler.s):**

```

135 init
136 LDR r0, =WDTCR_INIT ; Load r0 with the init const
137 LDR r1, =AT91C_WDTC_WDMR ; Load r1 with address of the
138 STR r0, [r1] ; *r1 = r0;
139
140 LDR r0, =PMC_PCR_INIT ; Load r0 with the init const
141 LDR r1, =AT91C_PMC_PCR ; Load r1 with address of PMC
142 STR r0, [r1] ; *r1 = r0;
143
144 LDR r0, =PIOB_PER_INIT ; Load r0 with the (location
145 LDR r2, =AT91C_PIOB_PER ; Load r2 with address of PIO
146 STR r0, [r2] ; *r2 = r0;
147
148 ; From here, r2 will stay on AT91C_PIOB_PER so we do not have to waste
149 ; to load it with different values. Instead, we can just encode the o
150 ; instructions we use to access the other PIOB registers.
151 LDR r0, =PIOB_OER_INIT ; Load r0 with the init const
152 STR r0, [r2, #PER_OER_OFFSET] ; *(r2 + PER_OER_OFFSET) = r0
153 STR r0, [r2, #PER_OWER_OFFSET] ; *(r2 + PER_OWER_OFFSET) = r0
154 LDR r0, =PIOB_CODR_INIT ; Load r0 with the init const
155 STR r0, [r2, #PER_CODR_OFFSET] ; *(r2 + PER_SODR_OFFSET) = r0
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172

```
- Registers 1 Window:**

Current CPU Registers	Value	Access
R0	0x00004E20	ReadWrite
R1	0x20000004	ReadWrite
R2	0x00000004	ReadWrite
R3	0x00004E20	ReadWrite
R4	0x00000004	ReadWrite
R5	0x054142A4	ReadWrite
R6	0x0140060C	ReadWrite
R7	0x00000000	ReadWrite
R8	0x0B550025	ReadWrite
R9	0x0000001D	ReadWrite
R10	0x440C0522	ReadWrite
R11	0x00024000	ReadWrite
R12	0xFFFFFFFF	ReadWrite
APSR	0x00000000	ReadWrite
IPSR	0x00000000	ReadWrite
EPSR	0x01000000	ReadWrite
PC	0x00000060	ReadWrite
SP	0x2007D000	ReadWrite
LR	0x000001AB	ReadWrite
- Registers 2 Window (PIOB group):**

PIOB	Value	Access
PIOB_PER	0xFFFFFFFF	WriteOnly
PIOB_PDR	0xFFFFFFFF	WriteOnly
PIOB_PSR	0xFFFFFFFF	ReadOnly
PIOB_OER	0xFFFFFFFF	WriteOnly
PIOB_ODR	0xFFFFFFFF	WriteOnly
PIOB_OSR	0x00000000	ReadOnly
PIOB_IFER	0xFFFFFFFF	WriteOnly
PIOB_IFDR	0xFFFFFFFF	WriteOnly
PIOB_IFSR	0x00000000	ReadOnly
PIOB_SODR	0xFFFFFFFF	WriteOnly
PIOB_CODR	0xFFFFFFFF	WriteOnly
PIOB_ODSR	0x00000000	ReadWrite
PIOB_PDSR	0x00000000	ReadOnly
PIOB_IER	0xFFFFFFFF	WriteOnly
PIOB_IDR	0xFFFFFFFF	WriteOnly

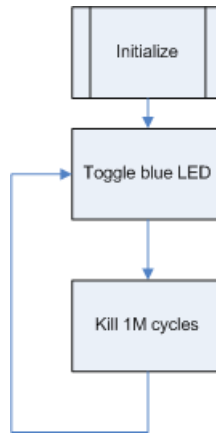
Figure 3-9 CPU registers

Step through each line of code and predict what will happen before each step. On reset, the development board should initialize with the white, purple, blue, and cyan LEDs on fully, and the other four just dimly lit. This is because of how the signal lines will “drift” before they are initialized. By the time the init section of code is complete, all the LEDs will be properly initialized to off.

Some registers you write to do not appear to change – any register whose value in the debug window shows “WWWWWWWWW” is a write-only register so the debugger will never show you the value. Writing to these registers impacts other registers. For example, writing to PER\_OER (the Output Enable Register) will cause PER\_OSR (Output Status Register) to change. If you use an older version of IAR, the write-only registers usually show “0x00000000.”

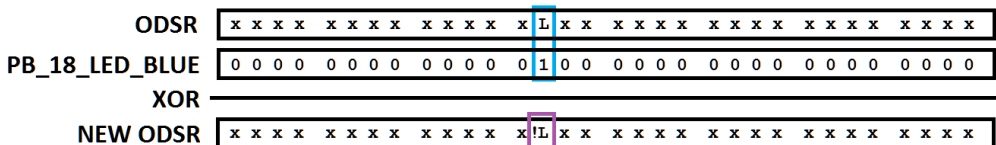
Coding the main loop is next. The first step is to make the LED blink. What is involved in making an LED blink? Even the simplest of tasks should be thought about and designed prior to coding. Draw a flowchart to express what exactly we need to do in code.





**Figure 3-10** Flowchart of what is needed in the code

The initialization is already done, so the `main_loop` code just needs to toggle the LED and wait. To toggle the LED, read `PER_ODSR` (Output Data Status Register) which contains the current state of the LED. To toggle a bit, use an Exclusive OR (XOR) operation which is the “EOR” instruction. We will use the blue LED which is bit 18 and has already been given the symbol definition `PB_18_LED_BLUE`. `PB_18_LED_BLUE` can be called a “mask” since it can be used to work with only the bit that corresponds to the blue LED (and thus “masks off” the other bits). When `PB_18_LED_BLUE` is XORed with `ODSR`, any bit that is 0 in `PB_18_LED_BLUE` will not impact `ODSR` so we can consider those “don’t care” bits. Only bit 18 may be affected. If we call the value in this bit “L” then after the XOR operation, the value in `ODSR` will be toggled to `!L` (not-L, or opposite-L). Graphically, the bits line up like this:




**Figure 3-11 ODSR bits XORed with PB\_18\_LED\_BLUE**

To do this in code, use `r3` to capture the current value of `ODSR` so we get the current state of the LED. Then XOR the `PB_18_LED_BLU` value with `r3` and store the result back into `r3`. This flips bit 18. Then use a store instruction to load `ODSR` with the modified value. Since none of the other bits are touched, we can safely re-write them to `ODSR`.

```
main_loop
update_LED
    LDR        r3, [r2, #PER_ODSR_OFFSET]    ; r3 = *(r2 + PER_ODSR_OFFSET)
                                                ; read the current AT91C_PIOB_ODSR value
    EOR        r3, r3, #PB_18_LED_BLU       ; r3 = r3 XOR the value to toggle LED
    STR        r3, [r2, #PER_ODSR_OFFSET]    ; Write the modified value back to ODSR
```

Even though the syntax of assembly language is difficult at first, the fact that each line of assembly code is such a small, discrete step makes it easier to understand than a line of C code.

If you are ultra-keen (and to avoid negative comments from embedded programming experts), you should ask, “but what happens if something else changes the ODSR register while the processor is working with the saved value in r3?” That will never happen in this example. In the future, it could and this is one of the most infamous sources of firmware bugs in embedded systems called the “read-modify-write” problem.  Because we are working in assembly language, it is obvious that there are three steps involved in modifying a bit in a memory location. The equivalent operation in C is just one line of code, which hides what is really happening because that one line of C code will become three lines of assembler. This is immensely important.

Try building and running the code now. You won’t see any light blinking, although the LED you chose to use might look a little bit dim. Why? Because it is actually turning on and off very quickly since we have not added the “wait” time to give each on or off state some time to persist. To do that, we will use a function called “kill\_x\_cycles” which will literally do nothing except waste millions of processor cycles for us.

The function has already been written and should be in the project. Open kill\_x\_cycles.s and take a quick look. There are two very important things to consider here.

1. What registers do we need to pass values to this function assuming it will be called from C?
2. How many instruction cycles does each instruction take so we can design this function to get as close as possible to killing the specified number of cycles?

Compilers have rules about how they use registers. You can find these rules in the compiler documentation. For IAR, see Help > C/C++ Development Guide and search for “passing parameters” or “Calling assembler routines from C.” Historically, compilers would rely on certain registers for parameter passing, usually R0-R3. A return value is always in R0. The IAR8 C/C++ Development Guide does not specify specific rules for C, though it does for C++. For C, the guide recommends writing a “skeleton function” that has the parameters you want in the assembler function, compile the code, and look at the register assignment.

Since we are not yet calling kill\_x\_cycles from any C code, we use any register we want to. We’ll choose r0 as it is most-likely the register that will be used by our C code when we integrate this function later.

Whenever a function enters and returns a few instruction cycles are used. Since we are trying to make this function as accurate as possible, the number of entry and exit cycles are figured out using the simulator and assigned to a value OVERHEAD. The calling function specifies the number of cycles to be wasted, so this value must be adjusted by OVERHEAD if the requested value is greater or equal than OVERHEAD, to begin with. Watch out for overflows and underflows! A few labels are added to allow branching within the function.

```
#define OVERHEAD 7


kill_x_cycles      ; [1] Function entry
    CMP    r0, #OVERHEAD    ; [1] Check if x is at least OVERHEAD.
    BLT    kill_x_cycles_end ; [1] If x is less than the overhead, exit
    SUB    r0, r0, #OVERHEAD ; [1] Reduce the count by OVERHEAD
```

To consume processor cycles, we set up a loop that will just count down the passed parameter which is still sitting in r0. It takes three instruction cycles for one iteration of the loop, so 3 is subtracted from r0 each time. Once the value underflows, the loop exists and the function returns.

```
kill_x_loop                ; [3 cycle loop]
    SUBS    r0, r0, #3      ; [1] Subtract the loop cycle cost
    BPL     kill_x_loop     ; [2] Check if positive or zero and repeat


kill_x_cycles_end
    MOV     PC, r14         ; [1] Move the return address back to the PC

END
```

 To see the `kill_x_cycles` function which is external to `main_assembler.s`, it must be declared at the start of `main_assembler.s` using the keyword `EXTERN`. Add this just above the `PUBLIC` `main` declaration.


```
; "EXTERN" declarations make labels from other files visible here
    EXTERN kill_x_cycles

; "PUBLIC" declarations make labels from this file visible to others
    PUBLIC main
```

 Add a constant `KILL_CYCLES` after all the other constant definitions for the number of cycles we want to kill. 1 million is a good value to use since by default the development board will be running on its 4MHz internal oscillator, so a million cycles are about 250ms. That will blink our light twice per second.

To illustrate another way to declare a symbol, we use `EQU` instead of `#define`. `EQU` stands for “equate” or assign a number to a symbol.


```
; -----
; Other constants
KILL_CYCLES    EQU    1000000    ; 250ms x (4e6 cycles/sec) = 1e6 cycles
```

 After the `ODSR` value has been updated with the existing `STR` instruction, load the `KILL_CYCLES` value to `r0` and use a “branch and link” instruction to call `kill_x_cycles`. Branch and link does two things: it loads the program counter with the address of the function, and it stores the return address of the next line of code in `r14`. Register `r14` is a special register called the “link register.” The sole purpose of the link register is holding the return address from the most recent function call.

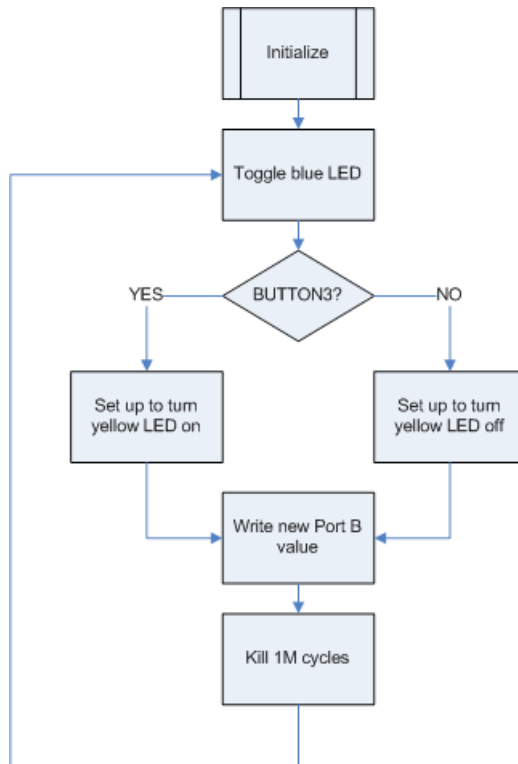
```
STR    r3, [r2, #PER_ODSR_OFFSET]    ; Write the modified value back to ODSR
LDR     r0, =KILL_CYCLES              ; r0 = cycles to kill
BL      kill_x_cycles                 ; Call the function

B       main_loop                    ; Repeat infinitely

END
```

 Once again, build and run the code and this time you should see the LED blinking. Note that single-stepping through the `kill_x_cycles` function would take a long, long time. You can run the code full speed, or you can use `F10` to “step over” the `kill_x_cycles` function call which will still run the code but does so at full speed without showing you what is happening.

The last thing to do is add the check for `BUTTON3` to decide if a second LED should be on. Note that all the buttons on the development board are “active-low” so they will put logic 0 on the processor pin when they are pressed. In this example, we will control the yellow LED. We should update our design document to include the check.



**Figure 3-12** Flowchart updated to show check

The current signal value of BUTTON3 can be read from the PDSR (Present Data Status Register). You can test this in the debugger by running the code, pressing the button and then halting while the button is still pressed. You should see bit 2 change states.

Checking the button is a very simple if-then-else statement. In pseudocode, it would be:

```

if(BUTTON3)
{
    LED on;
}
else
{
    LED off;
}

```

In assembler, you must consider the registers involved and use the correct instructions to work with the processor flag bits properly. Checking the button involves reading PDSR and looking just at the BUTTON3 bit. In one case it will be set, in the other case it will be clear, so we can set up those two cases. Since code runs sequentially, you must jump over the second case if the first case executes, so we add the label “continue” that is the common point where both cases resume. Remember that the BUTTON3 signal is active-low.



```

check_button
    LDR    r4, [r2, #PER_PDSR_OFFSET]    ; r4 = *(r2 + PER_PDSR_OFFSET) read
                                          ; the current AT91C_PIOB_PDSR value
    ANDS   r4, r4, #PB_02_BUTTON3       ; Mask off all bits except BUTTON3
    BNE    button_not_pressed           ; if (BUTTON3)

button_pressed

button_not_pressed

continue

```

Once the state of the button is decided, the associated LED action is taken. The yellow LED is controlled in the same ODSR register as the blue LED, so make sure you do not corrupt the value there. That value is sitting in r3 from when we toggled it for the blue LED. To turn the yellow LED on, just set its corresponding bit while leaving every other bit alone. You can set bits using the logic OR instruction, “ORR.” A good way to test this is by setting a breakpoint on the line so the code halts when you press the button.



```

button_pressed                ; Add a breakpoint after this line
    ORR    r3, r3, #PB_17_LED_YLW    ; r3 |= PB_17_LED_YLW Turn on the
                                          ; yellow LED bit in r3

```



Jump over the next section using a branch instruction, “B”.

```

B        continue            ; Jump around the other case to continue the program

```



Turning off the LED is accomplished by clearing the LED’s bit. You could use a logic AND instruction with the inverse of PB\_17\_LED\_YLW, or you can use the dedicated bit-clear instruction, “BIC.” You do not need to branch afterward since you are back to the common code execution sequentially.

```

button_not_pressed
    BIC    r3, r3, #PB_17_LED_YLW    ; Clear the yellow LED bit

```

Register r3 now has the updated blue and yellow LED states and the code to store this value back to ODSR is already there from blinking the blue LED, so everything is done. Build the code and test!

It should work well with the only complaint perhaps being the response of the LED to the button sometimes being a little slow. This depends on where in the “WAIT” part of our design the button is pressed since the LED will not respond until kill\_x\_cycles has finished executing. This effect would get worse and worse as the rate of blinking slowed down. If this was a product sold to consumers, they would probably complain. A hack would be to check the button and update the LED inside kill\_x\_cycles, but then you make the function entirely application-specific. If you didn’t call the function and just built the delay into the main program, we would not have kill\_x\_cycles available for future use.

This example illustrates very important points in embedded development:

- The processor does exactly what you tell it to do
- Application-specific code cannot be reused
- Making code reusable can have undesired effects.

As systems grow in functionality and complexity, the programmer must be careful to write code that will keep up and consider when effort can be made so that code can be re-used. We generally want our systems to appear as though they are fully multi-tasking

where they can do everything they need to do simultaneously. But one core can only ever do one instruction at a time, so it is up to the programmer to design a system that appears to manage everything it needs to do concurrently, even though that's impossible.

The remaining chapters of this book take you through the design and implementation of a system that appears to be multitasking despite having to serve many different functions. While this could all be written in assembler, it is arguably a lot easier to write it in C and even borrow some object-oriented concepts from C++. As you write in C, keep in mind what you learned about how a processor really works and what the assembly language consequences of every line of C code will be.





## Chapter 4 • Embedded C

C is still, and might always be, the most widely used language for embedded development. Therefore, it is likely the language on which much of your work as an embedded firmware designer will be done. In no way is EiE intended to be a course in C programming. EiE is the application of C programming in the embedded world, with focus on system design, planning, and carefully documented implementation. The design of the EiE firmware system requires a lot of C code, so if you are already a competent C / C++ programmer, much of the work to come is based on the skill set you already have. However, our emphasis is on describing the thoughts and processes that have helped us write a lot of successful embedded software.

If you don't know C-programming, you can likely pick up most of what you need to know as you go along. However, when it comes to designing and writing your own code, a solid knowledge of C syntax, data structures, and object-oriented design will be useful. There are so many textbooks and online resources available for C. Make sure you know where to look when you get stuck. For C-syntax, you need at least proficiency in the following:

- Variable types including enums and using typedefs
- Logical expressions and bit-wise logic operations
- Data structures like Structs and Arrays
- Pointers
- Function calls, if-then-else, loops
- Memory allocation and usage (const, statics, the stack, the heap and dynamic allocation)

There is a “C-primer” module on the EiE website that goes through a simulator-based example program that touches on these base skills. It would be a good place to start if you're new. It will help you get used to the coding conventions used in EiE, and it takes more time to detail the various syntax and structures common to embedded C programming.

In EiE, you will initially be writing low-level code as you construct processor setup and initialization functions, but it will then progress to what you are probably more used to as we write more complex and higher-level driver and API functions. Once you reach a certain level of abstraction from the hardware, the target processor is barely part of the equation.

### 4.1 • Documentation

One of the greatest skills any programmer can have is the ability to clearly and concisely document their source code. Even if you are brand new to writing code, as soon as you start looking at other people's programs you will see very quickly how good they are at communicating what their code does and how it works.

When working in industry on projects that span months if not years of full-time work between yourself and others in your team, you will be amazed at how helpful coding conventions and good documentation can be. Looking back at your old code – or other people's code – can be very confusing if that code is put together without good documentation practices. Being consistent, commenting and documenting, and striving for good design is essential for success.

Documentation does not just refer to code comments but is the complete picture of the design that is created when the code, comments, function declarations, API descriptions,

drawings, flowcharts, state diagrams, written notes, and user guides that are produced for a design come together. Excellent documentation is critical for the maintainability of code.

There is a delicate balance between too much and too little. “Self-documenting code” is a misnomer or at least highly subjective as people tend to believe that what they are thinking is being communicated by the code that they write. Your code should tell a story. A good story has a purpose, a plot and enough details to ensure you don’t get lost. If you have too many details, the story is boring and reads too slow.

If you think of variables as your characters, their names should be recognizable, so you know immediately what the context of the story is about. A function is like a chapter that must be introduced, have a purpose and conclusion. Together, the chapters form a coherent idea and support the overall objective of the story.

As a basic example, consider the following three variable declarations and decide which one is the best:

```
u8 x = 0; /* Counter variable for button presses, max 255 */
u8 ButtonPressCounter = 0; /* Counter variable for button presses, max 255 */
u8 u8ButtonPressCounter = 0;
```

The first line does not give the variable a meaningful name but at least the purpose is described in the comment. Anyone reading the code where operations involve “x” will likely not know what it means without searching back to the declaration every time to read the comment. If another function uses a variable named “x” for something totally different, reading code can become confusing. The second line uses a self-documenting name, so it is clear what the purpose of the variable is although we do not know the type just by inspection when the variable is used. In this case, adding a comment is redundant. The last line communicates the purpose and type clearly and concisely, so in our opinion is the best choice.

This chapter introduces the formal guidelines that we have built up over nearly two decades of writing code. Guidelines are not rigid but exist to offer a model to aim for. They are always evolving. Programmers will agree and disagree on certain styles and conventions when it comes to writing code, but within an organization, it is important to agree to a standard.

Regardless of personal preference, perhaps there is one thing that everyone can agree on when it comes to documentation:

- **Worst:** wrong documentation
- **Bad:** no documentation
- **Not as bad:** poor documentation
- **OK:** good quality, but inconsistent documentation
- **Ideal:** good quality, consistent documentation

Though it might take a few days to set up and document a coding standard, it’s something you do only once but it is useful forever. The earlier you do it, the better. It is unrealistic to expect people to go back and update their work to meet a new standard. New rules should be grandfathered in, or the overwhelming prospect of extra work will kill any buy-in. Once a decision to use a standard is made, it must be adhered to. Ideally, your standard is in place before the first line of code is written, but even if it’s not, it’s never too late to start.

## 4.2 • Doxygen

Structuring your story is one thing, but let's start by introducing an automated tool that may help. There are several documentation tools that you can use to further enhance (or detract) from the overall picture. Doxygen is one used frequently in C and C++, and is self-described as the "...de facto standard tool for generating documentation..." From our experience, the claim is mostly true, not just because it works quite well, but also because, as far as we have seen, there isn't any alternative if your programming language of choice is C. Doxygen is essentially the same as Javadocs if you happen to have used that in Javaland.

Doxygen uses a knowledge of C-syntax and special tags that you add in your comments to parse all your code and put together a summary of what it finds in a nicely formatted HTML document. Each source file is presented with a list of functions and variables for you to view. You can run Doxygen on a project that has not been tagged and it will do its best to understand and map the code.

The usefulness of these output files is debatable. There are two general problems we have found with Doxygen:

1. An excessive number of tags in source code makes directly reading the source code files awkward.
2. Without a defined process for structuring code and comments, Doxygen output can be tremendously confusing and/or useless.

Doxygen output can be useful for API-level firmware, where developers need a good summary of all the functions available. If the documentation is structured well, having this resource can be very helpful. However, a lot of embedded firmware is not used this way, so letting Doxygen parse everything ends up producing a lot of information that is discouragingly overwhelming and debatably functional. If you're writing a single-purpose firmware base that will never be used again, the extra effort to use Doxygen is likely not worth it.

Embedded designers often work directly in the source files with different functions and rely on reading the comments in the code. This is where the Doxygen tags can be irritating if they're all over the place, as they make reading the regular comments difficult. If you are new to using an existing embedded system, understanding the code so you can use it will take much more than a high-level summary of the functions.

While no software can prevent you from writing a wrong description or failing to update some text after code changes, Doxygen goes a long way in helping to keep your documentation up to date. Choosing to use Doxygen is a big – and important – step. If using Doxygen is an afterthought where you cross your fingers and tell your boss about the hundreds of pages of "documentation" your code now has, you might want to reconsider your position. Automatic generation does not magically provide great documentation. You need to intelligently add Doxygen tags to your code and existing comments so that the resulting documentation output is useful and includes the right information.

### 4.2.1 • Documenting the "right information"

The starting point is to make sure that good documentation already exists in your code. It is our belief that the documented source code should be able to stand on its own. Doxygen just makes it easier to reference the entire project conveniently in a separate browser window so you can be writing code in your IDE while referencing the system API in a browser. Since members are hyperlinked to their definitions, it becomes easier to quickly find the information you need to use the API. The "right information" to capture of

course depends on what you want people to reference.

The steps below detail what we thought about the first time we added Doxygen documentation to the EiE firmware source code. We initially spent a day learning the required syntax and how best to integrate that into our existing documentation in a way that we were happy with. We did not want to do a lot of modification to our commenting style and wanted to produce an output that was useful.

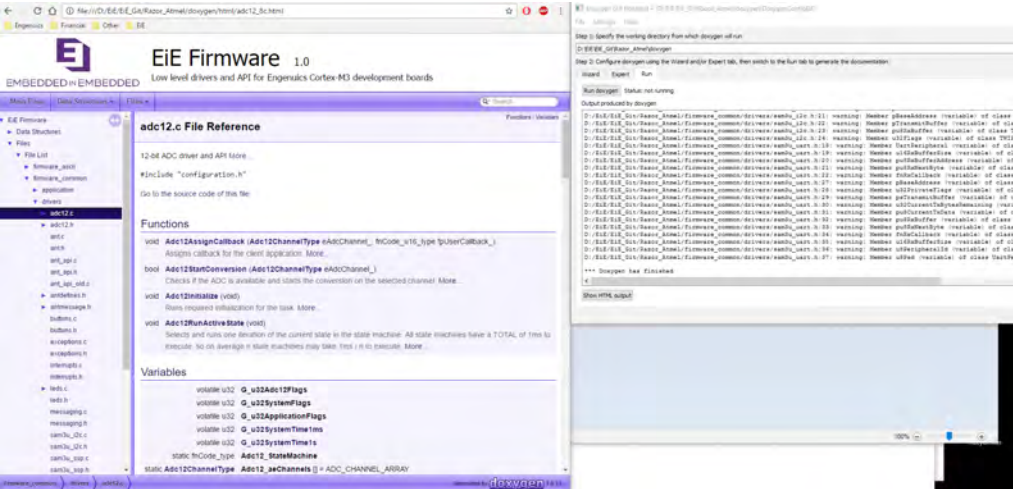


Figure 4-1 Adding Doxygen documentation

We worked with just one of our files and set up Doxygen and a browser, so we could make changes to the source code and then quickly re-generate the HTML and refresh the browser to see the result immediately. Focusing like this and testing let us experiment with a lot of the different tags and options in a controlled way. Not everything did what we wanted, especially when we tried to force a few object-oriented concepts into our documentation. Through many iterations of making incremental changes in the source and header files, we settled on the style and subsequent conventions for our code. In the end, we were quite happy with the result and it was a minimal impact on the look and feel of the original source code.

If you want to try generating the HTML for the EiE system, download the latest version of Doxygen from [doxygen.org](http://doxygen.org) or get it from the archive on the EiE website. Select the "embeddedC" branch in Git and browse to the "doxygen" folder in the repository. There you will find the DoxygenConfigEiE file. We do not archive all the HTML files in every branch, only the final branch in the tree has them (zipped) due to space and maintenance concerns. The HTML folder should automatically be excluded from Git in the gitignore file. If you work through this example, you can generate the files and look at them, and do the same for any subsequent chapter.

#### 4.2.2 • Create a Configuration File

Doxygen's user documentation is quite good but of course subject to one's own interpretations. There aren't many references to the configuration GUI, so we will detail that here.

At the top ("Step 1"), enter the "working directory" which is the reference point that any other path will use. There are a LOT of files generated and updated by Doxygen, so

during development, this would be highly irritating to maintain and confusingly clutter any commits you made. We have seen many releases zip the documentation and just include that with each release. If you follow our model, create a “doxygen” folder to keep your configuration file, but exclude the “HTML” folder from version control. Unfortunately, you cannot specify a relative path here.

We chose to use the “Wizard” settings in the GUI to see the initial results and then worked with the Expert settings to tweak a few things. We found that both tabs simultaneously change the resulting configuration file that is used to generate the output, so be careful you don’t undo a change by switching inputs. The “Project” Topic is almost self-explanatory. All our source code is at the previous directory level “../” and the destination is the working directory.

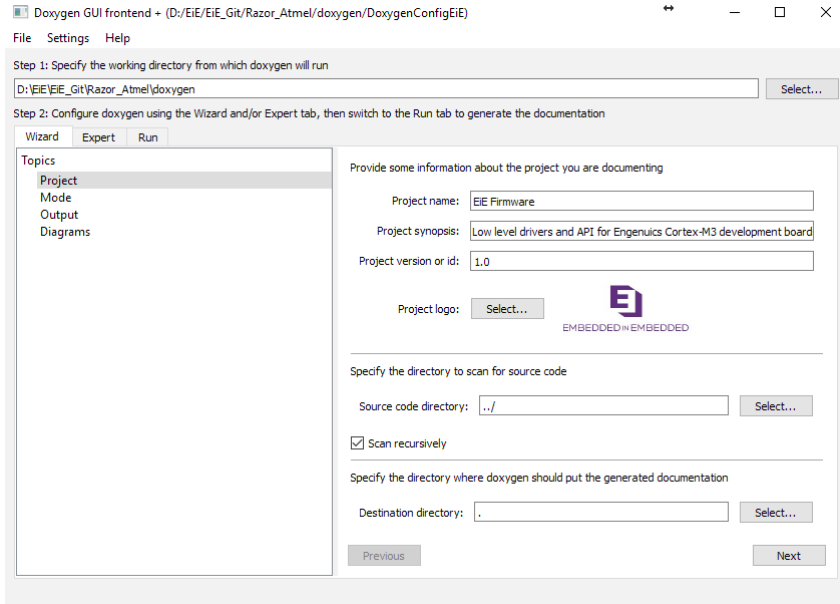
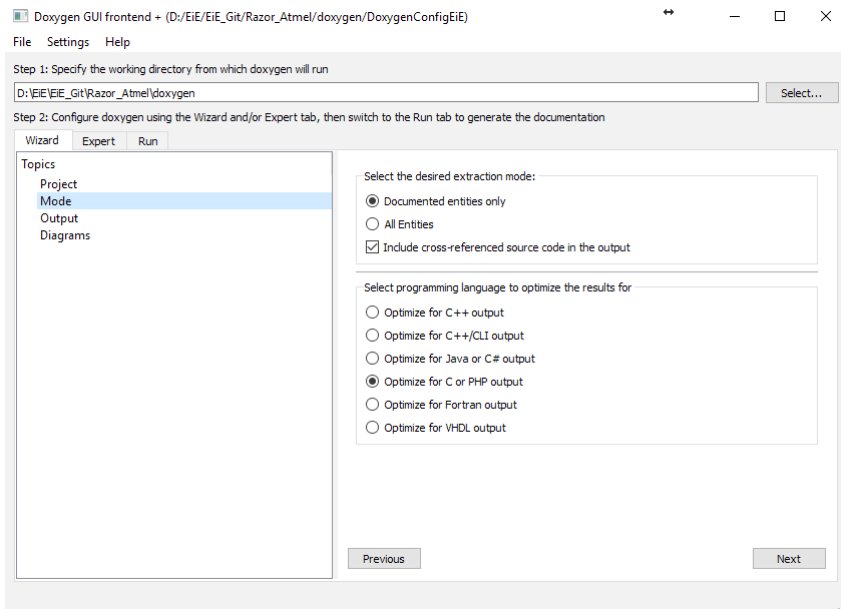


Figure 4-2 Doxygen GUI "Wizard"

The “Mode” section is where you choose what information is pulled out of your code by the Doxygen parser. Try “All Entities” at first to see what you get. It’s a lot, especially on a large code base. Every #define will be added to the documentation package which might not be what you want. Having too much information that you don’t need makes finding the information you do need difficult.



**Figure 4-3 Doxygen GUI "Mode"**

We started by working with just one .c and .h source file pair and added in the documentation tags to extract what we wanted in what we thought was a useful format. "Documented entities only" must be selected, and it is our recommendation to use this Mode all the time. You can play around in the Expert tab to turn on more options about what the parser picks out automatically if you have specific needs.

The "Output" topic is trivial, and "Diagrams" are not applicable to C. All that's left is to click "Run doxygen" on the Run tab and see what happens. Click the "Show HTML output" button to launch the result in your browser. If you are just starting the Doxygen documentation, then you will have tons of warnings about undocumented Members. Once you have figured out your style and updated all your code, then it makes sense to start addressing the remaining warnings.

Save your configuration file. This could be a nice distribution option for Doxygen users who could then generate their own documentation in the way you intended. This would save including the generated documents with the source code, though explaining to someone how to use Doxygen to generate the HTML might be tricky (we have most of this guide on the EIE website if you want to send them the link).

#### 4.2.3 • Special Comment Blocks

To control what Doxygen will parse out of your code you have to set up special comments. The Doxygen parser can read programming syntax to extract many details about the code, but it has to have a way to know what parts of your code comments it should look at. Therefore, a tagging system was designed that offers a fairly unobtrusive way to modify C comment blocks so that Doxygen knows to pay attention to them.

Typically, the Doxygen special comment blocks are described like this:

<code>/*!</code>	OR	<code>/**</code>	OR	<code>///</code>
<code>*</code>		<code>*</code>		<code>///</code>
<code>*/</code>		<code>*/</code>		<code>///</code>

If you read the manual carefully, you see that all that really matters is the starting character sequence. If you diligently comment with legacy C-style `/*` and `*/` comment blocks, then you only have to change the start of a comment block with `/**` or `/*!` instead of just `/*`. This, of course, is not anything special to C since once you are inside `/*` nothing matters until `*/` is reached. If you use `///` then it's quite easy to add an additional `'/'` and it only looks mildly funny.

An important note: if you use Doxygen, don't forget to tell all your new employees who may have never heard of it. This will avoid breaking the heart of a junior engineer who happily reports to you that they removed all the extra `'/'` characters from the source code library for you.

Many people use `/******... "fences" around comments to make them stand out which means those comment blocks are already going to be parsed by Doxygen since they start with /**. However, Doxygen is smart enough to ignore these blocks unless it finds some additional tags which are described in the next section.`

The intermediate `'*'` characters are not required. We find these irritating and time consuming to add if you don't already use them. They make absolutely no difference unless you are standardizing their use to make a distinction between comments you expect to be included in Doxygen output and ones that you do not. We believe there should be a distinction, albeit a subtle one, which is why we have adopted the `/*!` as the marker to use for all Doxygen intended comment blocks.

Two other things to note:

1. Comment blocks will not be parsed if the file in which they live is not flagged to be included. We'll get to that in the next section.
2. We can generally say that comment blocks will be associated with the code that follows the comment. This is described more thoroughly in the Doxygen documentation.

Here is an example:

```
/*!*****
The comments in here are to be parsed by Doxygen because the '!' was added after
the first /*. If Doxygen finds some tags in the body of this comment, it will add
this information into the output document. This comment block will be associated
with the function below.
*/
void function(void)
{...}
```

If you want to follow some code with a comment, you can use `'<'` to indicate that the comment applies to the code that comes before the special comment block. Documenting members in structs or comments on preprocessor definitions are two good examples where this is useful.

```
typedef struct
{
    u32 u32Example,           /*!< Example 4 byte number */
    bool bExample           ///< Example with C++ style comment
} ExampleStructType;
```





associated with the function where the parameter is listed. You can specify [in], [out], or [in,out] which will appear in the documentation. Since we set up our code to document inside our existing Requires / Promises indicators, we found the use of [in] to be redundant.

- **@return <value>** describes the return values of the associated function.

There are also some good tags to help identify or control the parser in ways we were looking for.

- **@publicsection** can group and thus control how functions intended for public API use are documented. This would be automatic for Object-Oriented languages.
- **@protectedsection** can group and thus control how functions intended for protected API use are documented. This would be automatic for Object-Oriented languages.
- **@privatesection** can group and thus control how functions intended for private use are documented. By default, private functions are NOT included in the output. This would be automatic for Object-Oriented languages.
- **@cond <name>** starts a section that will be conditionally parsed. This is the best way we found to exclude anything you do not want to appear in the documentation, like a long list of #defines that are used locally and would otherwise just clutter up the documentation pages.
- **@endcond** pairs to **@cond** to end the section

#### 4.2.5 • Doxygen Example

Here is an example of a shortened source file which will show how we're using the majority of these tags. The @file tag must be included at the start for the file to get parsed by Doxygen.

```

/******
@file adc12.c
@brief 12-bit ADC driver and API

Driver function to give access to the 12-bit ADC on the EiE development boards.
*****/
static fnCode_type Adc12_StateMachine; /*!< The state machine function pointer */
/
/*! @publicsection */

/*!-----
@fn void Adc12AssignCallback(Adc12ChannelType eAdcChannel_, fnCode_u16_type fpUser-
Callback_)
@brief Assigns callback for the client application.

This is how the ADC result for any channel is accessed. The callback function
must have one u16 parameter where the result is passed. Define the function that
will be used for the callback, then assign this during user task initialization.

Different callbacks may be assigned for each channel.

Example:

void UserApp_AdcCallback(u16 u16Result_);

void UserApp1Initialize(void)

```

```
{
    Adc12AssignCallback(ADC12_BLADE_AN0, UserApp_AdcCallback);
}
```

Requires:

@param eAdcChannel\_ is the channel to which the callback will be assigned  
@param fpUserCallback\_ is the function address (name) for the user's callback

Promises:

- Adc12\_fpCallbackCh<eAdcChannel\_> ADC global value loaded with fpUserCallback\_

```
*/
void Adc12AssignCallback(Adc12ChannelType eAdcChannel_, fnCode_u16_type
fpUserCallback_)
{...}
```

When Doxygen processes this file, the top of the HTML output looks like this:



Figure 4-5 Example HTML output

The Adc12AssignCallback function documentation looks like this:

◆ **Adc12AssignCallback()**

```
void Adc12AssignCallback ( Adc12ChannelType eAdcChannel_,
                          fnCode_u16_type   fpUserCallback_
                          )
```

Assigns callback for the client application.

This is how the ADC result for any channel is accessed. The callback function must have one u16 parameter where the result is passed. Define the that will be used for the callback, then assign this during user task initialization.

Different callbacks may be assigned for each channel.

Example:

```
void UserApp_AdcCallback(u16 u16Result_);

void UserApp1Initialize(void) { Adc12AssignCallback(ADC12_BLADE_AN0, UserApp_AdcCallback); }
```

Requires:

**Parameters**

- eAdcChannel\_** is the channel to which the callback will be assigned
- fpUserCallback\_** is the function address (name) for the user's callback

Promises:

- `Adc12_fpCallbackCh<eAdcChannel_>` ADC global value loaded with `fpUserCallback_`

**Figure 4-6 Doxygen output for Adc12AssignCallback function**

The only thing we don't like is that paragraph text is combined into a single paragraph meaning that whitespace disappears if there is no complete empty line between lines in the comment block. You can add '\n' in the comments but then the source code comments start to look crazy even though the Doxygen output is cleaned up. There's probably an easier way to fix this, but at the time of writing, we haven't discovered it.

Once you have decided exactly how you will use Doxygen, capture the process and always be sure all developers reference it when writing code. When a good Doxygen process is combined with well-defined coding conventions, the overall quality of firmware can be greatly improved.

### 4.3 • Coding Conventions

The coding conventions for EiE are a combination of style guidelines, processes, and rules. This complements the more rigorous coding standards defined by the MISRA C:2012 standard. The MISRA C rules shall apply in cases where specific comments are not supplemented here. If you have not read the MISRA standard, get a copy and look through it (it's not expensive).

Experienced programmers will likely know exactly the reason that the majority of the MISRA rules exist because they have probably come across a bug that resulted from the rule being broken.

#### 4.3.1 • Type Definitions

C supports 8-bit, 16-bit, and 32-bit integers both signed and unsigned. It also supports floating-point numbers but in many cases, embedded systems do not, so we will focus on integers. There are quite a few different conventions for type names these days. The following list shows just a few for unsigned integer values that you might come across:

- unsigned char | UCHAR | uint8\_t | u8
- unsigned short | USHORT | uint16\_t | u16
- unsigned long | ULONG | uint32\_t | u32

Type definitions are used to equate these and can be found in typedefs.h. We prefer the u8/u16/u32 style because they are self-documenting and fast to type. They also cooperate well with Hungarian notation. The signed variants are s8, s16 and s32.

#### 4.3.2 • Hungarian Notation

There are many conventions for naming variables. We will use Hungarian notation which means the variable type is included in the variable name. All variables shall be named starting with an abbreviation of their type. Structs start with “st”, enums start with “e” and C strings start with “au8” since they are arrays of u8.

Variable names should be proper English words that fully describe the variable. There is no restriction on length and full words should be used in favor of ambiguous or misleading abbreviations.

Multiple word names are capitalized with no spaces or underscores. All uppercase acronyms should not be used – only the first letter should be capitalized (e.g. Led not LED).

##### Examples:

```
u8 u8Counter;
ExampleStructType stExampleStruct;
u8 au8WelcomeMessage[] = “Hello”;
LedColorType eLedColor
```

Pointers to any variable type shall be named “p<type>” for example “pu8” is a pointer to a u8, “pau8” is a pointer to array of u8, and “ppu8” is a pointer to a pointer to u8. The pointer “splat” (the ‘\*’) should always be on the variable type NOT on the variable name.

Function parameters shall end in an underscore.

##### Examples:

```
u8* pu8TransmitByte
void function(u8 u8Parameter1_, u8* pu8Parameter2_)
```

The only naming exception is for locally scoped index variables like the traditional i, j, and k used for loops.

```
for(u8 i = 0; i < 10; i++)
{
    au8Example[i] = 0;
}
```

If the scope of the index is beyond the local loop, define the index variable at the beginning of the function with all other variable definitions and follow proper naming.

```
void function(void)
{
    u8 u8BufferIndex;
```

```

for(u8BufferIndex = 0; u8BufferIndex < 10; u8BufferIndex++)
{
    au8ExampleBuffer[u8BufferIndex] = 2 * u8BufferIndex;
}
au8ExampleBuffer[u8Bufferindex] = 0;
}

```

#### 4.3.3 • Preprocessor Symbol Definitions

Preprocessor definitions are all uppercase letters beginning with the type and with each word separated by an underscore. TYPE\_UPPER\_CASE\_NAME. They shall be type cast to their declared type and should have an associated comment for their purpose. If they are global in scope, they should have a Doxygen tag.

```

#define U8_EXAMPLE_CONSTANT (u8)20 /* Example constant declaration */

```

Bit locations as pre-processor definitions shall have a leading underscore and then follow the same convention as other constants.

```

#define _U32_EXAMPLE_BIT (u32)0x00008000 /* Example bit declaration */

```

#### 4.3.4 • Braces { }

Braces for functions, loops, and conditional execution shall always begin on the line following the function, loop, or conditional execution line of code. Both the opening and closing braces shall be at the same indent level as the encompassing line of code. All functions shall include a comment following the brace in the form `/* end function() */`. Code within braces that exceeds more than 20 lines, or code where a comment is deemed helpful, should have a closing comment.

```

void ExampleFunction(void)
{
    for(u8 i = 0; i < 10; i++)
    {
        /* Loop code */
    }
} /* end ExampleFunction() */

```

#### 4.3.5 • Switch statements

Switch statements shall use braces to contain all the code within the switch statement. A comment `/* end switch <condition> */` shall be included after the closing brace. Each case statement shall be indented by two spaces and the case statement's code shall be enclosed in braces. Switch statements must have a default case. Any case statement where the code is expected to fall through to the next case shall include the comment `/* fall through to next case */`

```

switch (u8Value)
{
    case 1:
    {
        /* fall through to next case */
    }
    case 2:
    {
        break;
    }
}

```

```

}
default:
{
    break;
}
} /* end switch (u8Value) */

```

#### 4.3.6 • White Space

Sufficient white space shall be used to facilitate code readability and quality appearance. File sections and functions shall be separated with two empty lines.

```

void AntApiRunActiveState(void)
{

    AntApi_StateMachine();

} /* end AntApiRunActiveState */

/*****
State Machine Function Definitions
*****/

/*-----*/
/* Wait for a message to be queued */
static void AntApiSM_Idle(void)

```

Operators and nested parenthesis shall have one space between characters.

```

NVIC_DisableIRQ( (IRQn_Type)(psSspPeripheral_>u8PeripheralId) );

*pu8Result_ = (u8AbsoluteValue / 10) + NUMBER_ASCII_TO_DEC;

```

Where appropriate, additional spaces may be inserted to assist in formatting code to be readable and facilitate recognition of intentional similarities or unintended differences.

```

psNewMessage->u32Token      = Msg_u32Token;
psNewMessage->u32Size       = u32CurrentMessageSize;
psNewMessage->psNextMessage = NULL;

```

#### 4.3.7 • Global Variables

Global variables are necessary for resource-limited systems, so are allowed but must be carefully managed. All source files have three sections near the top of the file where Globals are captured. Each section has rules that govern how the variables are named, their scope, and how they should be commented.

Variables with a full Global scope should be declared volatile. Their name shall start with “G\_<type><taskname><variable\_name>” and should be initialized in the declaration. For arrays or structs where this is not practical, the variable shall be initialized in the task’s Initialization function where the variable is declared.

```

****
Global variable definitions with scope across entire project.
All Global variable names shall start with “G_<type>Example”
****/

```



```
/* New variables */
volatile u32 G_u32ExampleCounter = 0; /* counter in Example task used everywhere */
```

Global variables from other tasks are made visible using the “extern” keyword. They shall not be re-initialized. Their description shall only be the origin file name of the variable to avoid any stale comments accumulating where they are used.

```
/*-----*/
/* Existing variables (defined in other files -- should all contain the “extern”
keyword and indicate the source file from which they originate) */
extern volatile u32 G_u32Example; /* From example.c */
```

Variables that are global only to a task where they are declared are in the third section. These are always static and start with the task’s name, underscore, then regular Hungarian notation. <taskname>\_<type><variable\_name> such as Example\_u32CallCounter.

```
/******
Global variable definitions with scope limited to this local application. Variable
names shall start with “Example_<type>” and be declared as static.
*****/
static u32 Example_u32CallCounter; /* Track # of task calls */
```

#### 4.3.8 • Doxygen Tags

Doxygen tags to define special function blocks shall be /\*! style.

#### 4.3.9 • Function Declarations

Function definitions in .c files shall have the following information and must be formatted for use with Doxygen. The sections are:

- @fn and a copy of the complete declaration code followed by a blank line
- @brief a short description of the main purpose of the function followed by a blank line
- Additional description text if required followed by a blank line.
- Example usage if required followed by a blank line.
- Requires / Promises that describes the pre- and post-conditions for the function
  - Any parameters are specified with @param.
  - To preserve Doxygen formatting, comments in Requires / Promises should appear before the @param descriptions and be prefixed with “- ” (a dash followed by a space).
  - @Returns may be used where return values can be described with a single description.
  - Boolean return values may be described separately as “- ” prefixed comments.
  - There must be a space between the Requires and Promises sections.
  - The comment block ends with a blank line and closing \*/

```

/*!-----
@fn static bool AntQueueExtendedApplicationMessage(AntApplicationMessageType eMes-
sageType_, u8* pu8DataSource_, AntExtendedDataType* psExtData_)

@brief Creates a new ANT message structure and adds it to G_psAntApplicationMsgList.

The Application list is used to communicate message information between the ANT
driver and the ANT_API simplified interface task.

Requires:
- Enough space is available on the heap

@param eMessageType_ specifies the type of message
@param pu8DataSource_ is a pointer to the first element of an array of 8 data bytes
@param psTargetList_ is a pointer to the list pointer that is being updated

Promises:
- A new list item in the target linked list is created and inserted at the end
  of the list.
- Returns TRUE if the entry is added successfully.
- Returns FALSE if the malloc fails or the list is full.

*/

```

Production code should verify any pre-condition as best as possible, or use ASSERT to check parameters and anything else that could possibly be checked in debug mode. For internal purposes, it is permissible to assume that the “Requires” specifications are fully met. Therefore, when using any function, it is imperative that the calling task meets the function’s requirements.

#### 4.3.10 • State Declarations

State definitions in .c files have only the function name and brief description of the main purpose of the state. Promises / Requires are not necessary unless the author feels they are needed. State diagrams should be provided outside of the code.

```

/*!-----
@fn static void AntSM_TransmitMessage(void)

@brief Wait for an ANT message to be transmitted. This state only occurs once the
handshaking transaction has been completed and transmitted to ANT is verified and
underway.

*/

```

#### 4.3.11 • Tabs and Indenting

The development IDE shall be set to insert two spaces when “Tab” is pressed. Code shall be indented with a clear hierarchy which in most cases is automatically performed by the IDE.

#### 4.3.12 • Operator Precedence

The compiler shall not be relied on for operator precedence. Parenthesis shall be used to clearly define the expected order of operations.

```
while( (ANT_SSP_FLAGS & _SSP_RX_COMPLETE) &&
      (Ant_u32RxTimer < ANT_ACTIVITY_TIME_COUNT) )

au8CommandLine[0] = (i / 10) + 0x30;
```

#### 4.4 • C project file overview

Now that you know the rules and style guidelines for writing firmware in EiE, we can begin building our system. Our goal with this book is to share our thinking and thus our experience in designing embedded systems. This is not THE way to design a system, it is A way to design a system that has very particular characteristics and applications. We hope that by sharing this process you can see the depth and breadth of designing and writing a firmware system. You are encouraged to challenge the ideas presented if you do not agree. You are welcome to adopt or refute any style, process, or decision that we make.

You will benefit the most by reading our comments about what we plan to do and trying to write the code yourself. It will be easy to look at the solution, but you will learn much more if you can think through each problem, come up with your own solutions, then compare our method to solve it. From a black-box perspective (i.e. not caring HOW a function is implemented, just that the inputs and outputs are identical), your solution and our solution should be interchangeable. If you look inside that box and compare source code, it may be entirely different. Look for efficiency, robustness, edge case coverage, error handling and anything else that affects the quality of the code. Understand that the decisions made are always influenced by our experience and present knowledge.

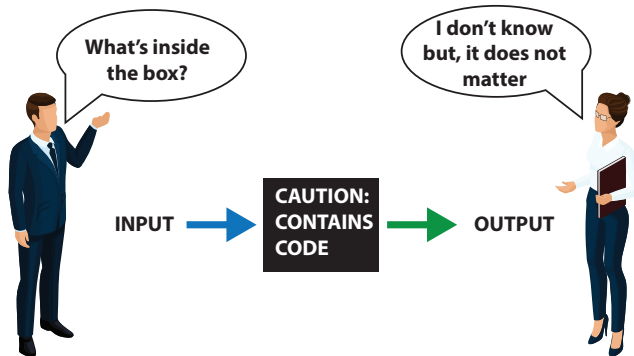
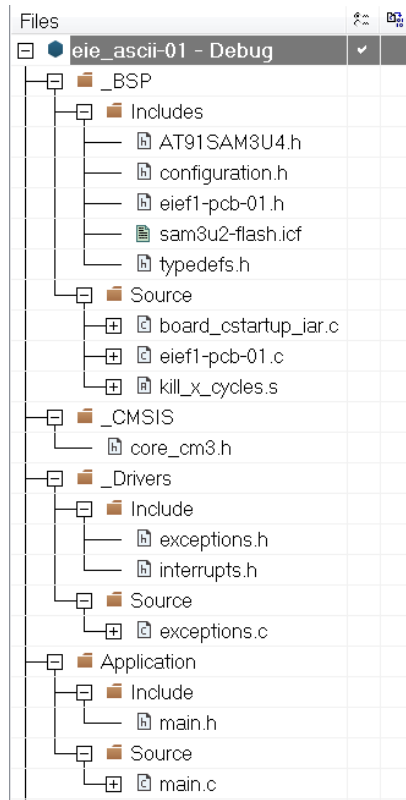


Figure 4-7 Black-box perspective

You already know how to create an assembly-based project in the IAR IDE and the process is identical for projects in C. We start by introducing the files that the remainder of EiE will be based on along with the program structure. These are very close to what you have been working with in the previous chapter with a few more hooks in place to get us going in the right direction. You will also start to see the application of conventions for our code.

Make sure to have the “embeddedC” code for this module downloaded and open in IAR. If you want to get all the code for this chapter, simply load the next chapter’s branch because from here on each chapter’s starting point is the previous chapter’s ending point other than new files that will be added to the project for each chapter.

Expand all the sections in the workspace window and notice that there are 14 files in the project and the program does not even really do anything yet!



**Figure 4-8 Workspace file structure**

The files are grouped to try and make it easy to hide those that you do not need to worry about depending on what you are developing. We like to keep our source and include files all in the project but split up into folders so we can find them. The structure is explained here:

1. **\_BSP** (Board Support Package): This is the collection of files specific to the hardware target you are working with. In most cases, you must write a BSP for any development board or product that you are working on, because of the hardware-specific nature of what these files do.
2. **\_CMSIS** (Cortex Microcontroller Software Interface Standard): header files that meet CMSIS naming conventions. These functions generally provide the low-level register access that must be done in assembler as there is no equivalent C syntax to do them. These are called “intrinsic” functions.
3. **\_Drivers**: All the low-level and mid-level peripheral drivers that provide basic functionality to the development board. These may be somewhat board specific but are intended to have compile-time definitions or options that allow them to be generic for the processor family. Some of these could arguably be in the BSP group.
4. **Application**: high-level functions that should be entirely abstracted from the hardware and run main services or functionality on the target.
5. **Output**: IAR-generated folder for debug files.

Here is a list of files in each folder. These will be part of every project that is done from this point, though some will grow in content as we add more drivers. These files have Doxygen tags added per our conventions.

1. **AT91SAM3U4.h**: processor-specific header file with all the register and bit names from the datasheet for the processor. Use the file specific to your target processor as some definitions may be different between processors in the same family. For the SAM3Ux series, they all reference the same file.
2. **configuration.h**: collection of system-specific configuration settings. Hardware abstraction is difficult to do in embedded and using this central file is part of our answer to that challenging problem. All the header file `#includes` will be here, so every source file should include `configuration.h`. You are welcome to argue that this is bad style since it obfuscates the dependencies of each source file. However, try working with a 100-source file project and adding or changing a header file `#include` to each one.
3. **eief1-pcb-01.h**: target board-specific header file with relevant definitions and function declarations.
4. **sam3u2-flash.icf**: processor linker file that contains all the memory region mappings. This file has been updated from the version you saw in the Assembly chapter to better access and allocate RAM, stack, and heap space.
5. **typedefs.h**: header file with type definitions for the typical variable types used for 8, 16 and 32-bit numbers (signed and unsigned). If the code is ported to a different platform, then the appropriate type definitions can be modified if required to ensure the type sizes are correct.
6. **board\_cstartup\_iar.c**: the start-up file that sets the vector table, does basic initializations and calls `main()`.
7. **eief1-pcb-01.c**: target board-specific source file.
8. **kill\_x\_cycles.s**: assembly file with the killing time function detailed in the Assembler chapter. In a few chapters, this will be replaced by a much better method of killing time.
9. **core\_cm3.h**: header file for Cortex-M3 CMSIS interface. The corresponding `.c` source file is not required to be added to the project as everything we need is inlined within the header file. This file is provided by ARM and Doxygen tagged using different conventions than our own.
10. **exceptions.h**: required declarations for all the exception handlers (more on this later!)
11. **interrupts.h**: required declarations for system interrupts (more on this later!)
12. **exceptions.c**: source file for all the exception handlers (more on this later!)
13. **main.h**: header file for `main`.
14. **main.c**: source file for `main`.

There is not too much code written yet, though you will probably notice some existing definitions in the header files. For example, `eief1-pcb-01.h` has some comments describing setup values specific to the PCB. The details of these values will be described later in this chapter, but for now look to see our method of setting these values up. All the bits in those registers need to be loaded with values to make the corresponding peripheral function correctly. We create initialization values for the entire register at once. The image below shows an example of how the initialization value is documented.

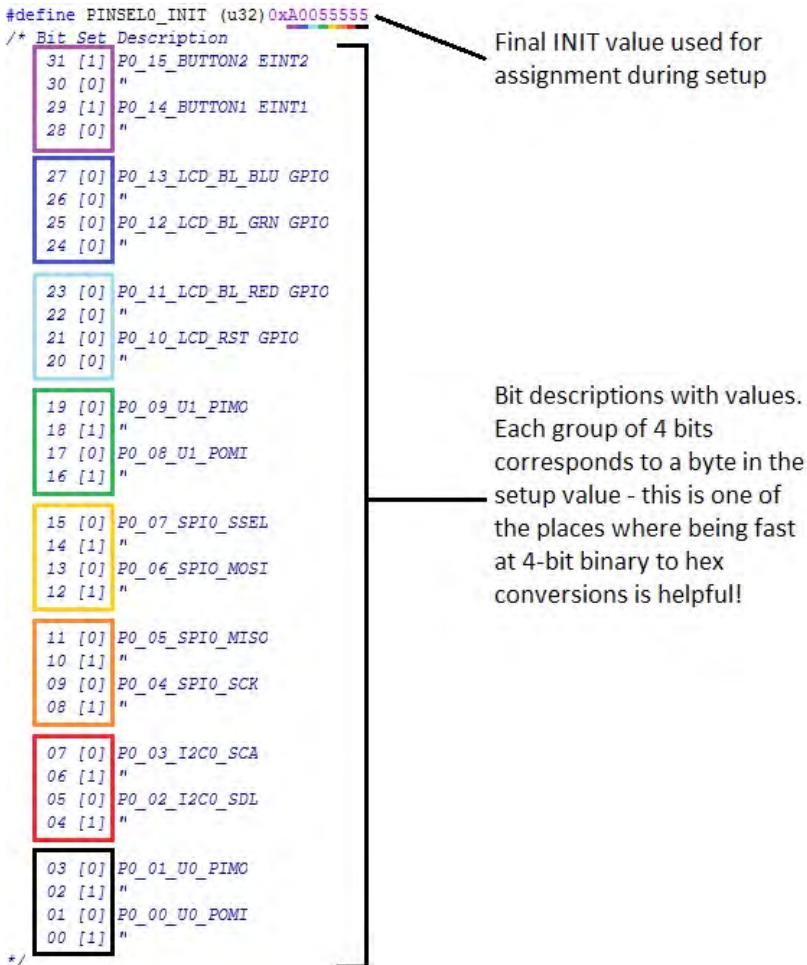


Figure 4-9 Final INIT value and Bit descriptions

Once the full 32-bit value is defined, it can be loaded directly to the register in a single line of code. Other developers may do bit-wise definitions using vendor configuration functions. We find that to be more confusing and it does not force the developer to understand and assign each register bit. Part of working with a new processor is to understand everything about it and solve problems at the beginning. Our configuration values comment each bit so we know its purpose and we explicitly choose its value. There are a lot of registers to set up, but copy and paste helps a great deal. There is even a template definition at the very end of the eief1-pcb-01.h file to help. In the end, it will take several hours to fully set up a file like this which may feel like a lot, but the benefits far outweigh the costs of that time.

#### 4.4.1 • Accessing Registers

There is a standard CMSIS convention for organizing and accessing peripherals on Cortex processors. At first glance, it can be confusing. The key is to learn the process of finding the information you need.

Each peripheral is set up as a big struct of registers in the vendor-provided header file like AT91SAM3U4.h. For this example, we'll use the "PIO" peripheral. Grouping register names in a struct guarantees that the address assignments are sequential, so every register address does not have to be specified and is thus easily portable.

```
typedef struct _AT91S_PIO {
    AT91_REG    PIO_PER;      // PIO Enable Register
    AT91_REG    PIO_PDR;      // PIO Disable Register
    AT91_REG    PIO_PSR;      // PIO Status Register
    AT91_REG    Reserved0[1]; // Spacer
    . . .
    /* about 50 more! */
} AT91S_PIO, *AT91PS_PIO;
```

Each register is of type "AT91\_REG" which is just a u32 type. Giving it a unique type name helps the compiler detect type mismatches when you are making assignments to variables. The first symbol in the struct typedef, PIO\_PER, is a 32-bit number for a register called PIO\_PER. Since it's first, its address offset in the struct is 0.

PIO\_PDR is the second 32-bit member, so its address offset is 4 bytes. Addresses are always byte-wise. PIO\_PSR is at offset 8. The "Reservedx[y]" values you see are spacers because -- for whatever reason we do not know -- the peripheral register is not sequentially next in the address space. The struct definition still must account for the addresses so that the system works.

At the end of the struct is the struct's name provided as a non-pointer and pointer. We will always use the pointer version. The struct for each peripheral will be referenced from a base address of that peripheral. So, what we have is a pointer to a big struct of registers. The arrow operator is used to access a member of a pointer to struct. The base addresses are defined in AT91SAM3U4.h:

```
#define AT91C_BASE_PIOA (AT91_CAST(AT91PS_PIO) 0x400E0C00) // (PIOA) Base Address
```

The value 0x400E0C00 is the starting address of the registers that belong to the PIOA peripheral. This is set by the hardware design of the microcontroller. The #define sets the symbol AT91C\_BASE\_PIOA to this address, and the type cast restricts its usage to the AT91PS\_PIO typedef. The C-syntax to load a value in any peripheral register is:

```
BASE_ADDRESS -> REGISTER_NAME = SOME_NUMBER;
```

For example:

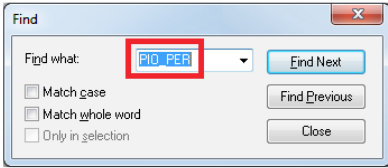
```
AT91C_BASE_PIOA->PIO_PER = PIOA_PER_INIT;
```

The value PIOA\_PER\_INIT would be the constant value defined to configure the bits in the register. A symbol is used so the value can be documented in a header file where the #define is. The register names are taken directly from the user guide, so all you need to remember is the syntax for the code and the process for finding the base address.

1. Get the register name from the user guide, and search for it in AT91SAM3U4.h. This takes you to the type definition of the struct where the register is defined.
2. Go to the name of the struct type at the bottom of its definition and then search for the pointer version of this name to get the #define symbol for the base address of the peripheral you want.

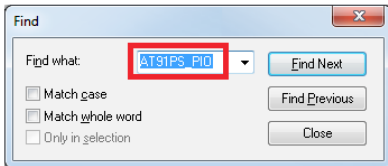


## 1. Search the register name to find the struct to which it belongs



```
1451 typedef struct AT91S_PIO {
1452     AT91_REG PIO_PER; //
1453     AT91_REG PIO_PDR; //
1454     AT91_REG PIO_PSR; //
1455     AT91_REG Reserved0[1];
1456     ...
1459     AT91_REG PIO_KKRR; //
1460 } AT91S_PIO, AT91PS_PIO
```

## 2. Search the struct's pointer name to find the base addresses that reference it



```
6707 #define AT91C_BASE_PIOA (AT91_CAST(AT91PS_PIO)
6708 #define AT91C_BASE_PIOB (AT91_CAST(AT91PS_PIO)
6709 #define AT91C_BASE_PIOC (AT91_CAST(AT91PS_PIO)
```

Figure 4-10 Search for register name and struct's pointer

In this case, there are 3 base address options because there are 3 available PIO peripherals. Each one would have a register struct associated with it. You must know which register you need to work with based on what you are trying to accomplish. For most peripherals, there is just one choice.

This must be understood very clearly because you will do it dozens if not hundreds of times just in this book. Every vendor of every Cortex processor we have worked with follows this model. Making the assignment in C is very simple to type once you get past the syntax. The resulting code that is created by the assembler is essentially the same as what you had written with a few lines of assembly language to do the same thing in the previous chapter.

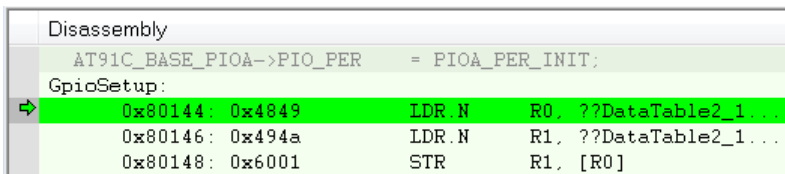


Figure 4-11 Assembly code to initialize PIO\_PER

Remember all that is taking place here:

1. The address of PIO\_PER is loaded into one of the available processor core registers (R0).
2. The literal value PIOA\_PER\_INIT is loaded into a second processor core register (R1) (it may be a value that can be generated, or it may have to come from a literal pool).
3. The value PIOA\_PER\_INIT in one core register is then written to PIO\_PER using the other register as a pointer.

The concept of all this is straightforward. Take your understanding of C syntax, the information you learned from the previous chapter, and the guidance provided here and you should have no problem accessing registers.

### 4.5 • How A Processor Starts Up

An unprogrammed MCU has very limited capability. For the class of processors that small embedded systems use, usually, the only capability that the MCU will have is to select some type of default clock source, reset the program counter to the beginning of flash, and start executing instructions. If no instructions have been programmed, the program counter still parses through the memory but the op-codes it reads are all instructions that do not impact any device operation.

As we saw in the Assembly chapter, `board_startup_iar.c` holds the “vector table” which is a list of addresses that will be selected by hardware events called “exceptions.” The very first address is the “reset vector” which jumps the program counter around the vector table to the C-runtime library-provided code. By the end of the startup file, the processor is directed to `main()` to finish setting up the processor which is where we begin this chapter.

Function calls from `main.c` will be used to configure essential features including the Watchdog timer, the clock we want to use, any power control bits that need to be on, and GPIO. This order is chosen because some Watchdog timers are extremely fast and may not let the other functions run before resetting the processor. Power control and clock come next to ensure the processor is running as expected. GPIO is third because we want to ensure that the circuits connected to the hardware are in a correctly initialized state. On some processors, the GPIO peripheral needs to be configured to define clock crystal input pins. Different processors may require a different startup sequence.

The other thing to remember from the Assembly chapter is that `main()` will have two sections: one with initializations that will run once after reset, and the second with the main program loop that will loop infinitely. Start by setting up those two sections in `main.c`:

```
void main(void)
{
    /* Low level initialization */

    /* Super loop */
    while(1)
    {
        /* end while(1) main super loop */
    }

    /* end main() */
}
```

#### 4.5.1 • Watchdog Timer

A Watchdog timer is simply a counter that resets the processor if it reaches 0. The purpose of the Watchdog is to prevent code from ever getting stuck. The intent is that a program should reload the Watchdog timer at regular intervals so that the Watchdog never times out. Ideally, the Watchdog reload happens exactly one time per iteration of the regular loop code assuming the system timing is deterministic. If you find yourself “sprinkling” Watchdog reloads throughout different functions in your code “just in case” then you should revisit your design strategy.

The SAM3U2 uses the processor’s “slow clock” source for the Watchdog, which is separate from the main oscillator that we will use to clock the system. In this way, the Watchdog can still operate if the main clock dies which is rare but technically possible when using a crystal. It is nearly impossible for an internal RC oscillator to die, so the Watchdog will keep running. On reset, the main clock would restart on the RC and a good system would be able to detect that the crystal was no longer working and issue an alert of some sort. This level of robustness is not added to EiE but would be typical for a

released product.

The SAM3U2 user guide describes all this functionality in section “Watchdog Timer (WDT)”. This shows how to initialize the Watchdog timer using the WDT\_MR (Watchdog Mode Register) as we saw in the Assembly chapter. This time we do want the Watchdog to be active. Since this is a processor-specific function, we will code it in the eie1-pcb-01.c file but call it from main. The initialization value is already written for you in the eie1-pcb-01.h header file.

Looking at our choices for the how the Watchdog timer is configured, we set up WDT\_MR\_INIT for these requirements:

- It should run during idle mode (sleep mode) so if for some reason our alarm clock doesn't go off we'll still wake up
- It should not run during debug mode since we manually halt the processor and can spend a long time examining variables and registers
- The timer should be able to be reset anytime we wish
- The timer is active and will reset the processor
- The timeout period is 5 seconds.



Only one line of code is required to make the setting, but to abstract the hardware and leave room for documentation and future updates, it is written as a function. Don't forget to put the function declaration in the header file.

```
void WatchDogSetup(void)
{
    AT91C_BASE_WDTC->WDTC_WDMR = WDT_MR_INIT;
} /* end WatchDogSetup() */
```

In main(), make this the first call in the “Low level initialization” section.

```
void main(void)
{
    /* Low level initialization */
    WatchDogSetup();
    /* Super loop */
    while(1)
    {
    } /* end while(1) main super loop */
} /* end main() */
```

This code should build and run, but when it does, it is going to reset every 5 seconds because we are not reloading the Watchdog yet. The method of doing this varies between processors. For the SAM3U2, a special bit sequence must be written to the Watchdog Control Register (WDTC\_WDCR) along with a reset bit. The special sequence helps to prevent erroneous writes to this register. There is a finite chance that the code that causes the processor to be stuck also randomly writes memory addresses and could accidentally be resetting the Watchdog timer so it never gets unstuck. There is much less of a chance if the value to reset the Watchdog is unique.



To make things easy for programmers using this system, implement the Watchdog reload as a macro in eie1-pcb-01.h. Find the base address for the Watchdog peripheral by searching WDTC\_WDCR. The value WDT\_CR\_FEED is already in the header file that contains the Key and the reset bit. Dogs like bones, so we'll name the macro

WATCHDOG\_BONE();

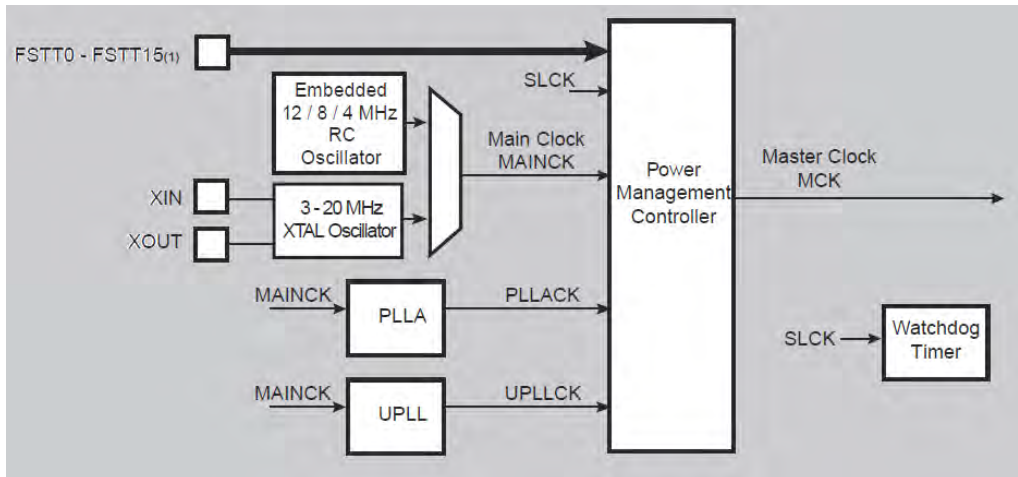
```
#define WATCHDOG_BONE() (AT91C_BASE_WDTC->WDTC_WDCR = WDT_CR_FEED) /* reloads the
Watchdog countdown timer.*/
```

Now feed the dog every time the main loop runs through! Build and run the code to make sure you have typed everything correctly so far.

```
while(1)
{
    WATCHDOG_BONE();
} /* end while(1) main super loop */
```

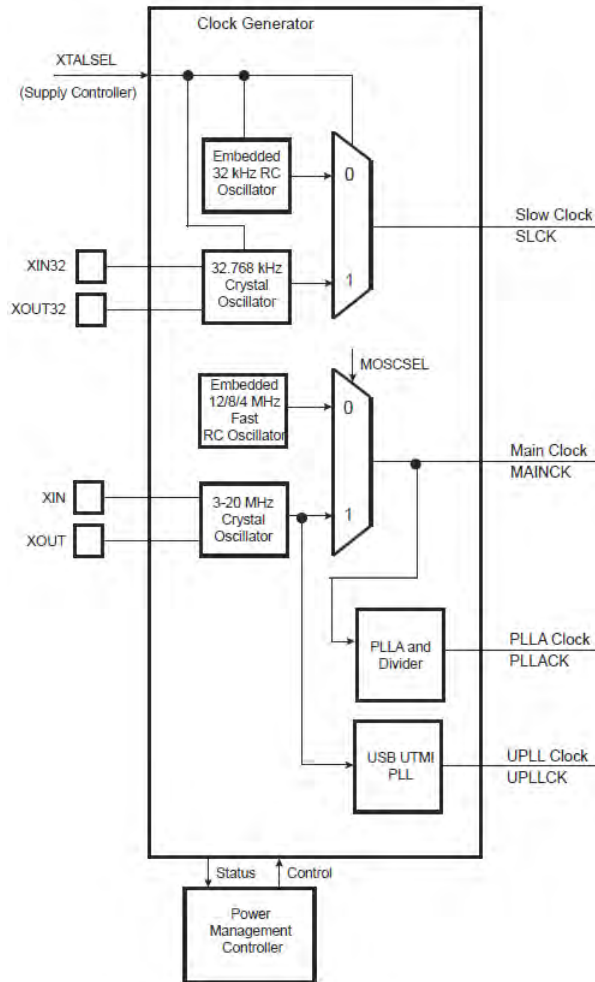
#### 4.5.2 • Clock and Power Initialization

The SAM3U2 has a functional block called the Power Management Controller that is responsible for controlling and providing the status of the system clocks.



**Figure 4-12 Atmel SAM3U2 function block - power management controller**

All the available clocks are described in the “Clock Generator” section of the user guide. A nice summary is provided by the Clock Generator Block Diagram. See Figure 4-13 on page 140.



**Figure 4-13 Atmel clock generator block diagram**

Being able to read diagrams like this is an essential skill for an embedded designer. Let's go through it all to ensure it is fully understood.

1. **XIN32/XOUT32** are pins to which a slow 32.768kHz watch crystal can be attached to provide a precise but low power timing source. Either this or a built-in 32kHz RC oscillator can be selected which gives the Slow Clock SLCK signal
2. **XIN/XOUT** are pins to which a high-speed crystal can be attached. A multiplexer then selects this source or a built-in fast RC oscillator as the Main Clock source, MAINCK
3. **MAINCK** clocks the "PLLA" circuit and "UPLL" circuit. These are special circuits called "phase lock loops" that can up-convert a periodic signal to a faster periodic signal while preserving the stability. For EiE, we will take the 12MHz crystal input and scale it up to 48MHz with the PLLs. The UPLL circuit is dedicated to USB if a separate USB clock is desired.

### 4.5.3 • Implementing the clock setup

The PMC controls how all the clocking signals work, so read the Power Management Controller (PMC) section of the user guide. A complete clock source block diagram is shown, which is repeated here with the section from the Clock Generator we already looked at removed.

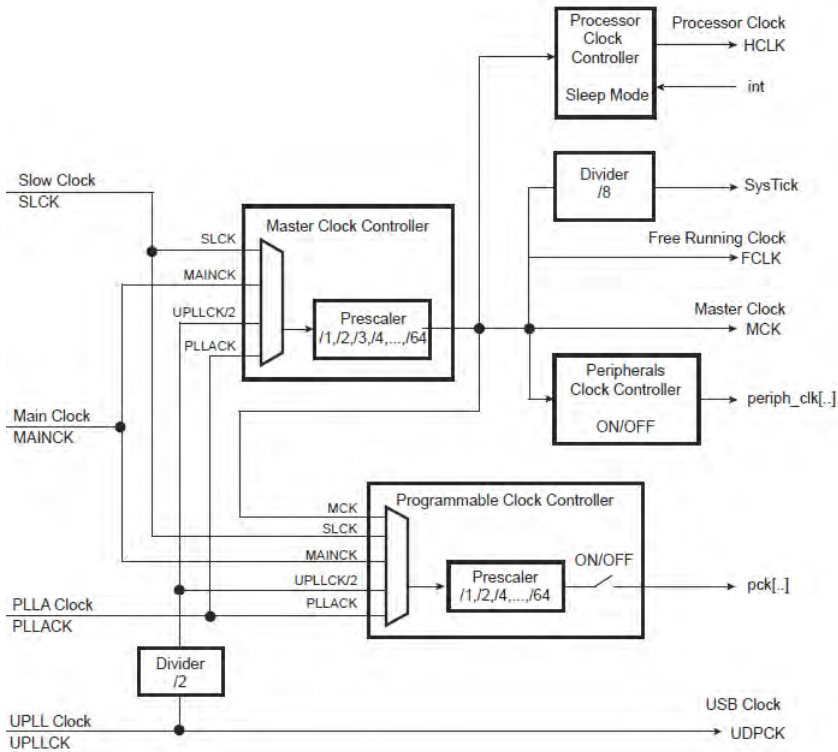


Figure 4-14 SAM3U2 clock control block diagram

There is a lot of clocking information about this processor, which makes this a rude awakening if you just started working with embedded systems. The PMC section does provide a nice walk-through of clock configuration which you would study closely if you were doing this on your own. To ensure we don't get stuck on details, let's focus on the final decisions made for configuring the EiE system. You can reference the following INIT values that are in `eief1-pcb-01.h`:

- `PMC_PCER_INIT` – peripheral clock enable pins
- `PMC_MOR_INIT` – parameters that define the main oscillator (12MHz)
- `PMC_MCKR_INIT` – bits to define the main clock signal
- `PMC_MCKR_PLLA` – second sequence of bits for MCKR
- `PMC_PLAAR_INIT` - PLLA configuration (note the calculations included in the header file above this definition so you know where the values come from)



Before we do any clock configuration, we must prepare the processor for the clocks we are going to set and we might as well turn on power to the peripherals that will be used right away. Create a function called `ClockSetup` which is called immediately after `WatchDogSetup()` in `main`. Don't forget the header file declaration. If the core is going to run at 48MHz, that's too fast for flash memory accesses so "wait states" must be added. You would not know this unless you read the electrical characteristics of the processor. This is done in the `EFC_FMR` register in the `EFC0` peripheral. Peripheral clocks are set in the `PMC_PCER` register using the init value in `eief1-pcb-01.h`.

```
void ClockSetup(void)
{
    /* Set flash wait states to allow 48 MHz system clock (2 wait states required) */
    AT91C_BASE_EFC0->EFC_FMR = AT91C_EFC_FWS_2WS;

    /* Activate the peripheral clocks needed for the system */
    AT91C_BASE_PMC->PMC_PCER = PMC_PCER_INIT;
} /* end ClockSetup */
```

After every power-up, the internal fast RC oscillator is selected because this is always present and must always work even though it may vary in frequency substantially based on the operating environment. If it doesn't work, there is absolutely nothing we can do. We want to select the crystal, give it the proper time to start up, then activate the PLL. Once the PLL is ready, we activate that as our Main Clock. The comments in the code below tell you what is happening. The INIT values are described above and available in the header file. There are a few bit definitions that are taken from the `AT91SAM3U4.h` file.

```
/* Enable the Master clock on the PKC0 clock out pin (PA_27_CLOCK_OUT) */
AT91C_BASE_PMC->PMC_PCKR[0] = AT91C_PMC_CSS_SYS_CLK | AT91C_PMC_PRE_CLK;
AT91C_BASE_PMC->PMC_SCER = AT91C_PMC_PCK0;

/* Turn on the main oscillator and wait for it to start up */
AT91C_BASE_PMC->PMC_MOR = PMC_MOR_INIT;
while ( !(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_MOSCXTS) );

/* Assign main clock as crystal */
AT91C_BASE_PMC->PMC_MOR |= (AT91C_CKGR_MOSCSEL | MOR_KEY);

/* Initialize PLLA and wait for lock */
AT91C_BASE_PMC->PMC_PLLAR = PMC_PLAAR_INIT;
while ( !(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_LOCKA) );

/* Assign the PLLA as the main system clock using the sequence suggested */
AT91C_BASE_PMC->PMC_MCKR = PMC_MCKR_INIT;
while ( !(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_MCKRDY) );
AT91C_BASE_PMC->PMC_MCKR = PMC_MCKR_PLLA;
while ( !(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_MCKRDY) );

/* Initialize UTMI for USB usage */
AT91C_BASE_CKGR->CKGR_UCKR |= (AT91C_CKGR_UPLLCOUNT & (3 << 20)) |
    AT91C_CKGR_UPLLEN;
while ( !(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_LOCKU) );
```

Since various calculations in the rest of the development will use these clock names, it is helpful to do two things:

1. Keep a printout of the clock diagram and label it with the frequencies and divider values that have been set



2. Create some constant definitions with the values for the frequencies and dividers used because they will be used in other parts of the design. Capture these in `eief1-pcb-01.h` since they are board-specific.

```

/*****
* Constants
*****/
#define OSC_VALUE      (u32)12000000 /* Crystal oscillator value */
#define MAINCK         OSC_VALUE      /* Main clock is base crystal frequency */
#define MULA           (u32)7        /* PLL multiplier */
#define DIVA           (u32)1        /* PLL divider value */
#define PLLACK_VALUE   (u32)(MAINCK * (MULA + 1)) / DIVA /* PLL scaled clock */
#define CPU_DIVIDER    (u32)2        /* Divider to get CPU clock */
#define CCLK_VALUE     PLLACK_VALUE / CPU_DIVIDER /* CPU clock 48 MHz */
#define MCK            CCLK_VALUE    /* Alternate name for CPU clock 48 MHz */
#define PERIPHERAL_DIVIDER (u32)1    /* Peripheral clock divider */
#define PCLK_VALUE     CCLK_VALUE / PERIPHERAL_DIVIDER /* Peripheral clock 48 MHz */

```

If you understand the concept behind clock setup, you're doing well. At the bare minimum, you need to know that the main system clock and peripheral clocks are 48MHz. This is a perfect example of why getting an LED to blink correctly as the first thing you do is not such a trivial task.

#### 4.6 • GPIO Initialization

In the next chapter, we will fully set up the processor pin configurations to work with the development board. The hooks will be added here along with a quick setup of just one of the pins we need for the demonstration program for this chapter.

When we set up GPIO using assembler, we went through the process of identifying the button and LED IO lines from the schematic so that we could configure the processor to correctly interface to those pieces of hardware. Buttons had to be set as inputs, LEDs had to be set as outputs. Remember that each IO line is grouped in either Port A or Port B, and specific bits in various registers correspond to the physical pin on the processor.

This same sort of initialization is required in this new program, but right now we are only going to set up a single output line to the "Heartbeat" LED. Find this pin on the schematic, and write a `#define` to identify the pin and its corresponding bit in the GPIO pin names section of `eief1-pcb-01.h`.



Figure 4-15 Heartbeat hardware schematic


```


/*****
!!!!!! GPIO pin names
*****/
/* Hardware Definition for PCB eieF1-PCB-01 */

/* Port A bit positions */

#define PA_31_HEARTBEAT      (u32)0x80000000

```

 Notice the “!!!!” in the title of this section. We use groups of symbols like this as “bookmarks” in the file. As the file grows, each section will have a unique title and a table of contents will be included at the top of the file to help navigate easier.

 Now create a function `GpioSetup` in `eief1-pcb-01.c` to write this value to the `PIO_PER` and `PIO_OER` registers. The function type is `void` and it does not need any parameters.

```
void GpioSetup(void)
{
    /* Set up the pin function registers in port A */
    AT91C_BASE_PIOA->PIO_PER = PA_31_HEARTBEAT;
    AT91C_BASE_PIOA->PIO_OER = PA_31_HEARTBEAT;
} /* end GpioSetup() */
```

Open `main.c` and add the call to `GpioSetup()`. Build the code to check for errors or warnings.

```
/* Low level initialization */
WatchDogSetup();
ClockSetup();
GpioSetup();

/* Super loop */
```

That completes the lowest-level initialization required for the processor. As an engineer, you would test this extensively to ensure that all the clocks were working correctly. Since the clock is output on `PA27`, you can scope this signal to verify the 48MHz `MCLK`.

## 4.7 • Program Structures

When you have an idea for a new embedded design, you must figure out how the program will execute to accomplish the task(s) you wish to do. There are many ways to go about structuring the main program, with three ways presented here:

1. The Infinite Loop
2. Operating Systems
3. State Machine Super Loop

### 4.7.1 • The Infinite Loop

The simplest of embedded systems will run forever inside an infinite while loop. You have already seen this type of program implementation with the programs so far. The `main()` function carries out a sequential set of instructions to do the same task repeatedly. The loops grow in complexity by adding more function calls as the program adds capabilities or tasks, but each function call does the same thing on each iteration. Each task also completes its action without paying too much attention to how long it takes. A basic assumption is that everything happens so quickly, the exact timing does not matter.

For example, a gas detector can be programmed to call functions once per second that read an analog sensor, calculate the gas reading based on the analog-to-digital conversion, update an LCD to show the user the value, log the value to a memory card, decide if the value is high enough to trigger an alarm (and start the alarm function if it is), sleep for a while, then wake up and do it all again. Since the processor can do that all in much less than a second, the instrument can spend most of its time sleeping and

conserving power. One second update intervals are plenty fast enough to keep the user safe and offer a good experience in seeing updated readings and alarm alerts in the event of an unsafe condition.

One second is a lifetime for a microcontroller, and there may be some events that need faster service than just being polled every second. The loop functionality can be advanced by enabling certain interrupts to occur. Interrupts are very important in embedded designs as they allow urgent events to force the processor to stop what it is doing (even if it is sleeping), and run a short piece of code to take care of something urgent.

Using the same gas detector example, the instrument might need to respond to a button press from a user. If the button just polled within the loop, then the 1-second loop time may be too long and the signal missed or feel very laggy like you saw with BUTTON3 in the last chapter. The signal could instead be on an interruptible input so the regular program could quickly jump and respond to the button press.

If the timing of the loop is critical, adding more functions or enabling a lot of interrupt functions could impact the overall loop timing. If the gas detector's main loop normal execution time is exactly one second (including some sleep time), then the main loop timing could be used to blink an LED in an alarm state. The sleep time can vary automatically to handle some variation in function calls that take place, but if the total loop time does not exceed 1 second, then the LED will blink consistently. Look at it like this:

```
while(1)
{
    ToggleLed(); /* Toggle LED: 10us*/
    Function1(); /* Perform reading and analog-to-digital conversion: 300ms */
    Function2(); /* Update LCD: 300ms */
    Sleep();     /* 1s - (ToggleLED time + Function1 time + Function2 time) */
}
```

Since ToggleLed() is the first function call and Sleep() ensures that any time remaining in the loop's 1 second period is consumed, the LED will toggle at exactly 1s. However, if Function2's execution time varied between 300-800ms (depending on what it had to do) the total loop time could potentially be longer than 1 second. In this case, Sleep() would not do anything because it knows that more than 1 second has elapsed, so ToggleLED() would immediately execute but would still be late by 100ms. This would cause the LED to blink at a noticeably different rate at least for that iteration. This could be overcome by reducing the execution time of the functions or using an alternate timing source that could provide a fixed-time signal to toggle the LED (and it would be removed from the main loop).

A designer can specify the maximum time functions are allowed to take and document critical timing requirements for the system. If the system grows within the defined limits then all is well. If the system starts to stretch the limits, then either the system must shrink back down in complexity, become more efficient, or else the system rules must be changed so that it can work in a new form.

#### 4.7.2 • Operating Systems

Jumping to the opposite end of the complexity spectrum, we find operating systems (OS) or real-time operating systems (RTOS). Real-time operating systems are most often used in the embedded world because timing tends to be very important. There are a few varying definitions, but generally speaking, the timing in an RTOS to service an event is fast enough that an event would never be missed (or at least no one would notice). At the very least it will be serviced in a deterministic amount of time. Common embedded

operating systems are FreeRTOS, MicroC/OS, MQX, QNX, and many more!

As device complexity and the amount of firmware services increases, so do the requirements to properly manage resources and applications. Sequential execution will eventually fail if any task starts taking too long to do what it needs to do, or there are so many tasks in the system it is impossible to get through them all in a reasonable or predictable amount of time. At this point, an operating system could be a great solution.

At the highest level, an operating system is a program called a kernel that manages all the tasks in the system. The main component is called a “scheduler” which is responsible for giving each task some processor time to progress. Even if the task isn’t finished what it is doing, the scheduler will take the processor away for other tasks. When it returns to the task that was still working, that task picks up where it left off without ever knowing it stopped. Stopping (and then restarting) a function or task in the middle of what it is doing must be very carefully done and requires a fair amount of overhead. The operating system also manages the various system resources like peripherals and memory.

Deciding if or when your system has reached this point is a critical design decision that should be considered before the first line of code is written. A big factor is whether your embedded system will be single purpose like a gas detector or a GPS tracker, or if it will support multiple applications perhaps contributed to by many different programmers like a smartphone. Implementing an operating system – even a small embedded one – requires relatively large overhead in time and resources. Many aspects of the system are designed very differently, and there will be hundreds of work-hours spent setting up just the operating system before device drivers and any application code can be written or adapted from existing code in the OS.

The principal advantages of using an OS come from sharing resources, managing memory, and allowing task priority to guide how the system responds to events and application needs. To implement that, a lot of code is required, and a lot of processor resources are used. Even the smallest of embedded operating systems need a minimum of 4kB of flash and 2kB of RAM just for the kernel to operate, but that can quickly balloon. Often the choice to go to an OS includes the choice to move away from a microcontroller with built-in flash and RAM and use a microprocessor with external memories that come in much larger sizes and can be scaled as the application grows. If you really want to step up and run an OS like Embedded Linux, Android, Windows Embedded, etc., then you are looking at a whole new level of embedded processor with MB of flash and RAM – these systems resemble desktop computers more than little embedded systems and are way beyond the scope of what we are doing here. Raspberry PI and Beagleboard are great examples of this kind of system.

Another significant consideration for going to an OS is real-time operation. A device that runs in “real time” is fast enough to respond to events, though “fast enough” is relative. It is pretty much guaranteed that an embedded system without an OS can respond faster to an event than one with an OS if it is set up correctly. Instantaneous response is not necessarily essential. For example, if an application is supposed to turn on an LED when a button is pressed, then the system probably needs to be able to respond to events in less than 100ms. The requirement is based on the user’s perception – even if the delay between the button press and the LED activation is the full 100ms, it is fast enough to appear instantaneous to the user. As far as the user is concerned, it is in real time.

For system-level interactions such as communication between peripherals or external communications, the maximum system response time must be faster. If data is coming into the processor at 4 million bits per second for example, then the system must be able to respond quickly enough to the arrival of each bit or byte to grab it and put it somewhere safe before it gets overwritten by the next incoming data. In this case, the latency to respond becomes microseconds which may be unachievable since other operations must take time to operate. The specification could be revised such that a

high-priority interrupt could be used to grab the byte and store it in a buffer. If the data is bursty like an occasional message that comes in, it can be stored and then processed later when the system has time.

Operating Systems become essential when products are multi-purpose, task time is not deterministic, and/or there are multiple, complex applications that require so much processing time and system resources that the resources must be divvied up between all of the apps that need them. For now, unless you are writing an embedded system that has multiple applications running and where the total resources on the processor are exceeded by the combined requirements of all the applications, then you do not need to consider using an RTOS.

That being said, if you start working for a company that already uses an RTOS, or if you just want to play with one, then learning one will be a lot of fun. It is also arguably much easier to write code that will run in an OS because you can focus almost entirely on your own task as if it is the only one running in the system.

Understanding the concepts behind an operating system can enable you to build a more robust system that does not use an OS since many of the concepts essential to making an OS work are useful in infinite loop and state machine-run systems. That happens to be how the EiE firmware system came to be.

#### 4.7.3 • State Machine Super Loop

Implementing a state machine (SM) super loop is a great way to optimize a system and keep it flexible for new functionality. A state machine is a model of a program or task, where the task is broken down into smaller tasks called states. While certain conditions are true the state machine runs code that matches those conditions. This is one state. If some other conditions become true, then a new state is entered, and the code is run that matches those conditions.

Driving a car is a good example. When the car is in the off state, you can press the gas, turn the steering wheel, shift gears, etc. but nothing happens. Insert the key and switch the engine on, and now the gas will rev the engine, but the steering wheel and brakes still do effectively nothing. If you put the transmission in drive, you are now in the driving state where the gas and brakes do work. The only way to return to the idle state is by putting the transmission back in to park. You must remove the key to go back to the off state. You could start the car and be in Idle, and decide to go back to Off if you changed your mind. You can't go from an off vehicle to driving without going through idle (a car won't let you start it if it's not in Park), though you could be driving a car and take the key out. The very high-level state diagram is shown here.

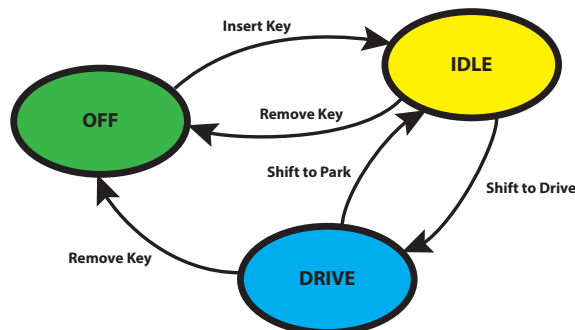


Figure 4-16 Driving high-level state diagram

Breaking down programming problems into tasks and running those tasks as state

machines is an excellent way to design firmware. It inherently helps you manage complex systems since each state is coded independently from other states. Every task in the EiE system is designed and coded as a state machine. Some tasks have only a single state, while others have many states. Whole modules can be written as their own state machine and then included into the system as needed. By combining state machines in an infinite loop structure, you can build an embedded system that very closely resembles the resource-sharing behavior of an OS. This approach includes many object-oriented design techniques and advantages, too.

A single SM is essentially a small application (or “task” or “process” if you prefer) that carries out a particular function. The EiE applications will start as very simple programs to run on the development board. Let us define an application as the following:

1. An independent code set that performs one or more related actions
2. Requires volatile and non-volatile memory resources
3. Requires processor time to execute
4. May require microcontroller peripheral resources like a communications bus (either exclusively or shared)
5. May or may not interact with other applications

An application usually starts with some sort of initialization and then moves to an Idle state where it checks for the trigger(s) that can make it advance to a new state. The Idle state may also do the basic processing required by a task. An interrupt or associated call-back function might be the only signal that will move a state machine out of its Idle state, so it can react to the information that came in with the interrupt. Whatever kicks it out of Idle will be dealt with through a sequence of subsequent states that will eventually end and bring the SM back to Idle.

The task is called on each iteration through the main loop, but the function that executes depends on the current state. This is accomplished by using a function pointer in the main loop which can be changed by the state machine. From the previous chapter, we know that all a function call is is loading the address of the function into the program counter and saving the return address in the Link register (if there are parameters being passed, these are passed in other registers or pushed to the stack). The most complicated part about coding a function pointer in C is figuring out the C syntax. There are a few ways to do this, but one that works in the IAR compiler is shown here. This type definition is in `typedefs.h`.

```
typedef void(*fnCode_type)(void); /* State machine function pointer type */
```

This says that the variable type `fnCode_type` is a pointer (i.e. an address) to a function that returns void and has no arguments. This allows assignments to be made where function addresses are set to a variable of this type to give us our function pointers. All state machine state functions must be of type void with no arguments for this to work - anything else would be impossible since there is just a single call to each state machine. If information must be shared between states or passed to a particular state, then you have to use alternate methods to pass the data like local globals or private member access functions within the state machines.

Function pointers in the EiE system hold the address of the active state for its associated state machine. The super loop does not know what state is currently set, nor does it care. A particular state (like an Idle state), might be the current function for thousands of executions of the main loop. When a state machine advances states, it updates the global function pointer to the address of the function of the state that should run next. When the super loop calls the function pointer on the next iteration, the function pointer has a new

address and thus a different state will execute.

For EiE, we use a function called “RunActiveState()” in the main loop which hides the function pointer notation since it always confuses people at first. The following image illustrates the point where three separate tasks are running in the system. Each has their own unique states and are entirely independent of one another. Task1 is currently in its State 3, Task 2 is in its State 1, and Task 3 is in its State 2.

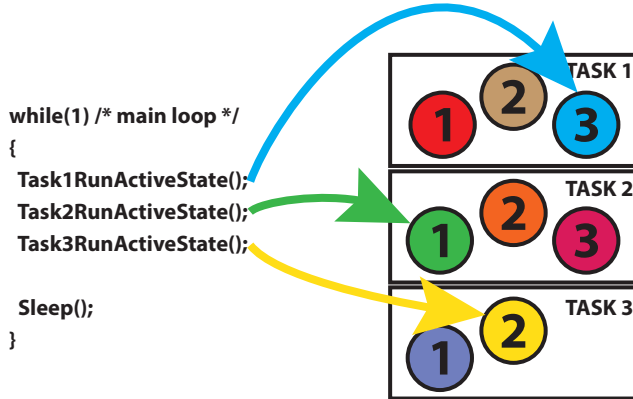


Figure 4-17 RunActiveState() functions

Since every SM is running and each has its own memory footprint, the total system resources must be enough to handle the sum of all the SM requirements. This will still be considerably less than an RTOS-based system where every task has its own stack and heap. Only one common stack and heap are allocated for the SM-based system.

Some SMs may need to share other hardware resources like communication peripherals, timers, or I/O pins. In this case, care must be taken to manage these resources to ensure that conflicts do not occur. This is no different than an operating system.

The most difficult part to manage is processor time. There is no scheduler to tell a task it must pause or stop if it is taking too long. Therefore, any programmer who writes a task in this system must agree to design the state machine in a way that will not block other tasks from running.

The complexity with which this is done can make an SM system start to look very much like an operating system. Semaphores are used to ensure only one SM accesses shared resources, and priorities are assigned to SMs or events in SMs to make sure that critical information is sent or received (sometimes at the expense/loss of non-critical information) as needed. Interrupts and direct-memory access are used extensively for communications so that even large amounts of data flow do not disrupt the main loop. Each task is carefully coded to run quickly.

The neat thing about an SM approach is that it can be very modular and give designers freedoms that start to rival an OS, but with barely any overhead. The rules of the system are dictated by documentation rather than enforced by compilers and added code. Task interaction can borrow ideas from object-oriented design but also use some very low-level techniques to avoid extra code. On the downside, since rules are just described and not enforced, it becomes harder to share development beyond a small group. It is unlikely that an SM-based platform would ever be intended for distribution to other companies or individuals to write applications for.





For the duration of this course, all code will be written as single-purpose state machines designed to run inside a super loop. The design target will be a system loop that always executes in 1ms no matter how many state machines are being serviced. Some SM states will have to take multiple iterations through the main loop to fully execute or wait for events to occur. For example, an analog to digital conversion requiring an average of 100 samples over one second would need a “sampling” state that would take a sample every 10 loop iterations and keep a running total until the 100<sup>th</sup> sample occurred (on the 1000<sup>th</sup> iteration of the loop). Once the 100<sup>th</sup> sample was ready, the calculation could be made quickly and then the process started again.

```
void SampleSM_Sampling()
{
    static u8 u8SampleDelay = 10;
    static u8 u8SampleCounter = 100;

    u8SampleDelay--;
    if(u8SampleDelay == 0)
    {
        u8SampleDelay = 10;
        ReadAdc();
        u8SampleCounter--;
        if(u8SampleCounter == 0)
        {
            u8SampleCounter = 100;
            CalculateAverage();
        }
    }
}
```

When states are not busy doing something, they will be in an idle state and thus process through very quickly which will maximize the sleep state time. The low power sleep state at the end of the super loop will consume any remaining balance of the 1ms loop time by entering sleep mode until either a system tick or another interrupt event wakes up the processor. Therefore, when nothing is being done in the system, the system will spend most of its time sleeping with very low power consumption. If the system is very busy, then it will spend much less time sleeping and overall power consumption will be higher.

#### 4.8 • Implementing the SM Super Loop

The super loop framework is just a formalized version of the init and main loop sections of code we’ve already looked at:

```
main()
{
    /* Initialization */
    /* Low level initialization */
    /* Driver initialization */
    /* Application initialization */

    /* Super Loop */
    while(1)
    {
        /* Application code: sequential calls to all State Machines */

        /* Entry to low power mode */
        /* Wake up with system tick */
    } /* end while (1) Main loop */
} /* end main */
```

To make this system work, we assign rules for each of those two sections. Dictating rules in documentation allows us to save a massive amount of coding to otherwise implement those rules in the software. If we wrote a system that did not rely on written rules but provided the multitasking capabilities of this system, we would essentially be writing an operating system.

#### 4.8.1 • Initialization

Initialization code only runs once when the system is powered on or reset. Every task must have an initialize function that is called here. Regardless of what the task does, its initialization function must follow three rules:

1. It cannot make use of non-initialized system functionality.
2. When the task's Initialize function is finished, the task is either ready to run or assigned a safe error state.
3. Initialize functions may take as long as they want but eventually, they must return.

#### 4.8.2 • State Machine Super loop

Once all tasks are initialized, the system enters the main super loop. The loop continuously executes a sequence of function calls that give every task some processor time. This is really the simplest form of an operating system scheduler: a non-prioritized, non-preemptive, round-robin scheduler.

The fundamental difference between this scheduler and that of an operating system is that the processor time allocated to each task is completely up to the programmer. Therefore, we define only one rule that an application must adhere to:

- The cumulative execution time of all tasks that run in a single iteration of the super loop shall not exceed one system tick period.

The system tick period we chose is 1ms. That means that if there are 10 applications in the finished system, each should do what it needs to do each cycle in less than 100us. A hundred microseconds might not sound like a lot of time, but for a system running at 48MHz each of the ten tasks could use 4,800 processor cycles. With knowledge about the other tasks that are running in the system, tasks can safely extend the number of cycles they use. If a task takes too long, it will potentially affect timing with other tasks that may or may not be detrimental to the system. The system will continue to run as long as a rogue task eventually returns to the main loop to allow the other tasks to run, but there may be noticeable lag or glitches in operation.

Since all tasks in the system are supposed to adhere to this rule, you can time events based on the number of times a particular state is called. For example, if you wanted to blink an LED every second, your idle state could keep a static counter that increments each time the function is called (which is every 1ms with each iteration of the super loop). When the counter reaches 1000, then you know  $1000 \times 1\text{ms} = 1\text{s}$  has passed and you can toggle the LED. A function like this is a great check to visually see if the system timing has been compromised. If the LED blink rate is disrupted, you know that at least one task is running longer than a millisecond on a regular basis. Visually detecting this will not allow you to catch tasks that only periodically violate timing, but at least this is one way to help.

The system will also make extensive use of interrupts for high-priority, non-deterministic operations like sending and receiving data from the various communication buses. In this way, it achieves real-time performance on critical tasks subject only to the user-defined

interrupt priority assignments. Currently, there are no restrictions on what interrupts may be used. A general rule is that interrupt service routines execute as quickly as possible. If they trigger longer actions, those actions should be handled within the associated task's regular states.

We can limit exposure to timing issues by coding applications that are minimally impacted by violations of the 1ms rule should they occur. This choice can be debated infinitely, but it comes down to the complexity of the operating system and simplicity of writing new applications for the environment. Again, an operating system by no means guarantees that users will not break the system and will still have rules or at least guidelines for developers to use to ensure new tasks play along well with others.

What we do not have yet is a tool to count the 1ms periods for us. We will figure out how to take care of that automatically and even sleep the microcontroller while unused cycles tick by as soon as we learn about interrupts. For now, we will use `kill_x_cycles` from the previous chapter to make time go by and get a rough estimate of 1ms periods.

To get ready for this, do the following:



Create a pair of volatile u32 global variables in `main.c` to keep track of the system time. Having a time reference in an embedded system is essential, and it does not have to be “real-time” as in the exact date, hours, minutes, and seconds of the day. A counter that starts at 0 when the system starts up and starts counting milliseconds provides enough timing information for most applications. A 32-bit counter of milliseconds can count almost 50 days. A 32-bit counter of seconds can count over 100 years. Together that is plenty for this system. Use “volatile” here because they will later be incremented by an interrupt. They are global so that any task in the system can reference them.

```

/*****
Global variable definitions with scope across entire project.
All Global variable names shall start with “G_”
*****/
/* New variables */
volatile u32 G_u32SystemTime1ms = 0; /* Global system time incremented every ms,
                                     max 2^32 (~49 days) */
volatile u32 G_u32SystemTime1s = 0; /* Global system time incremented every
                                     second, max 2^32 (~136 years) */
volatile u32 G_u32SystemFlags = 0; /* Global system flags */

```



Add another volatile u32 global variable in `main.c` for “G\_u32System flags” and in the “Constant definitions” section of `main.h` use a `#define` to assign the MSB as the “\_SYSTEM\_SLEEPING” bit. More system flags will be added as the system grows.

```

/*****
* Constant Definitions
*****/
/* G_u32SystemFlags */
#define _SYSTEM_SLEEPING (u32)0x80000000 /* Sleep mode is active */
/* end G_u32SystemFlags */

```



Add a `do/while` loop inside the main loop to handle the system sleep. In this loop call a function `SystemSleep()`. Condition the loop on the `_SYSTEM_SLEEPING` flag.

```

/* Super loop */
while(1)

```

```

{
    WATCHDOG_BONE();

    /* System sleep */
    do
    {
        SystemSleep();
    } while(G_u32SystemFlags & _SYSTEM_SLEEPING);

} /* end while(1) main super loop */

```



Write the function `SystemSleep()` in `eie-pcb-01.c` to set the `_SYSTEM_SLEEPING` flag, then call `kill_x_cycles` with an argument of 48,000. Why 48,000?

Clear `_SYSTEM_SLEEPING`, increment `G_u32SystemTime1ms`, and if it gets to 1000, increment `G_u32SystemTime1s`. Then exit the function. You will need to declare the external `kill_x_cycles` function as well as the three Global variables that were just declared in `main.c`.

At the top of `eie-pcb-01.c`:

```

#include "configuration.h"
extern void kill_x_cycles(u32);
/*****
Global variable definitions with scope across entire project.
All Global variable names shall start with "G_xxBsp"
*****/
/* New variables */

/*-----*/
/* Existing variables from other files -- must contain the "extern" keyword) */
extern volatile u32 G_u32SystemTime1ms;      /*!< @brief From main.c */
extern volatile u32 G_u32SystemTime1s;      /*!< @brief From main.c */
extern volatile u32 G_u32SystemFlags;       /*!< @brief From main.c */

```

In the function section of `eie-pcb-01.c`:

```

void SystemSleep(void)
{
    /* Set the sleep flag (which doesn't do anything yet) */
    G_u32SystemFlags |= _SYSTEM_SLEEPING;

    /* Kill the desired number of instructions */
    kill_x_cycles(48000);

    /* Clear the sleep flag */
    G_u32SystemFlags &= ~_SYSTEM_SLEEPING;

    /* Update Timers */
    G_u32SystemTime1ms++;
    if( (G_u32SystemTime1ms % 1000) == 0)
    {
        G_u32SystemTime1s++;
    }
}

} /* end SystemSleep(void) */

```

Build the code to make sure everything works properly. Step into `kill_x_cycles` to ensure the parameter is passed correctly, and watch the system flags and 1ms counter variables.

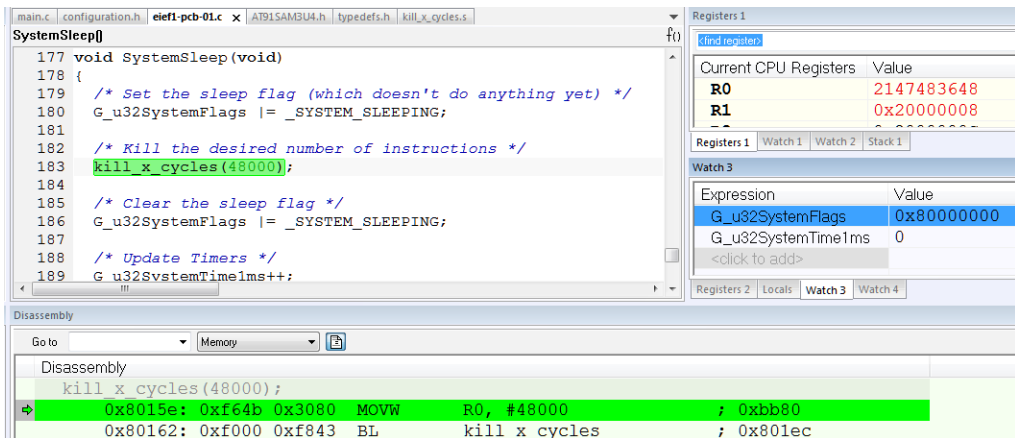



Figure 4-18 kill\_x\_cycles passing parameters


Now the system is ready to run code that generally follows the structure and timing that we want to use.

#### 4.9 • helloworld.c

As we mentioned in the previous chapter, the classic first step for embedded developers working on a new platform is to blink an LED. To many people it is “just a blinking light,” but to an engineer who has spent hours, sometimes days learning, testing, debugging, and trying again and again, that blinking light may represent the very essence of their soul! That might be going a bit far, but it really is exciting and an extremely important step to being successful with microcontrollers.

 So far, we have loaded two registers to give us control of the Heartbeat LED output pin. Now we will use two other registers to turn this LED on and off. The commands will be set up as macros, in the same way the WATCHDOG\_BONE() macro was written. Start by adding the two definitions in eief1-pcb-01.h:

```
#define HEARTBEAT_ON()      ()      /*!< @brief Turns on Heartbeat LED */
#define HEARTBEAT_OFF()    ()      /*!< @brief Turns off Heartbeat LED */
```

 Setting bits that will make corresponding pins high is done with the SODR register. Clearing bits that will make the corresponding pins low is done with CODR. The Heartbeat LED is active-low, so use CODR for HEARTBEAT\_ON and SODR for HEARTBEAT\_OFF. The pin name is already defined earlier as PA\_31\_HEARTBEAT.

```
#define HEARTBEAT_ON()      (AT91C_BASE_PIOA->PIO_CODR = PA_31_HEARTBEAT)
#define HEARTBEAT_OFF()    (AT91C_BASE_PIOA->PIO_SODR = PA_31_HEARTBEAT)
```

It is critically important to understand that the main loop executes once per millisecond. By extension, any code, function call, or state machine call will also execute every millisecond as long as the code that is called runs quickly.

The fastest blink rate that our eye can see is around 15-20 Hz. Let’s make the LED blink 5 times per second, which means we need to change the state of the LED 10 times per second. First, write it in a brute-force way following this flowchart:

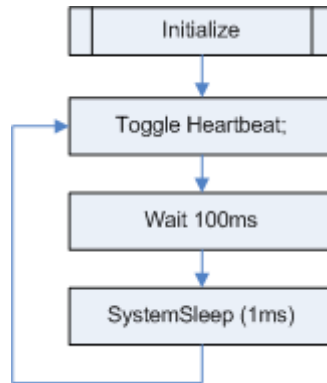


Figure 4-19 Brute-force flowchart



Completely ignore the system timing concept right now and write the code brute force. Don't forget to add the extern declaration at the top of main.c so you can call `kill_x_cycles`. Build and run the code and think through what is happening.

```

void main(void)
{
    bool bLedOn = FALSE;

    /* Low level initialization */
    WatchDogSetup();
    ClockSetup();
    GpioSetup();

    /* Super loop */
    while(1)
    {
        WATCHDOG_BONE();

        /* Toggle the LED */
        if(bLedOn)
        {
            HEARTBEAT_OFF();
            bLedOn = FALSE;
        }
        else
        {
            HEARTBEAT_ON();
            bLedOn = TRUE;
        }

        /* Wait 100ms (4,800,000 instruction cycles) */
        kill_x_cycles(4800000);

        /* System sleep */
        do
        {
            SystemSleep();
        } while(G_u32SystemFlags & _SYSTEM_SLEEPING);

    } /* end while(1) main super loop */
} /* end main() */
  
```

The LED should be blinking as you expect, but the system timing is now totally broken. If you wrote the blinking code in a task, the task would be “blocking” the processor for 100ms every time it was called. This violates the 1ms rule.

Since we know the main loop runs once every ms, change the algorithm to keep track of how many times the main loop has executed and only when it has run 100 times, then run the code that toggles the LED. The flowchart looks like this:

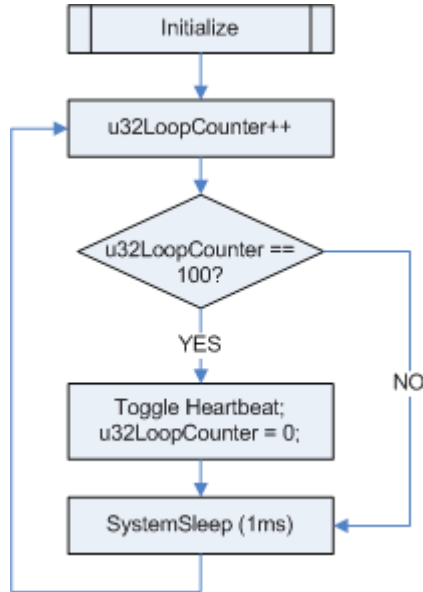


Figure 4-20 Toggling the LED

Implement the functionality above and test the code. The blinking LED should look the same as before, but what is happening is fundamentally different. The processor is no longer blocked and would be able to properly service other tasks in the system at the required 1ms rate.

```

void main(void)
{
    bool bLedOn = FALSE;
    u32 u32LoopCounter = 0;

    /* Low level initialization */
    WatchDogSetup();
    ClockSetup();
    GpioSetup();

    /* Super loop */
    while(1)
    {
        WATCHDOG_BONE();

        /* Increment counter and watch for 100ms */
        u32LoopCounter++;
        if(u32LoopCounter == 100)
        {
            u32LoopCounter = 0;
        }
    }
}
    
```



```

    /* Toggle the LED */
    if(bLedOn)
    {
        HEARTBEAT_OFF();
        bLedOn = FALSE;
    }

    else
    {
        HEARTBEAT_ON();
        bLedOn = TRUE;
    }
}

/* System sleep */
do
{
    SystemSleep();
} while(G_u32SystemFlags & _SYSTEM_SLEEPING);

} /* end while(1) main super loop */

} /* end main() */

```

This might seem obvious, but as you build up a busy system working on one task at a time, it is easy to forget that there are other tasks in the system that need time to run. Perhaps more importantly, you can use the system timing in your tasks to time out events just by using a counter. If you didn't care about some initial mis-timing, you could also use `G_u32SystemTime1ms` and the modulus operator (%) to detect whenever a certain period has passed.

Since this is so important to the system, the Heartbeat LED will be used to indicate the system 1ms timing from this point forward. When the system is awake, the light should be on. When it is asleep, the light should be off. With 1ms timing, the Heartbeat LED change states too quickly for our eyes to see each discrete on and off cycle, but the more time it is on compared to the total time (which is called "duty cycle"), the brighter it will look. This is a basic form of digital to analog conversion called pulse width modulation – you will learn all about that in a few chapters.



Delete the code you have written and instead just wrap the do/while sleep loop in the off and on calls for the Heartbeat LED.

```

/* System sleep */
HEARTBEAT_OFF();
do
{
    SystemSleep();
} while(G_u32SystemFlags & _SYSTEM_SLEEPING);

HEARTBEAT_ON();

```

Build and run the code. If you have an oscilloscope, probe the HB test point and you should see a nice 1kHz frequency signal even though the low time of the signal is very short. This low time represents the main loop running, and the high time represents the sleep time. As you add more tasks to the system, the "on" time will start to increase.

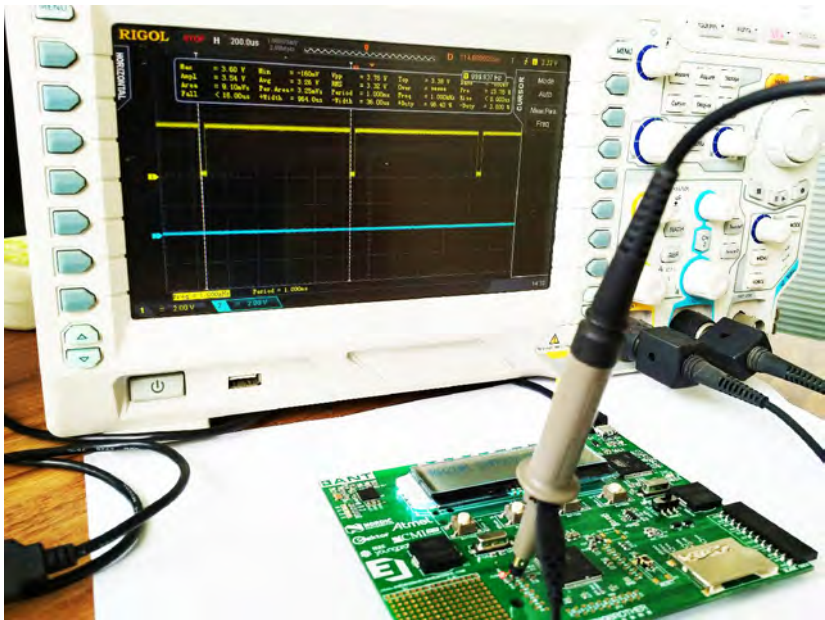


Figure 4-21 1kHz signal on the oscilloscope

Right now, the `SystemSleep()` call always waits 1ms with the assumption that any other code written will be negligible with respect to the overall period. Eventually, that won't be true, so we will later update this to automatically adjust for the time that the tasks take.

#### 4.10 • Next Steps

The subsequent chapters will add functionality to the system. In each chapter, the underlying concepts will be discussed followed by an introduction to the configuration of any required processor peripherals. We'll take you through the coding of the low-level driver and the API that we chose to develop to make accessing the functionality easy for any task in the system. Layering the code like this helps to break down the tasks into manageable pieces. Each layer is designed to abstract the details of the previous layer(s) which increases the chance of portability.

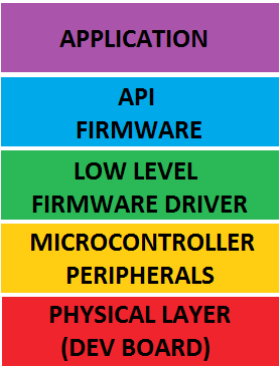


Figure 4-22 Coding layers

The first three layers are difficult to totally abstract in some cases, but by the time you are writing the API, the code should be target-independent. The EiE website has summaries of each chapter, examples, and suggested applications that you can write to use the API developed.

By the time you turn the final page in this book, the firmware system will be able to access all the development board functionality giving you an excellent platform on which to build a myriad of applications. If you follow this entire guide, you will have solid foundational knowledge about the kind of thinking and development decisions that went into creating this system.

Below is a block diagram preview of everything that will be covered. The chapters roughly line up to columns of functionality across the various layers, though there are some interdependencies that are not shown. The SD card and USB Driver and APIs will be left as online exercises.

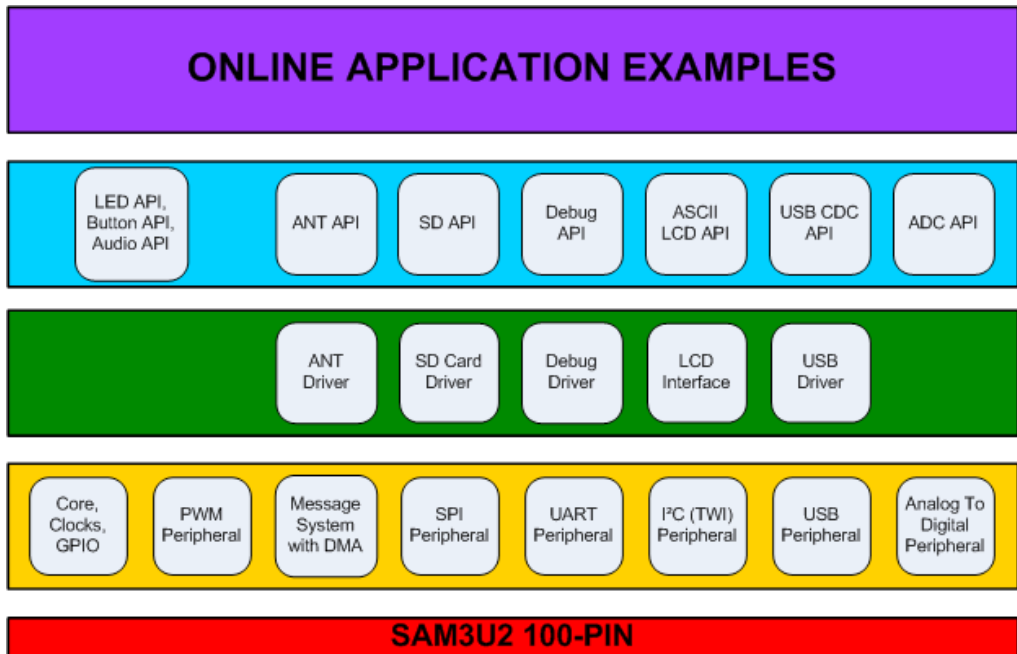


Figure 4-23 EiE System layer block diagram

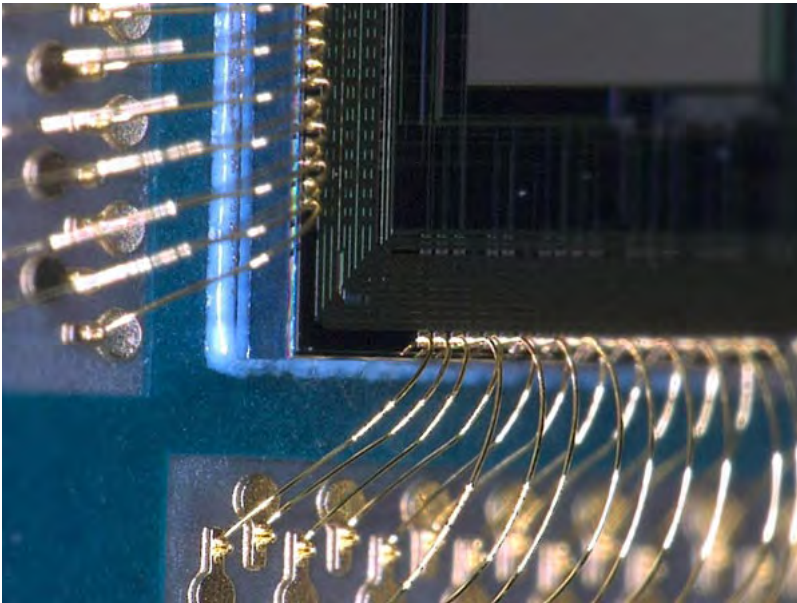


## Chapter 5 • GPIO & LED Driver

### 5.1 • SAM3U2 General Purpose Input Output

Perhaps the most interesting thing about an embedded system is that it is a mechanism to transform software to hardware. Programming on a computer or even writing an app for a smartphone generally outputs only to a screen which, although often entertaining, still has a strong feeling of “virtual” to it. In contrast, turning on an LED, driving a motor, or making sound in hardware that you have constructed is tangible and therefore has a “real” feeling. Being in control of every single bit and knowing exactly why the system is doing what it is doing is also really fun.

Microcontrollers come in all sorts of shapes and sizes, from tiny 8 pin packages all the way up to parts with hundreds of connections. If you look inside a microcontroller, you will see the physical connections between the pins and the chip die. The pin is held in place by the plastic enclosure and is what you solder to attach the MCU to a PCB. Inside the case are very tiny bond wires attached to each pin. These extend to the silicon die where they are bonded to the processor.



**Figure 5-1 Internal bond wires inside a microcontroller package**

Pins can be very broadly grouped as analog or digital. In most cases, the majority of pins will be digital signals. The digital pins can be further broken down to “General Purpose Input Output” (GPIO), and Special Function or Peripheral Pins. As the name implies, GPIO pins could be inputs or outputs. For example, an active-high LED needs a pin output that can be a positive voltage to turn the LED on, or grounded to turn it off. An active-low button connection needs a processor pin input so the processor can read if the button is pressed (logic 0) or not pressed (logic 1). Special Function pins have additional hardware inside the microcontroller to do more advanced tasks like run different communication protocols, track encoders or provide waveforms to control motors.

Embedded designers need to understand the physical and logical relationships that operate in the microcontroller and out to the circuits it controls. Looking out from the processor, the schematic diagram shows all of the different circuits that are connected to the micro. Inside the MCU the signals are connected to logic gates that are controlled by registers. With GPIO, writing a line of code to load a register can translate to logic high and low voltage levels on the hardware pin. Regardless of the particular package, the embedded programmer needs to tell the processor how to handle all those connections.

## 5.2 • External Hardware

When a new design is started, someone has to decide what processor to use and how to assign the pins. In the first chapter, we introduced the block diagram for the EiE development board. From there we can count pins and special peripherals that will be needed, then take that information to choose a processor that has enough physical resources to meet those requirements. Knowledge about how the processor can be configured logically is essential here because in most cases only certain pins are connected to certain peripherals. For example, a processor's datasheet might claim to have 3 serial peripherals and 3 analog connections, but if they use the same pins you are limited to just a few combinations.

A good way to verify the processor is to get the schematic symbol that lists all of the pin functions. You might have to build the symbol yourself. Building the schematic symbol is admittedly tedious but also very critical to complete correctly. If a pin is assigned incorrectly, then the entire design will be flawed along with any other design that makes use of the symbol from the schematic library. It is therefore highly recommended that the symbol is checked and rechecked by at least two pairs of eyes, even if you download it from the vendor. The SAM3U2 schematic that we created is shown here. This symbol doesn't show the power or clock pins as it's too crowded.

90	PB0/PWMH0/A2/WKUP13	PA10/TWCK0/PWML5/WKUP4	39
91	PB1/PWMH1/A3/WKUP14	PA9/TWD0/PWML2/WKUP3	38
92	PB2/PWMH2/A4/WKUP15	PA16/NPC50/NCS1/WKUP5	13
7	PB3/PWMH3/A5/AD12B2	PA15/SPCK/PWMH2	12
8	PB4/TCLK1/A6/AD12B3	PA13/MISO	10
97	PB5/TIO1/A7/AD0	PA14/MOSI	11
98	PB6/TIOB1/D15/AD1	PA12/UTXD/PWMF11	41
99	PB7/RTS0/A0/NBS0/AD2	PA11/URXD/PWMF10	40
100	PB8/CTS0/A1/AD3		
71	PB9/D0/DTR0		
70	PB10/D1/DSR0		
93	PB11/D2/DCD0	PA18/TXD0/PWMF12/WKUP7	17
94	PB12/D3/RI0	PA19/RXD0/NPC33/WKUP8	18
95	PB13/D4/PWMH0		
69	PB14/D5/PWMH1	PA26/TD/TCLK2	25
16	PB15/D6/PWMH2	PB21/A21/NANDALE/RTS2	64
15	PB16/D7/PWMH3	PB23/NWR0/NWE/PCK2	62
68	PB17/NANDOE/PWML0	PB24/NANDRDY/PCK1	58
67	PB18/NANDWE/PWML1	PA22/TXD2/RTS1/AD12B0	5
66	PB19/NRD/PWML2	PA25/SCK2/WKUP12	24
65	PB20/NCS0/PWML3	PB22/A22/NANDCLE/CTS2	63
		PA23/RXD2/CTS1	21
14	PA17/SCK0/AD12BTRG/WKUP6	PA3/MCCK/PCK1	29
84	PA28/TK/PWMH0	PA4/MCDA/PWMH0	30
85	PA29/RK/PWMH1	PA5/MCDA0/PWMH1	31
		PA6/MCDA1/PWMH2	32
		PA7/MCDA2/PWML0	33
96	PA27/RD/PCK0	PA8/MCDA3/PWML1	37
		PA24/SCK1/WKUP11	23
26	PA0/TIOB0/NPC51/WKUP0	PA20/TXD1/PWMH3/WKUP9	19
27	PA1/TIOA0/NPC52/WKUP1	PA21/RXD1/PCK0/WKUP10	20
28	PA2/TCLK0/ADTRG/WKUP2		
86	PA31/RF/TIOB2		
		DFSDP	81
		DFSDM	80
6	PA30/TF/TIOA2/AD12B1	DHSDM	77
		DHSDP	76

Figure 5-2 SAM3U2 schematic symbol

### 5.2.1 • Pin Allocation

Once the base symbol is verified, the next task is assigning pins to the circuits in the design. The block diagram of the device is a good tool to assist in this, and it can be improved at this point by adding more specific information. Since virtually every GPIO pin on a processor has several options available for how it can be used, it is important to investigate each pin before mapping begins to determine if there are any critical functions you need and how using different pins will impact the remaining functionality available on the other pins. For example, Pin 39 on the SAM3U2 can be GPIO, a communications clock signal, a square wave output, or a special wake-up pin. If you need that clock signal, then you don't want to assign that pin as GPIO.

From experience, the best way to do this is to print out a hard copy of the schematic symbol, then systematically go through and circle which functionalities are essential, and which can be crossed off. We find the following order of assignment to be quite successful.

1. **No-choice pins:** power, ground, clocks, etc. can be hooked up first just to get them out of the way. All power and ground pins need to be connected. It is suggested that JTAG pins for programming / debugging fall into this category even though some processors allow you to multiplex GPIOs onto them. This is a highly unadvised practice if you plan to load or debug your device over JTAG.
2. **One-off/special function pins:** any signal/peripheral that you need that is available on only one pin must be assigned first since there are no other options. If two such signals you need are only available on the same pin, then you either have to select a different processor, change your design, or figure out if you can multiplex them somehow (perhaps with an external mux – highly not recommended at this point in the design).
3. **Peripheral-connected pins:** these include communication ports, interrupt pins, analog pins, PWM pins, timer/counter pins, etc. Often there are multiple choices to connect with. For example, you may only need 3 analog inputs, so pick pins that do not have other useful functions on them. Communication ports can also be selected to ensure you get everything you need.
4. **GPIO:** signals that only need to be 0, 1 outputs, or Hi-Z inputs can be allocated next. In general, it does not matter where you put these but you should still be aware of what functions you are giving up if you claim a pin. You must also ensure that the pin will do what you think it will do. Some processor GPIOs will only accept inputs up to the processor supply rail which may be an issue if you run a 3.3V rail but talk to another device with a 5V signal. Some pins might have open-drain drivers instead of push-pull drivers. GPIO assignment could also be influenced by firmware. For example, if you wanted to show an 8-bit counter on LEDs, then the LEDs could be assigned to consecutive pins on the same port if possible. This will help when writing code because the LED driver can write to consecutive bits in a single register to turn the lights on and off. If you randomly assign LEDs to pins all over the place, then it will require a bit more code to map those and update the LEDs when required. Though a lot of times this cannot be avoided, any consideration and optimization will likely lead to a better product delivered in less time. Physical routing of the PCB should be considered if possible, too. Grouping common function pins together that will go to similar locations on the PCB can save a lot of routing headaches. Even selecting the side of the processor where there is a straight-line path to the board location of the target hardware can help routing massively. This can be an iterative process as long as you have the freedom to move GPIO.



5. **Unused pins:** many designs will not need all the processor pins available. In this case, be sure to consult the processor datasheet to ensure you do the right thing with unused pins. You may be able to leave them floating, or you might want to tie them high or low directly or through a resistor. If you have space, it is highly recommended that you stick a test point on them so they are accessible should you decide to add something to the board or make a mistake. If you are really unsure what to do with them, add both a pull-up and pull-down resistor in addition to the test point. Then you have all the options available and the only cost is board space. This can come in handy later for signaling board options or revisions if the pins remain unused. Almost all development boards take the test point approach to maximize flexibility. Even if the test point is physically small and right next to the pin, it is much easier to connect a wire to than the pin itself.

With all that information, the pins and peripherals for the EiE development board are allocated as follows:

1. ANT communication: SSP2 and Port B GPIO.
2. SD card SPI: SSP1 and Port A GPIO
3. SD card high-speed: MC peripheral (and a good option for two pairs of PWM)
4. LCD communication and backlight: I<sup>2</sup>C0 (common bus) and Port A GPIO.
5. UART debug port: UART0.
6. Buzzers: PWMH0, PWMH1
7. Button inputs: GPIO Port A and B.
8. Blade Daughterboard: I<sup>2</sup>C0 (common), SPI, UART and Port B analog.
9. USB: DHS and DFS
10. Remaining GPIO as required and all power, ground, and clocks.

The resulting marked-up schematic symbol is shown. Since this design has relatively few requirements and the processor itself does not allocate too many similar functions per pin, assigning pins was quite simple. It's still a tedious process, and there is still a chance of making a mistake which is probably impossible to fix without a board spin. See Figure 5-3 on page 165.

From here, the schematic symbol can be further updated to group the pins by function or external circuitry so the schematic is well-organized. A pin-mapping spreadsheet can be created to list all of the pin assignments, functions, and configuration information. A table like this is important to interface the hardware and firmware design, especially if you are in an organization where hardware is handled by a different team than firmware. This document shows that everyone is on the same page, and forces both sides to look at all of the configuration options to ensure the selections will work. The EiE development board is fairly simple and both the hardware and firmware were done by the same person, so a pin-mapping table was not produced. Figure 5-4 on page 166 shows a purposefully-illegible drawing of a customer's design using a 144-pin processor.

When configuring GPIO, it is extremely important to understand the system schematics and how to line them up to the firmware. You should have a hard copy of the schematics to reference, but at least have a soft copy readily available. Examine the schematic page with the main processor. See Figure 5-5 on page 167.



ETEF1-PC6-01

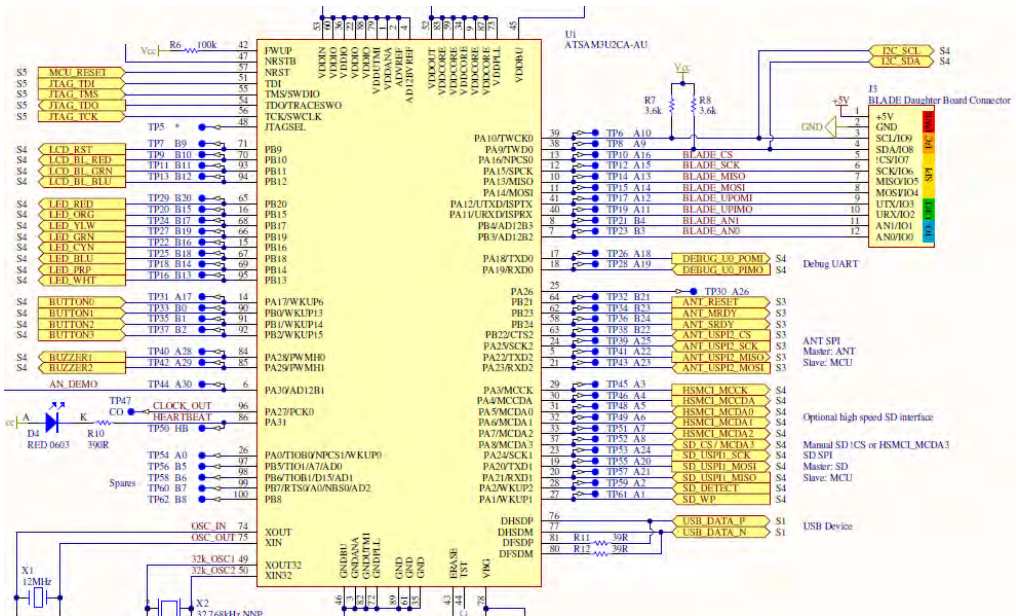
B1	90	PB0/PWMH0/A2/WKUP13	PA10/TWCK0/PWML5/WKUP4	39	BLADE.1°C
B2	91	PB1/PWMH1/A3/WKUP14	PA9/TWDO/PWML2/WKUP3	38	BLADE.2°C
B3	92	PB2/PWMH2/A4/WKUP15	PA16/NPCSD0/NCS1/WKUP5	13	BLADE.SPE
BLADE.AN	7	PB3/PWMH3/A5/AD12B2	PA15/SPCK/PWMH2	12	BLADE.SPE
BLADE.AN	8	PB4/TCLK1/A6/AD12B3	PA13/MISO	10	BLADE.SPE
TP	97	PB5/TIO1/A7/AD0	PA14/MOSI	11	BLADE.SPE
TP	98	PB6/TIOB1/D15/AD1	PA12/UTXD/PWMFI1	41	BLADE.UAH
TP	99	PB7/RTS0/A0/NBS0/AD2	PA11/URXD/PWMFI0	40	BLADE.UAH
TP	100	PB8/CTS0/A1/AD3			
LCD.AST	71	PB9/D0/DTR0			
LCD.BL.R	70	PB10/D1/DSR0			
LCD.BL.G	93	PB11/D2/DCD0	PA18/TXD0/PWMH2/WKUP7	17	DEBUL.UAH
LCD.BL.B	94	PB12/D3/RD0	PA19/RXD0/NPCS3/WKUP8	18	DEBUL.UAH
WAT	95	PB13/D4/PWMH0			
PEP	69	PB14/D5/PWMH1	PA26/TD/TCLK2	25	TP
OR.G	16	PB15/D6/PWMH2	PB21/A21/NANDALE/RPS2	64	ANT.PST
CAL	15	PB16/D7/PWMH3	PB23/NWR0/NWE/PCK2	62	ANT.MROT
YILW	68	PB17/NANDOE/PWML0	PB24/NANDRDY/PCK1	58	ANT.SROT
BLU	67	PB18/NANDWE/PWML1	PA22/TXD2/RPS1/AD12B0	5	ANT.MTROT
GRN	66	PB19/NRP/PWML2	PA25/CK2/WKUP12	24	ANT.SCK2
RED	65	PB20/NCS0/PWML3	PB22/A22/NANDLE/CTS2	63	ANT.CS2
			PA23/RXD2/CTS1	21	ANT.MCS2
B0	14	PA17/SCR0/AD12B4/WKUP6	PA3/MCK/PCK1	29	OPT.SWMT
BUT1	84	PA28/TX/PWMH0	PA4/MCCDA/PWMH0	30	HSUM
BUT2	85	PA29/RX/PWMH1	PA5/MCCDA0/PWMH1	31	SD
			PA6/MCCDA1/PWMH2	32	OE
CLOCK	96	PA27/RD/PCK0	PA7/MCCDA2/PWML0	33	PWM
			PA8/MCCDA3/PWML1	37	DIRTY
			PA24/CK2/WKUP11	23	
TP	26	PA0/TIOB0/NPCS1/WKUP0	PA20/TXD1/PWMH3/WKUP9	19	SD-SCLK
SD-WP	27	PA1/TIOA0/NPCS2/WKUP1	PA21/RXD1/PCK0/WKUP10	20	SD-MISO1
SD-DENFT	28	PA2/TCLK0/ADPRG/WKUP2			
H/B	86	PA31/RX/TIOB2			
			DFSDP	81	
TRIM POT	6	PA30/TX/TIOA2/AD12B1	DFSDM	80	
			DHSDM	77	LSB
			DHSDP	76	DATA

Figure 5-3 Resulting mark-up of schematic symbols

The processor's pins are grouped by "ports" and linked to registers where their functionality is controlled. On the SAM3U2, the I/O pins belong to either port A or port B and have a number such as PA10. This means pin 10 of port A. Note that numbering starts at 0 just like digital bit locations, so PA10 is the 11th pin on its port. The SAM3U2 has a pin corresponding to almost every bit in the two ports, though there are a few unused locations.

Since there are two ports, there are two PIO peripherals, PIOA and PIOB. Each of the two peripherals has a copy of a bunch of registers used to make the pins work. These are described in the "Parallel Input/Output Controller" section of the user guide. The bits in the registers correspond directly to the physical pins. This is the most important concept to understand here! For example, there is a port A register for reading the logic state of the pin. The 11th bit in this port A register corresponds to PA10, so if the value in the register is 0x00000400, then you know that the signal on pin PA10 is logic high and all the other pins on that port are at logic 0.

• 166



**Figure 5-5 Schematic of main processor**

All of the pin assignments to circuits are board-specific. As you saw in the previous two chapters, the logical and functional names of each pin are included in the header file definition. The schematic symbol names should match the logical signal names in firmware so they are easy to reference back and forth. We like to prefix the port and pin number which is helpful when coding.

Now is a good time to make those definitions. Open the `gpio_led` firmware branch and go to the “GPIO pin names” section of `eief1-pcb.01.h`. If you want the true experience of designing a system from scratch, add all of the Port A and Port B pin definitions now by reading them from the schematic. There are 64 of them. If you want to cheat, go to the next chapter branch and copy and paste. See Figure 5-6 on page 168.

A common term for these definitions is “bit masks” since they mask off only the bit of interest. A bit mask is used to isolate the bit and do bit-wise logical operations. For example, if you wanted to set a register bit corresponding to the RED LED, you could logic-OR the `PB_20_LED_RED` constant with the appropriate register. To clear the same bit, you can logic-AND the inverse of the `PB_20_LED_RED` constant. The syntax in C and assembler to inverse bits is the `~` (tilde) operator.

```
PB_20_LED_RED = 0x0010 0000
~PB_20_LED_RED = 0xFFEF FFFF
```



The relationship between bits in registers and the physical pins in hardware is massively important. Do not move past this point if you do not understand this yet.

```

/* Port A bit positions */
#define PA_31_HEARTBEAT      (u32)0x80000000
#define PA_30_AN_DEMO       (u32)0x40000000
#define PA_29_BUZZER2       (u32)0x20000000
#define PA_28_BUZZER1       (u32)0x10000000
#define PA_27_CLOCK_OUT     (u32)0x08000000
#define PA_26_ANT_PWR_EN    (u32)0x04000000
#define PA_25_ANT_USPI2_SCK (u32)0x02000000
#define PA_24_SD_USPI1_SCK  (u32)0x01000000
#define PA_23_ANT_USPI2_MOSI (u32)0x00800000
#define PA_22_ANT_USPI2_MISO (u32)0x00400000
#define PA_21_SD_USPI1_MISO (u32)0x00200000
#define PA_20_SD_USPI1_MOSI (u32)0x00100000
#define PA_19_DEBUG_U0_PIMO (u32)0x00080000
#define PA_18_DEBUG_U0_POMI (u32)0x00040000
#define PA_17_BUTTON0       (u32)0x00020000
#define PA_16_BLADE_CS       (u32)0x00010000
#define PA_15_BLADE_SCK      (u32)0x00008000
#define PA_14_BLADE_MOSI     (u32)0x00004000
#define PA_13_BLADE_MISO     (u32)0x00002000
#define PA_12_BLADE_UPOMI    (u32)0x00001000
#define PA_11_BLADE_UPIMO    (u32)0x00000800
#define PA_10_I2C_SCL        (u32)0x00000400
#define PA_09_I2C_SDA        (u32)0x00000200
#define PA_08_SD_CS_MCDA3    (u32)0x00000100
#define PA_07_HSMCI_MCDA2    (u32)0x00000080
#define PA_06_HSMCI_MCDA1    (u32)0x00000040
#define PA_05_HSMCI_MCDA0    (u32)0x00000020
#define PA_04_HSMCI_MCCDA    (u32)0x00000010
#define PA_03_HSMCI_MCKCK    (u32)0x00000008
#define PA_02_SD_DETECT      (u32)0x00000004
#define PA_01_SD_WP          (u32)0x00000002
#define PA_00_TP54           (u32)0x00000001

/* Port B bit positions */
#define PB_31_              (u32)0x80000000
#define PB_30_              (u32)0x40000000
#define PB_29_              (u32)0x20000000
#define PB_28_              (u32)0x10000000
#define PB_27_              (u32)0x08000000
#define PB_26_              (u32)0x04000000
#define PB_25_              (u32)0x02000000
#define PB_24_ANT_SRDY      (u32)0x01000000
#define PB_23_ANT_MRDIY     (u32)0x00800000
#define PB_22_ANT_USPI2_CS  (u32)0x00400000
#define PB_21_ANT_RESET     (u32)0x00200000
#define PB_20_LED_RED       (u32)0x00100000
#define PB_19_LED_GRN       (u32)0x00080000
#define PB_18_LED_BLU       (u32)0x00040000
#define PB_17_LED_YLW       (u32)0x00020000
#define PB_16_LED_CYN       (u32)0x00010000
#define PB_15_LED_ORG       (u32)0x00008000
#define PB_14_LED_PRP       (u32)0x00004000
#define PB_13_LED_WHT       (u32)0x00002000
#define PB_12_LCD_BL_BLU    (u32)0x00001000
#define PB_11_LCD_BL_GRN    (u32)0x00000800
#define PB_10_LCD_BL_RED    (u32)0x00000400
#define PB_09_LCD_RST       (u32)0x00000200
#define PB_08_TP62          (u32)0x00000100
#define PB_07_TP60          (u32)0x00000080
#define PB_06_TP58          (u32)0x00000040
#define PB_05_TP56          (u32)0x00000020
#define PB_04_BLADE_AN1     (u32)0x00000010
#define PB_03_BLADE_AN0     (u32)0x00000008
#define PB_02_BUTTON3       (u32)0x00000004
#define PB_01_BUTTON2       (u32)0x00000002
#define PB_00_BUTTON1       (u32)0x00000001

```

Figure 5-6 Port A and Port B pin definitions

### 5.3 • The PIO Peripheral

Now we can look at the logical side of GPIO. For the SAM3U2 processors, the GPIO peripheral is simply called the PIO controller. The steps to learn any peripheral in any microcontroller are very similar. If you get good at the process, then the details become the only thing to figure out regardless of the peripheral or the processor. The following process is recommended when learning a new peripheral:

1. **Read the whole datasheet/user manual section of the peripheral you want to use.** Make notes on anything that stands out such as register or bit names, special conditions, tips, and tricks, or additional information you might need. The notes you need to make become more obvious as you do this more often and start to know what you're looking for. When you are first learning, printing the user guide section and using a highlighter is helpful.
2. **Examine the block diagrams.** See if they make sense and contain information that you might need. How do they compare to peripherals you have used before?
3. **Study the registers.** Determine which registers will be necessary to accomplish what you need the peripheral to do. In many cases, you will not need to use all of them, but you need to know for sure if you can ignore something.
4. **Define the configuration values needed for the registers.** Pay attention to special bits that enable power or clock signals to the peripheral.
5. **Add the peripheral configuration firmware and thoroughly test.** Make sure that the peripheral is being configured correctly using the debugger to inspect each register that you modify. This may take some time, and if it is not working it may take hours of debugging looking for the correct bits to make the

peripheral do what you want. Again, watch out for power control or clock bits that need to be active before the peripheral will work.

```

469 #define PIOA_PER_INIT (u32)0x84030007
470 /*
471 31 [1] PA_31_HEARTBEAT PIO control enabled
472 30 [0] PA_30_AN_DEMO PIO control not enabled
473 29 [0] PA_29_BUZZER2 PIO control not enabled
474 28 [0] PA_28_BUZZER1 PIO control not enabled
475
476 27 [0] PA_27_CLOCK_OUT PIO control not enabled
477 26 [1] PA_26_ANT_EWR_EN PIO control enabled
478 25 [0] PA_25_ANT_USPI2_SCK PIO control not enabled
479 24 [0] PA_24_SD_USPI1_SCK PIO control not enabled
480
481 23 [0] PA_23_ANT_USPI2_MOSI PIO control not enabled
482 22 [0] PA_22_ANT_USPI2_MISO PIO control not enabled
483 21 [0] PA_21_SD_USPI1_MISO PIO control not enabled
484 20 [0] PA_20_SD_USPI1_MOSI PIO control not enabled
485
486 19 [0] PA_19_DEBUG_U0_PIMO PIO control not enabled
487 18 [0] PA_18_DEBUG_U0_POMI PIO control not enabled
488 17 [1] PA_17_BUTTON0 PIO control enabled
489 16 [1] PA_16_BLADE_CS PIO control enabled

```

Figure 5-7 PIOA\_PER bit initialization values

Spend the time to really understand the peripheral and be very confident it has been configured correctly. This time is a great investment. Front-loading development effort can yield substantial future time savings since problems identified early are much easier and faster to fix. Building your code on a weak foundation of knowledge is like building a house on quicksand.

#### 5.4 • PIO Internal Hardware

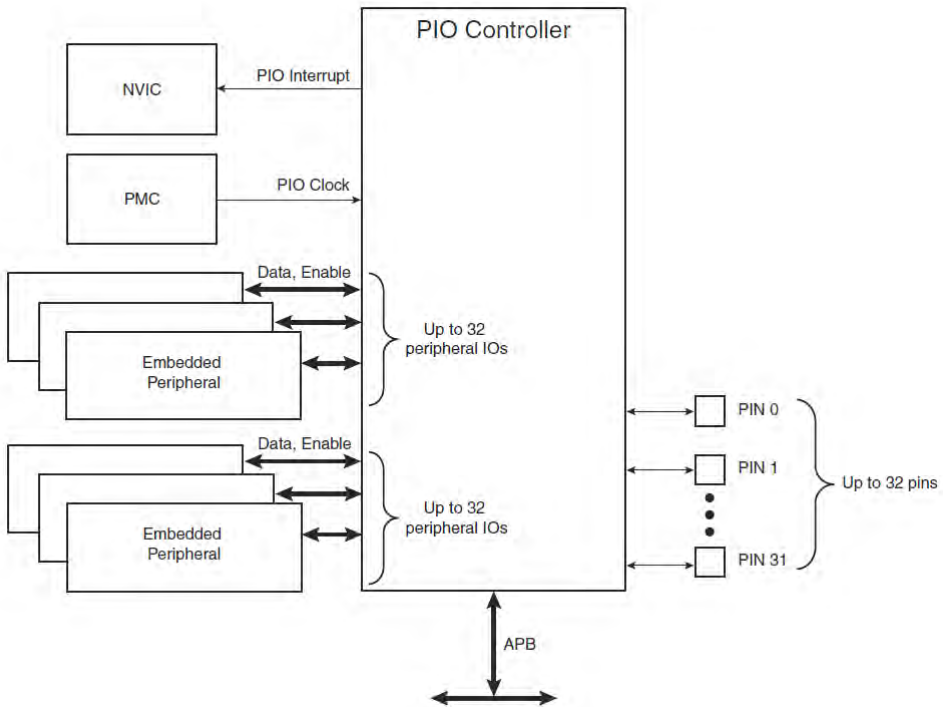
Block diagrams give you a hint of what dependencies a peripheral has on other parts of the microcontroller. This may be parts of the core, different signal busses, external pins, or other peripherals. Figure 5-8 on page 170 is the PIO Controller block diagram for the SAM3U from which we can learn some important information.

The obvious part of this diagram is that the PIO Controller connects directly to the physical hardware pins on the microcontroller which ultimately allows reading and writing digital logic signals to the world outside of the processor package (and in some cases analog levels, though that is not shown). The PIO ties into the processor's APB (Advanced Peripheral Bus) which is how most of the peripherals are wired back to the core.

Note that all of the PIO hardware and registers are duplicated because there are two PIO peripherals, PIOA and PIOB. They can be described generically, but they are physically independent of each other. The pins belonging to each are unique. Logically, the two sets of register addresses are offset from each other by a fixed number of bytes so it is easy to write code for both using the known offset value.

The two "Embedded Peripheral" rectangles refer to the other processor peripherals that can be multiplexed through the PIO Controller. If you wanted pin data to go in or out of a particular peripheral instead of the PIO controller, you would tell the PIO Controller to route the logic signals accordingly.





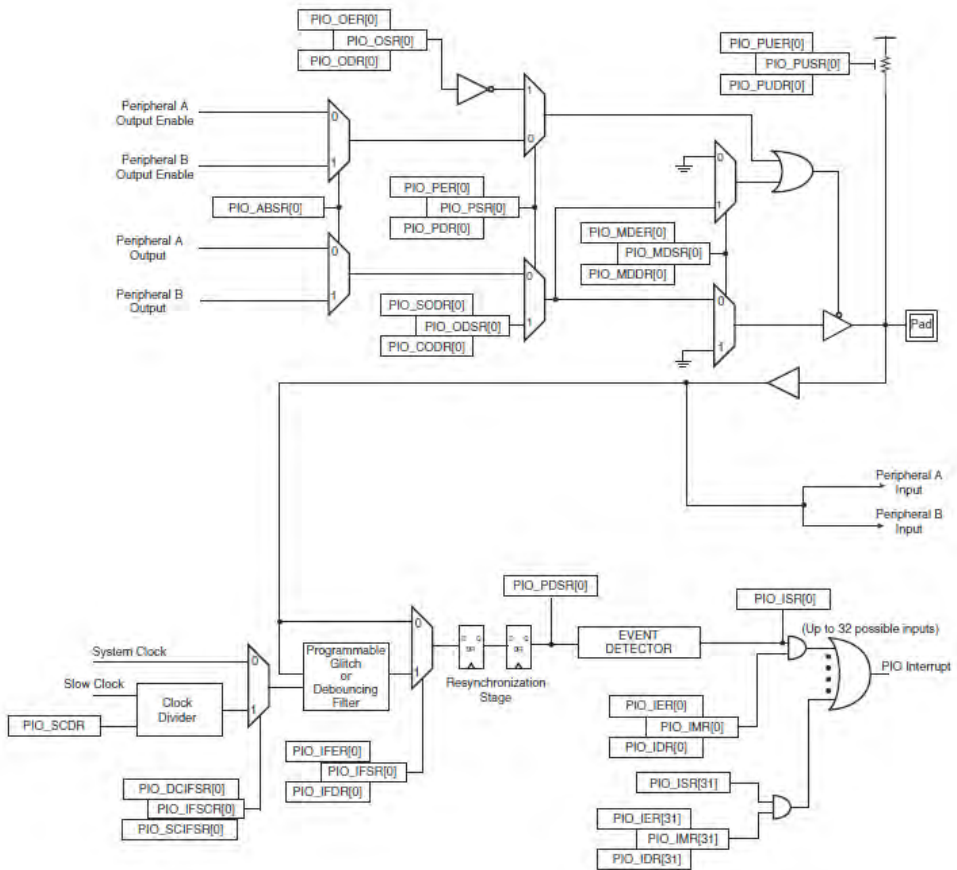
**Figure 5-8 PIO block diagram**

There is a single connection from a PIO Controller to the NVIC (Nested Vectored Interrupt Controller) where the PIO interrupt is provided. As you work with ARM Cortex processors you will see this is very standard and we will explore more about this in the Interrupt chapter.

A clock signal from the PMC (Power Management Controller) is required to drive the PIO Controller, which was mentioned in the previous chapter. To properly run the PIO Controller, the clock must be turned on in the PMC. Clock and power control are often a bit tricky to figure out! If you are working with a peripheral and getting unexpected results as you try to read and write the peripheral registers, this is a good indication that the peripheral might not have power and/or a clock. The most obvious clue is seeing that a peripheral's registers do not change at all when you write values to them when you're looking in the debugger. Some user guides are very clear at telling you this. Some don't give you any clue at all and you could spend hours trying to figure it out.

#### 5.4.1 • Logic Block Diagram

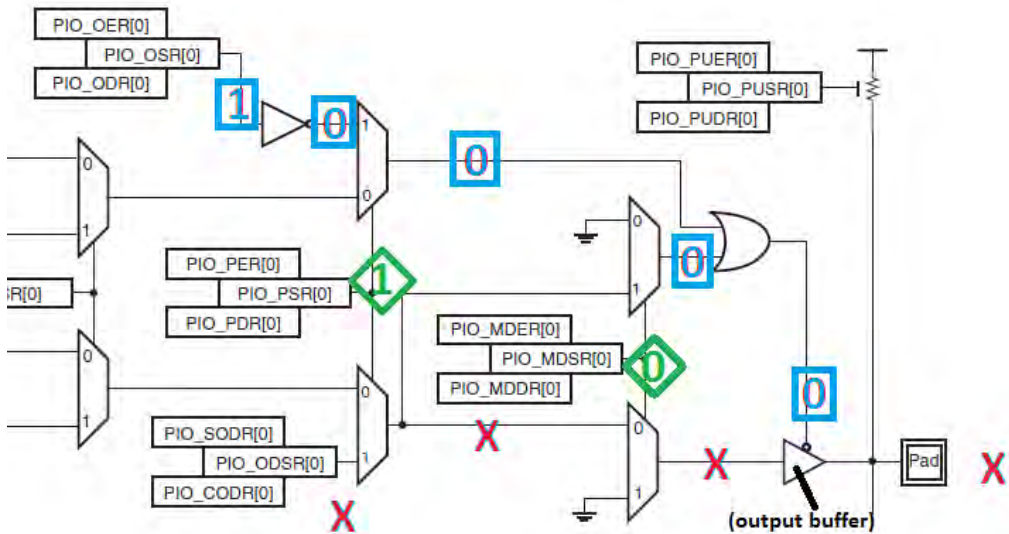
Microcontroller peripheral descriptions may also include a block diagram of the peripheral logic to give you a bird's eye view of how it works. These diagrams can end up being very useful to see how the registers influence the signal chain in the peripheral. Here is the main block diagram for the PIO peripheral in the SAM3U2 (See Figure 5-9 on page 171).



**Figure 5-9 I/O line control logic diagram**

This shows the hardware connected to one pin. In the actual processor, the registers would be connected to all the pins on a port (up to 32), but each pin has a copy of the logic hardware. Array notation `[]` is used to indicate which pin is being referenced. The block diagram references pin 0.

The effect of a lot of the PIO registers can be predicted without even knowing what they are. Let's use an example of the red LED (`PB_20_LED_RED`). Using the figure below, trace the impacts of a value in `PIOB_OER` (Output Enable Register) to see how it sets up the logic to allow the use of `PIOB_SODR` and `PIOB_CODR` (Set and Clear Output Data Register) to write the logic signal on the pin.



**Figure 5-10 Signal logic propagation through the PIO peripheral**

- The numbers in squares represent the logic levels that are propagating through the signal chain from PIO\_OER which first sets the value in PIO\_ODSR.
- The numbers in diamonds are the values to control the multiplexers (muxes) in the steering logic to get the signal. These come from PIO\_PSR and PIO\_MDSR.
- The "X" values are the LED state we want which would be in PIOB\_ODSR[20] depending on what we write to PIOB\_SODR[20] and PIOB\_CODR[20]. If X is 1, the voltage at PAD will be Vcc (logic high); if X is 0, the voltage at PAD is 0 (logic low).

By configuring PIOB\_OER[20] to logic 1, we are telling the processor that we want this pin to be an output. Ultimately this turns on the output buffer at the very end so the PAD signal is driven high or low depending on what X is at its input.

1. PIOB\_OER[20] is set which produces the logic 1 value into the inverter at the top left to get a logic 0 at the first mux.
2. The first mux is set to select path 1 which allows the PIOB\_ODSR logic 0 signal through. The complimentary mux here must then allow the PIOB\_ODSR[20] signal through (the current state of the LED we want).
3. The second pair of muxes are set for path 0 by PIO\_MDSR. Therefore, the OR gate output is 0, and the PIOB\_ODSR[20] is allowed to propagate through to control the output buffer.
4. The output buffer control sees the 0 from the OR gate which is inverted so it is turned on. Therefore, the X signal is driven on PAD.

In most cases, you do not have to look so carefully through the logic diagrams of a peripheral. Configuration values can typically be determined just by reading the register descriptions. However, understanding these diagrams and being able to trace through them to determine the register dependencies is really important especially if the configuration you set up does not work as expected.



A nice thing about the SAM3U2 micro is that the PIO peripheral registers are all one-to-one mapping, so no matter what PIO register you are working with if you are playing with bit “x” you are changing the behavior of pin “x” on that port. The downside of this is that there are quite a few registers to set up when initializing the processor.



Look at the “User Interface” section of the User Guide in the PIO section. You do not have to understand what all the registers are used for just yet (and in some cases, never), but you must get comfortable with how the information is organized and presented. You will be looking at register sets like this for every single peripheral driver you write in EiE and any other processor you work with. The register list for a peripheral tells you the following:

- **Offset:** the address of the register relative to the base address of the peripheral. If you skipped the last chapter, go back and read about this.
- **Register:** the full name of the register that usually doesn’t require any further explanation about its purpose. However, there may be individual bits that have special meaning.
- **Name:** the abbreviated logical name that you should be able to find in the header file for the processor so you can address the register.
- **Access:** indicates if you are allowed to read, write or do both operations on the register. Write-only registers will never change. In IAR version 8, their value is always WWWWWWWW. In some IDEs, these will show as 0x00000000. If they are read-only and you try to write to them, nothing should happen other than asking yourself why you are writing to read-only registers.
- **Reset:** gives you the defined initial conditions if there are any. If this column does not have a value, then there is no guarantee that it will start up with any particular setting, thus you should initialize it if you need to use it. Registers that are not hardware-initialized should retain their value when the processor is reset but not power cycled. However, you cannot be sure, so always initialize the registers you need.

On the SAMU32, registers are often grouped in threes with Enable, Disable, and Status registers. Similarly, there are Set, Clear, and Status register groups. Almost all the functionality provided by peripherals on the SAM3U2 are organized in this way. Look at the first set of these registers for the PIO. We briefly touched on these registers in the previous chapter.

Offset	Register	Name	Access	Reset
0x0000	PIO Enable Register	PIO_PER	Write-only	–
0x0004	PIO Disable Register	PIO_PDR	Write-only	–
0x0008	PIO Status Register	PIO_PSR	Read-only	(1)

**Figure 5-11 Register grouping example in the PIO peripheral**

This register grouping is responsible for assigning whether or not the PIO controller is operating the pin. If the PIO controller is assigned, the pin is general purpose digital and can be additionally configured and used as a high or low output for write operations, or high-impedance input for read operations. If it is not under PIO control, then a different peripheral controls the pin.

To set a bit in this register and thus assign the PIO controller to the pin, write the pin mask to the Enable register. To clear a bit and disable PIO control of the pin, write the pin mask to the Disable register. In both cases, you write the same mask. If you are watching these actions in the debugger, you will not see any change to the Enable and Disable

registers because they are write-only. To see the status of any bit, you must read the PIO\_PSR Status register. An example of writing the Enable register and then reading a copy of the Status register is shown. If you don't understand the pointer-to-struct register access, please review the previous chapter about this.

```
u32 u32PioStatusRegister;  
AT91C_BASE_PIOB->PIO_PER = PB_20_LED_RED;  
u32PioStatusRegister = AT91C_BASE_PIOB->PIO_PSR;
```

The purpose of each bit will be described for each register in the datasheet. For the SAM3U2, the descriptions follow the register summary table. Continue reading the user guide to see the detailed register descriptions. Most of the PIO registers operate “bit-wise” meaning that individual bits correspond to specific functions, but you will also come across peripheral registers where groups of bits have different functions. There are just two of these in the PIO register set, PIO\_WPMR, and PIO\_WPSR.

### 29.7.43 PIO Write Protect Status Register

**Name:** PIO\_WPSR  
**Address:** 0x400E0CE8 (PIOA), 0x400E0EE8 (PIOB), 0x400E10E8 (PIOC)  
**Access:** Read-only  
**Reset:** See [Table 29-2](#)

31	30	29	28	27	26	25	24
—	—	—	—	—	—	—	—
23	22	21	20	19	18	17	16
WPVSR							
15	14	13	12	11	10	9	8
WPVSR							
7	6	5	4	3	2	1	0
—	—	—	—	—	—	—	WPVS

- **WPVS: Write Protect Violation Status**

0 = No Write Protect Violation has occurred since the last read of the PIO\_WPSR register.

1 = A Write Protect Violation has occurred since the last read of the PIO\_WPSR register. If this violation is an unauthorized attempt to write a protected register, the associated violation is reported into field WPVSR.

- **WPVSR: Write Protect Violation Source**

When WPVS is active, this field indicates the write-protected register (through address offset or code) in which a write access has been attempted.

Note: Reading PIO\_WPSR automatically clears all fields.

**Figure 5-12 Write Protect status register**

## 5.5 • PIO Registers

Whatever the case, never guess at the purpose of the register settings. It is imperative to go through each register and understand each bit when first setting up a peripheral. As you do this, you can build the initialization values for the register and update the GpioSetup() function. To be complete, all the registers and their init values will be detailed here. Again, there is a lot of information that needs to be written in the header file if you follow the EiE documentation rules (over 1000 lines!). A substantial amount of time can be saved by setting up a template and doing a lot of cutting/copying and pasting as you fill out the values.

**PIO Enable / Disable / Status (PIO\_PER, PIO\_PDR, PIO\_PSR):** assigns the pin to the PIO controller. All the LEDs and buttons must be under the PIO control as well as circuit signals like the LCD reset line, and SD card control lines. Any circuit that requires a communications peripheral such as the LCD, SD and ANT must NOT be under PIO control. The associated Disable and Status registers are self-explanatory.

```
#define PIOA_PER_INIT (u32)0x84030007
#define PIOB_PER_INIT (u32)0x01BFFF57
```

**Output Enable / Disable / Status (PIO\_OER, PIO\_ODR, PIO\_OSR):** configures pins as a digital outputs. All LED pins must be set as outputs as well as any pin that drives a logic value out to another circuit. Some microcontrollers require pins that other peripherals output on to have the GPIO output function enabled. This is a good example of where the pin block diagram can be helpful if the operation is not immediately obvious from the datasheet. For the SAM3U2, when the pin is controlled by a Peripheral, there is a separate register that turns on the Output buffer, so the associated OER bit is not required.

```
#define PIOA_OER_INIT (u32)0x84010001
#define PIOB_OER_INIT (u32)0x01BFFFE0
```

**Glitch Input Filter Enable / Disable / Status (PIO\_IFER, PIO\_IFDR, PIO\_IFSR):** set to enable an optional built-in hardware filter that can remove fast changing signals. This is a cool option to have on every input pin to clean up noise in hardware rather than using firmware routines, but it won't be used for the EiE board. Features like this are nice to have as a backup, so don't forget about it.

```
#define PIOA_IFER_INIT (u32)0x00000000
#define PIOB_IFER_INIT (u32)0x00000000
```

**Set Output / Clear Output / Output Data Status (PIO\_SODR, PIO\_CODR, PIO\_ODSR):** Sets or clears the output drive on the PINs as long as the required other registers are configured. ODSR holds the values that are configured but does not necessarily mean that the pin level will be at that level. The pin voltage could be different if a different signal is driving the output, or if the pin is an input. Look at the block diagram to verify. The initialization values should be carefully chosen here so the circuit starts up correctly. The "Clear" register initialization is not necessarily the inverse of the "Set" register initialization since these values should only be operating on output pins.

```
#define PIOA_SODR_INIT (u32)0x88010000
#define PIOA_CODR_INIT (u32)0x30000000
#define PIOB_SODR_INIT (u32)0x01BFFE00
#define PIOB_CODR_INIT (u32)0x00000000
```

**Pin Data Status (PIO\_PDSR):** The true logic level on the pins. For pins that are configured as outputs, ODSR and PDSR should be the same. For input pins, PDSR can be totally different than ODSR. PDSR should always be used to read inputs.

**Interrupt Enable / Disable / Mask / Status (PIO\_IER, PIO\_IDR, PIO\_IMR, PIO\_ISR):** Allows any pin signal to contribute to the overall PIO interrupt signal from the PIO peripheral to the NVIC. More on this in the Interrupt module so it will not be initialized with the rest of the PIO registers.

**Multi-driver Enable / Disable / Status (PIO\_MDER, PIO\_MDDR, PIO\_MDSR):** This is more commonly called "open drain" output and it is brilliant that this is configurable on all pins of this processor. Some processors only have certain pins that are open-drain

and if you don't realize this and connect a regular digital IO to it, you are stuck. From the block diagram, we see that this allows both a peripheral and the ODSR to drive the output pin, sort of. In this state, a pull-up resistor would have to be used to get a logic high because the output buffer is always driving low. That's the essence of open-drain: only a low output is forced. To get a high output, the driver is off and in a high-impedance state so it will float if it is not pulled high. Only special hardware configurations need this. The EiE hardware requires open drain outputs on the ANT\_PWR\_EN line and the I<sup>2</sup>C (TWI) communication bus.

```
#define PIOA_MDER_INIT (u32)0x04000600
#define PIOB_MDER_INIT (u32)0x00000000
```

**Pull-up Enable / Disable / Status (PIO\_PUER, PIO\_PUDR, PIO\_PUSR):** Used to connect a built-in weak pull-up resistor after the output buffer on each pin. This can be a lifesaver if you accidentally need a pull-up on a signal line but forgot to put it on the PCB. It is also used commonly for open-drain drivers or just to ensure unused lines are not floating without having to consume board space with external resistors. It is safe to enable a pull-up resistor on an output line that is driven high or low, but in the low output state, a small amount of current will be wasted through the pull-up. From the electrical characteristics in the user guide, the nominal value of the pull-up is 100kOhms. The I<sup>2</sup>C lines need pull-ups, but 100k is too weak so external resistors are used. The internal pull-ups are great for test points to ensure they are not floating. Some processors also have configurable pull-down resistors.

```
#define PIOA_PPUER_INIT (u32)0x00000001
#define PIOB_PPUER_INIT (u32)0x000001C0
```

**Peripheral AB Select (PIO\_ABSR):** used to select which of two available peripheral outputs are chosen to drive the output when required. This is one of the few PIO registers that does not have a set/clear/status register group. The register is read-write so you can configure it or look at it. For every Peripheral-controlled pin used on the EiE development board, you must read the user guide section for that peripheral to see if the corresponding pins are mapped to peripheral A or B. There is no particular reason for either. You should read the user guide to understand where the following INIT values came from.

```
#define PIOA_ABSR_INIT (u32)0x7B000000
#define PIOB_ABSR_INIT (u32)0x00400018
```

**Filter configuration (SCIFSR, DIFSR, SCDR, IFDGSR):** These registers configure the behavior of the hardware filters. The first two select the "glitch" vs. "debounce" filter. The third allows a configurable time for the debounce. A fourth register, IFDGSR, is for the status of the current configuration. Even though the development board doesn't use the filters, the registers can still be configured. If the filters are required in the future, then it is easy to simply enable them just by updating the INIT values.

```
#define PIOA_SCIFSR_INIT (u32)0x00000000
#define PIOB_SCIFSR_INIT (u32)0x00000000
#define PIOA_DIFSR_INIT (u32)0x00000000
#define PIOB_DIFSR_INIT (u32)0x00000000
#define PIOB_DIFSR_INIT (u32)0x00000000
#define PIOB_SCDR_INIT (u32)0x00000000
```

**Output Write Enable / Disable / Status (PIO\_OWER, PIO\_OWDR, PIO\_OWSR):** A locking function to prevent accidental writes to the output data register. Writing directly to ODSR allows multiple output lines to change to either a high or low state all on the same instruction cycle. If you do not have this, it is always a two-step process to use the Set register to change pins high and then a second instruction to use the Clear register to pull pins low. If you needed to toggle two outputs simultaneously where one is high and one is low, you have to be able to write directly to the output register. Though sequential calls to SODR and CODR would normally involve only a few instruction cycles between them, that may be enough to cause a problem. A more likely problem would be that an interrupt (or a task-switch if using an RTOS) would occur between the two register writes and the pins would be in an undefined state for potentially much longer. This could actually be catastrophic to certain systems. Having the OWxR registers helps to solve a problem of messing up other pin IO levels when the write occurs. The EiE board uses this for the LED pins.

```
#define PIOA_OWER_INIT (u32)0x00000000
#define PIOB_OWER_INIT (u32)0x001FFC00
```

**Additional Interrupt Modes Enable / Disable / Mask (PIO\_AIMER, PIO\_AIMDR, PIO\_AIMMR):** This allows further customization of how the input interrupts work. The customization is configured in the ELSR and FRLHSR registers.

**Edge Select / Level Select / Status (PIO\_ESR, PIO\_LSR, PIO\_ELSR):** Defines interrupt behavior on input lines with interrupts enabled. A signal “Edge” is the transition from high to low (falling edge) or low to high (rising edge). The “Level” is the steady-state value.

**Falling Edge or Low Level / Rising Edge or High Level / Status (PIO\_FELLSR, PIO\_REHLSR, PIO\_FRLHSR):** For interrupt configured pins. Selects if the interrupt will be generated based on signal highs or lows (falling edges / low levels or rising edges / high levels).

**Lock Status / Write Protect Mode / Status (PIO\_LOCKSR, PIO\_WPMR, PIO\_WPSR):** Additional protection / security for IO behavior. Ultra-robust systems could definitely make use of these types of protection and status. However, we will not use these as it is beyond the scope of what we want to cover in EiE. The special “key” value is coded if it is needed later on.

```
#define PIO_WRITE_ENABLE (u32)0x50494F00
```



All of the required initialization values are ready. Now they need to be written to their corresponding registers in the GpioSetup module. Due to the nature of the PIO registers, the “Enable” register initialization values will always be the bit-wise opposite of the “Disable” registers. Write the code in GpioSetup. To keep things organized, do all the PIOA enable/disable registers first, then the PIO single-write registers. Then repeat for PIOB. Make sure you build and run through the code to check that everything is working. Most of the checking should be done with Register windows in the debugger, but you should see all the LEDs turn on. See Figure 5-13 on page 179.

```
void GpioSetup(void)
{
    /* Set all of the pin function registers in port A */
    AT91C_BASE_PIOA->PIO_PER = PIOA_PER_INIT;
    AT91C_BASE_PIOA->PIO_PDR = ~PIOA_PER_INIT;
    AT91C_BASE_PIOA->PIO_OER = PIOA_OER_INIT;
    AT91C_BASE_PIOA->PIO_ODR = ~PIOA_OER_INIT;
    AT91C_BASE_PIOA->PIO_IFER = PIOA_IFER_INIT;
    AT91C_BASE_PIOA->PIO_IFDR = ~PIOA_IFER_INIT;
```

```

AT91C_BASE_PIOA->PIO_MDER = PIOA_MDER_INIT;
AT91C_BASE_PIOA->PIO_MDDR = ~PIOA_MDER_INIT;
AT91C_BASE_PIOA->PIO_PPUER = PIOA_PPUER_INIT;
AT91C_BASE_PIOA->PIO_PPUDR = ~PIOA_PPUER_INIT;
AT91C_BASE_PIOA->PIO_OWER = PIOA_OWER_INIT;
AT91C_BASE_PIOA->PIO_OWDR = ~PIOA_OWER_INIT;

AT91C_BASE_PIOA->PIO_SODR = PIOA_SODR_INIT;
AT91C_BASE_PIOA->PIO_CODR = PIOA_CODR_INIT;
AT91C_BASE_PIOA->PIO_ABSR = PIOA_ABSR_INIT;
AT91C_BASE_PIOA->PIO_SCIFSR = PIOA_SCIFSR_INIT;
AT91C_BASE_PIOA->PIO_DIFSR = PIOA_DIFSR_INIT;
AT91C_BASE_PIOA->PIO_SCDR = PIOA_SCDR_INIT;

/* Set all of the pin function registers in port B */
AT91C_BASE_PIOB->PIO_PER = PIOB_PER_INIT;
AT91C_BASE_PIOB->PIO_PDR = ~PIOB_PER_INIT;
AT91C_BASE_PIOB->PIO_OER = PIOB_OER_INIT;
AT91C_BASE_PIOB->PIO_ODR = ~PIOB_OER_INIT;
AT91C_BASE_PIOB->PIO_IFER = PIOB_IFER_INIT;
AT91C_BASE_PIOB->PIO_IFDR = ~PIOB_IFER_INIT;
AT91C_BASE_PIOB->PIO_MDER = PIOB_MDER_INIT;
AT91C_BASE_PIOB->PIO_MDDR = ~PIOB_MDER_INIT;
AT91C_BASE_PIOB->PIO_PPUER = PIOB_PPUER_INIT;
AT91C_BASE_PIOB->PIO_PPUDR = ~PIOB_PPUER_INIT;
AT91C_BASE_PIOB->PIO_OWER = PIOB_OWER_INIT;
AT91C_BASE_PIOB->PIO_OWDR = ~PIOB_OWER_INIT;

AT91C_BASE_PIOB->PIO_SODR = PIOB_SODR_INIT;
AT91C_BASE_PIOB->PIO_CODR = PIOB_CODR_INIT;
AT91C_BASE_PIOB->PIO_ABSR = PIOB_ABSR_INIT;
AT91C_BASE_PIOB->PIO_SCIFSR = PIOB_SCIFSR_INIT;
AT91C_BASE_PIOB->PIO_DIFSR = PIOB_DIFSR_INIT;
AT91C_BASE_PIOB->PIO_SCDR = PIOB_SCDR_INIT;
} /* end GpioSetup() */

```

GPIO setup is a lot of work, especially if you take the time to properly look at and configure every register in the peripheral. When this all works, it is more reason to truly celebrate even though the result is barely noticeable on the board. What we have now is a fully configured development board from the IO perspective that is ready to be programmed.

## 5.6 • Adding a New Task

Each driver that is developed in EiE will be a new task. Any application you write should also be a new task. Adding a new task should never involve having to write or modify code in any other part of the system beyond the basic inclusions necessary to add the task in. The only exception is if you are adding to the existing API to provide additional functionality. For all of EiE, if you think you need to change any of the existing sources files you're not doing something correctly.

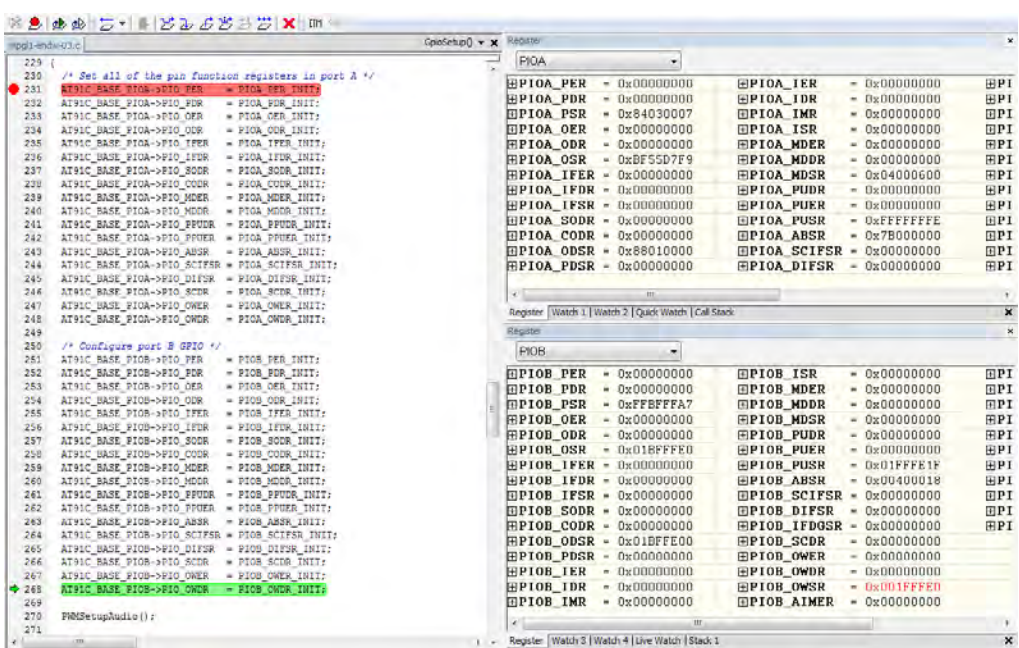


Figure 5-13 Register window in the debugger

In most cases, a new task is just a source file and a header file. Template files called `user_app1.c` and `user_app1.h` are provided in `\firmware_common\application`. They have the basic EIE structure and directions to use them at the top of each file. The intent is that you copy these files, rename them to the name of your task, and then insert them into the appropriate place in the project file structure.

Once new task files are added to the project, the task needs to be coupled into a few files in the system. We'll demonstrate putting in the User Task and then it will be there to use for all of the chapter exercises.



To add the files in, right-click on the "Application" file group in the Workspace and choose **Add... > Add Files**. Find the folder where your new source files are (in this case `\firmware_common\application`) and select both the `.c` and `.h` files. Drag them into their respective Include and Source sub-groups.

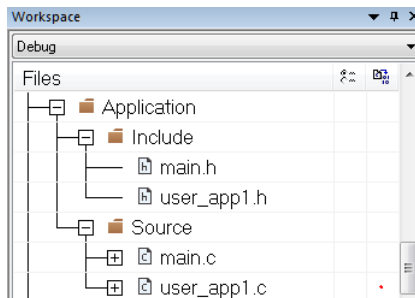


Figure 5-14 "Application" file group in the Workspace



Add the call to your task's initialization function in the appropriate place in the init section of main(). In the case of this example, we are adding a high-level application so the call should be made somewhere in the Application initialization section.

```
/* Application initialization */
UserApp1Initialize();
```



Next, add a call to the task's "RunActiveState" function inside the super loop. The order of function calls here usually does not matter. There may be special cases where you might want a driver function to run before an application function, but if you have coded something that relies on this, you might want to revisit what you have designed to make it position-independent. For consistency, use the same order as the initialization functions.

```
/* Super loop */
while(1)
{
    WATCHDOG_BONE();

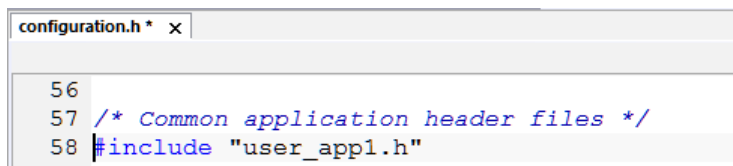
    /* Applications */
    UserApp1RunActiveState();

    /* System sleep */
    HEARTBEAT_OFF();
    do
    {
        SystemSleep();
    } while(G_u32SystemFlags & _SYSTEM_SLEEPING);

    HEARTBEAT_ON();
} /* end while(1) main super loop */
```



Open configuration.h and add the #include for "user\_app1.h" in the appropriate section.



```
configuration.h * x
56
57 /* Common application header files */
58 #include "user_app1.h"
```

Figure 5-15 Add user\_app1.h header in configuration.h



Build the code and make sure there are no errors. The task is ready to be coded.

Once your task is fully developed, make sure it meets all of the standards for use in the EiE system. Your code should be carefully documented to indicate what dependencies it has. For example, if it needs a UART, indicate that in a paragraph at the start of the .c source file. You do not need to document all of the private resources it needs, though it would be nice to include a summary of memory usage and any instructions about special requirements involved in adding the code to a similar system.

Your task should also have a documented API for the functionalities it provides (if any). A brief summary should be written at the top of the source file. Regenerate the Doxygen files and make sure that your task appears and has been correctly captured.

## 5.7 • The LED Driver

As a designer, your task is to identify a problem, fully understand and describe that problem, and then create a solution that addresses all aspects of the problem. This is the essence of engineering, and no matter how big or how small the problem is, good design follows a careful process. The first driver we will write provides a few functions to manage LEDs. In this case, the problem is relatively simple, but the solution still needs some careful planning and probably more work than you're expecting.

The design is critically important. Designing a task will often take longer than coding it. If you think through all of the problems and try to predict solutions, writing the code ends up being much easier. The code will be more efficient and robust, and there will be fewer surprises that can dramatically distort the amount of time and effort you predicted is required.

In firmware design, it is important to consider the following:

- What are all the current features and requirements of the program?
- What future features and requirements are reasonable to predict?
- What are the inputs and outputs to the system?
- What data structures can be used to efficiently handle information in the task?
- What hardware or other driver dependencies are there?
- If the task provides an API, what functions should be written?
- How much integration or abstraction is reasonable and worth doing in the task?
- Who or what other functions will use the task?
- How can the task be broken down into independent states or functions that can be coded?
- Are there any big problems to solve where a flowchart or other documentation is needed?

This list could go on. Depending on the environment you are working in, the design plan might be done very formally, or it might be done with basic sketches and rough notes in your notebook. Designing EVERYTHING up front does not make sense and is sometimes impossible. The process is likely more iterative with respect to implementing each state or function, but having an overall picture of what you are going to do and how you are going to do it is important. Some details are definitely left open and filled in along the way.

With each task described in EiE, the design process will be captured as much as possible. While it will be presented very sequentially in this book thanks to 20-20 hindsight, the actual process is likely more chaotic. We like to think that what you read is exactly how things work every time, but of course, that never happens. The most misleading thing about doing this is that the really hard problems and bugs that popped up during design are solved and possibly not even mentioned. You are encouraged to do all of this on your own as much as possible! If you can take the API summary and design the code yourself, you should be able to drop in your code to the final EiE firmware suite and any application should still work.



To get started, add the `leds.c` and `leds.h` files in `\firmware_common\drivers` to your project the same way that was just described for the user task. Put in the call to `LedInitialize()` in a new "Driver initialization" section of `main.c`. Add the call to `LedRunActiveState()` in a new "Drivers" section in the main loop. Don't forget to `#include leds.h` in the `configuration.h` file. The `#include` needs to be after the `eief1-pcb-01.h` `#include`. Build the code to make sure everything has been added correctly.

So far, we have toggled LEDs by writing directly to port registers, and at some point, the LED driver functions that are created will do the same thing. Working at the register level is complicated and requires very specific knowledge of the hardware. A good driver completely abstracts the hardware and presents an interface to an application that will function seamlessly without any knowledge of the underlying hardware. Not only does this make the application programmer happy, it contributes to code portability. An application written for this development board could be ported to any development board as long as the driver API remained the same.

For example, if you had 10 different products each with different processors and hardware configurations and you wanted to write a program to blink a green LED, having a driver API that supported a function call like `LedBlink (GREEN)` regardless of the platform would be extremely handy. Though the driver firmware would be different on each board, the application would not have to change (it would just have to be recompiled for the platform in use). While this is a very simple example, you can imagine how that expands to larger applications and becomes very attractive. The trade-off is that code to enable this kind of portability will be much longer than directly writing to the port registers. In a mature line of products, there will be much more time invested in high-level code that needs to “just work” as the underlying hardware evolves.

The LED driver for EiE will provide basic LED functionality. Defining the API functions upfront is critical because the firmware design will depend heavily on what the entire functionality set has to provide. Take a moment to imagine the features of an LED driver. The basic operations that we want to provide to applications are:

1. Turn an LED on
2. Turn an LED off
3. Toggle an LED
4. Blink an LED

Since development boards likely have more than one LED, we need a way to tell the functions what LED is being referred to. In the simplest case, you could just use the color (e.g. `LED_BLUE`, `LED_RED`, etc.). That assumes that there is only one LED per color. On the EiE ASCII development board, there are 8 discrete LEDs and three more in the LCD backlight. This part of the design has to be board-specific, so a strategy to abstract it as much as possible was formed. Ultimately, we want the API to provide a simple function call with an intuitive parameter for the LED of interest.

```
LedOn(<LED name>);  
LedOff(<LED name>);  
LedToggle(<LED name>);  
LedBlink(<LED name>, <rate>);
```

Where `<LED name>` will be the constant `WHITE`, `PURPLE`, `BLUE`, `CYAN`, `GREEN`, `YELLOW`, `ORANGE`, `RED`, `LCD_RED`, `LCD_GREEN`, `LCD_BLUE`. The user would be hard-pressed to ask for an easier interface than that.

The LED functionality we want in the API will work almost identically regardless of the LED. In this situation, we immediately think “array” to store anything that is LED-specific so that the same code can run but use an index to get any details that are unique to the LED at each index. There are three parameters that need to be considered for an LED: the pin, the port, and if it is active-high or active-low. The address between like registers in different ports is always a fixed offset. In the case of `PORTA` to `PORTB`, this is `0x80`. When working with digital pins, these parameters will always factor into how the code uses the pin, so it can be generalized.



In `typedefs.h`, define two enum types for the port offset and active-high/low parameters so that these will be self-documenting. Groups of parameters are arranged and stored nicely in structs, so create a struct type called `PinConfigurationType` for this information. The bit position is just `u32` type. The enums need to be defined first so they can be used inside the struct.

```
typedef enum {PORTA = 0, PORTB = 0x80} PortOffsetType;
typedef enum {ACTIVE_LOW = 0, ACTIVE_HIGH = 1} GpioActiveType;

typedef struct
{
    u32 u32BitPosition;           /*!< @brief Pin bit position within port */
    PortOffsetType ePort;         /*!< @brief Pin port position */
    GpioActiveType eActiveState;  /*!< @brief Pin hardware active type */
}PinConfigurationType;
```

Switch to `eief1-pcb-01.h`. To specify an LED and index the configuration array with a name that makes sense, create an enum type called “`LedNameType`.” We show the full comment here because it is important to document the relationship of this typedef to the LED array. Also, #define a constant for the number of LEDs in the system. This could be done automatically with the `sizeof()` function, but since that’s calculated in preprocessing, it might cause trouble for the compiler if it needs the value before it’s calculated. It’s also good to make the user more consciously aware of this value.



```
/*!
@enum LedNameType
@brief Logical names for LEDs in the system.

The order of the LEDs in LedNameType must match the order of the definition
in G_asBspLedConfigurations from eief1-pcb-01.c
*/
typedef enum {WHITE = 0, PURPLE, BLUE, CYAN, GREEN, YELLOW, ORANGE, RED, LCD_RED,
LCD_GREEN, LCD_BLUE} LedNameType;

#define U8_TOTAL_LEDS (u8)11 /*!< Total number of LEDs in the system */
```

The corresponding array is a global const array called `G_au32BspLedPinPositions` in `eief1-pcb-01.c` which lists all of the LED pins in the same order as `LedNameType`. By declaring it const, these values don’t have to sit on the stack consuming RAM.



```
/*! LED locations: order must correspond to the order set in LedNameType in the
header file. */
const PinConfigurationType G_asBspLedConfigurations[U8_TOTAL_LEDS] =
{ {PB_13_LED_WHT, PORTB, ACTIVE_HIGH},
  {PB_14_LED_PRP, PORTB, ACTIVE_HIGH},
  {PB_18_LED_BLU, PORTB, ACTIVE_HIGH},
  {PB_16_LED_CYN, PORTB, ACTIVE_HIGH},
  {PB_19_LED_GRN, PORTB, ACTIVE_HIGH},
  {PB_17_LED_YLW, PORTB, ACTIVE_HIGH},
  {PB_15_LED_ORG, PORTB, ACTIVE_HIGH},
  {PB_20_LED_RED, PORTB, ACTIVE_HIGH},
  {PB_10_LCD_BL_RED, PORTB, ACTIVE_HIGH},
  {PB_11_LCD_BL_GRN, PORTB, ACTIVE_HIGH},
  {PB_12_LCD_BL_BLU, PORTB, ACTIVE_HIGH}
};
```

Make sure you add the “extern” declaration for `G_au32BspLedPinPositions` in `leds.c`.

There are aspects of this design approach that we don’t like because of the dependencies of `leds.c` on the board configuration file. We considered using an API function in `leds.c` that would let other tasks “register” LEDs that were available in the system. This would allow `leds.c` to be totally autonomous to any board. In a previous version of `leds.c`, we also offered the ability for a task to check out an LED so it could be used exclusively by the task and not have conflicts with other tasks. Both of these features would be nice-to-haves but given the applications anticipated, the overhead is not worth it.

## 5.8 • Driver Implementation

Now the fun part: realizing the design in firmware. There are many ways to implement the driver and like any programming solution, they range from simple, brute-force methods, to potentially complicated yet elegant solutions.

All tasks in the EiE system must be coded to execute as quickly as possible to minimize their contribution to the 1ms total loop time. For a lot of the drivers that are written, a program that uses API functions will cause a change in the behavior of the associated state machine that will be carried out over multiple main loop cycles. For the basic LED functions ON, OFF, and TOGGLE, these can happen instantaneously since it is just as fast to update the LEDs immediately as it would be to queue those changes for the next run of the LED task. The blinking feature is different, where the API call should set up the blinking parameters, but the state machine will need to always be updating the timing and LED state.

State diagrams are really helpful to break down a big problem into smaller problems that are much easier to solve. States allow you to stop worrying about every detail at once and just focus on what the task needs to do when certain conditions are true. For the LED driver, only one state is needed, although you could add other features like a sleep mode that might involve adding more states. The value of drawing a diagram for a single state is debatable, but let’s do it to start practicing. Our state diagrams usually show the states and provide a very brief description of what is happening in that state.

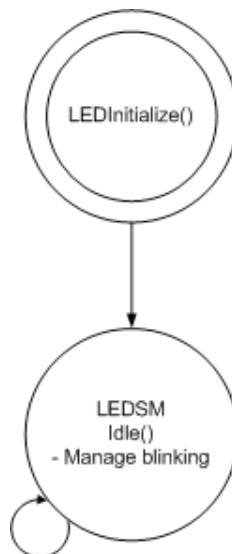


Figure 5-16 State diagram



How do we know if an LED is blinking? We can define “LedModeType” enum to track if the LED is in “LED\_NORMAL\_MODE” or “LED\_BLINK\_MODE”. Blinking requires a counter to keep track of the time, and every time the counter reaches 0 it must be reset back to the original value. This requires a counter variable and a saved starting value. The 1ms loop time can be used for timing. How will the user know what to specify for the blink rate? It makes sense to provide some specific values to make it easy, so “LedRateType” enum is defined, too.

```
/*!
@enum LedModeType
@brief The mode determines how the task manages the LED */
typedef enum {LED_NORMAL_MODE, LED_BLINK_MODE} LedModeType;

/*!
@enum LedRateType
@brief Standard blinky values for blinking.

Other blinking rate values may be added as required. The labels are frequencies,
but the values are the toggling period in ms.
*/
typedef enum {LED_0HZ = 0, LED_0_5HZ = 1000, LED_1HZ = 500, LED_2HZ = 250, LED_4HZ
= 125, LED_8HZ = 63} LedRateType;
```



Since every LED might be running differently, this suggests an array of parameters that line up with the LED array should be used to track this. The parameters are another group of information, so define a struct typedef in leds.h to hold this data.

```
/*!
@struct LedControlType
@brief Required parameters for the task to track what each LED is doing.
*/
typedef struct
{
    LedModeType eMode;        /* Current mode */
    LedRateType eRate;        /* Current rate */
    u16 u16Count;             /* Value of current duty cycle counter */
}
```

The control data array is global to leds.c so define it in the appropriate section at the top of the file.

```
/******
Global variable definitions with scope limited to this local application.
Variable names shall start with "Led_<type>" and be declared as static.
*****/


static LedControlType Led_asControl[U8_TOTAL_LEDS];
/*!< @brief Holds individual control parameters for LEDs */
```



Since we have an array of control data, it needs to be initialized which will be the first code we add to LedInitialize(). Starting all the LEDs in “normal” mode with LED\_0HZ and the counter at 0 seems to make sense.

```
/* Initialize the LED control array */
for(u8 i = 0; i < U8_TOTAL_LEDS; i++)
{
    Led_asControl[i].eMode = LED_NORMAL_MODE;
    Led_asControl[i].eRate = LED_0HZ;

    Led_asControl[i].u16Count = 0;
}
```

 Add the function declarations and definition placeholders for the three basic functions. LedOn() is shown here. The other two are nearly identical. When called, the three functions will always set the LED to normal mode otherwise there could be some confusion.

```
/*!-----
@fn void LedOn(LedNameType eLED_)

@brief Turn the specified LED on.

This function automatically takes care of the active low vs. active high LEDs. The
function works immediately (it does not require the main application loop to be
running).

Requires:
- Definitions in G_asBspLedConfigurations[eLED_] and Led_asControl[eLED_] are cor-
rect

@param eLED_ is a valid LED index

Promises:
- eLED_ is turned on
- eLED_ is set to LED_NORMAL_MODE mode

*/
void LedOn(LedNameType eLED_)
{
```

The syntax to handle the LED correctly is very low-level. To make the function run generically, we need to use the active-low or active-high parameter for the LED to choose the base address to either the set or the clear PIO register. Then the offset between the two ports is added. Follow the code carefully to see how this works. Indexing the LED arrays involves a little bit of a hack in that we typecast eLED since it is not technically a number that should be used for indexing. Since this enum is carefully documented to be LED indexes in the array, it's ok. Some people will freak out at this suggestion. The mess of a value into pu32OnAddress also needs to be typecasted for the compiler not to complain.

```
u32 *pu32OnAddress;

/* Configure set and clear addresses */
if(G_asBspLedConfigurations[(u8)eLED_].eActiveState == ACTIVE_HIGH)
{
    /* Active high LEDs use SODR to turn on */
    pu32OnAddress = (u32*)&(AT91C_BASE_PIOA->PIO_SODR) +
                    G_asBspLedConfigurations[(u8)eLED_].ePort;
}
else
{
    /* Active low LEDs use CODR to turn on */
```



```

    pu32OnAddress = (u32*)(AT91C_BASE_PIOA->PIO_CODR) +
                    G_asBspLedConfigurations[(u8)eLED_].ePort);
}

```



Once the address is ready, write the LED's bit mask to it. As a final step, make sure the LED is in LED\_NORMAL\_MODE.

```

/* Turn on the LED */
*pu32OnAddress = G_asBspLedConfigurations[(u8)eLED_].u32BitPosition;

/* Always set the LED back to LED_NORMAL_MODE mode */
Led_asControl[(u8)eLED_].eMode = LED_NORMAL_MODE;

```



Build the code to make sure everything is working. If errors stating “LedNameType is not defined”, check configuration.h to ensure that the #include for leds.h is AFTER the #include for eief1-pcb-01.h. A warning that Led\_asControl was set but never used is expected at this point.



Now do LedOff() and see if you can figure out LedToggle(). Toggling is much easier because you do not have to worry about active high or low. Instead, you can just XOR the ODSR register with itself and the LED bit mask. If you copy code as the base for different code, be very careful to update all of the comments that need to change or else you risk some very confusing documentation! Try not to forget the function's header file declarations (you will, and you will get errors).

```

void LedToggle(LedNameType eLED_)
{
    u32*pu32Address = (u32*)(AT91C_BASE_PIOA->PIO_ODSR) +
                    G_asBspLedConfigurations[eLED_].ePort);

    *pu32Address ^= G_asBspLedConfigurations[(u8)eLED_].u32BitPosition;

    /* Set the LED back to LED_NORMAL_MODE mode */
    Led_asControl[(u8)eLED_].eMode = LED_NORMAL_MODE;
} /* end LedToggle() */

```



The LED task can be self-testing if all the LEDs are turned on during initialization for a short period of time. Since this code runs in the initialization section of main, there are no timing requirements so the delay can just be a long loop in LedInitialize(). After about half a second, turn off all the discrete LEDs but leave the LCD backlight LEDs on so the screen looks alive. Try LedOff() and LedToggle() to turn the lights off to confirm both functions. To be more interesting, you can use a loop to sequentially turn on the LEDs with a short delay. This requires another typecast hack to make the loop variable a LedNameType and avoid a compiler warning. Yes, we know the enum abuse is terrible, but we're doing it under very controlled conditions.

```

for(u8 i = 0; i < U8_TOTAL_LEDS; i++)
{
    LedOn( (LedNameType)i );
    for(u32 j = 0; j < 300000; j++);
}

LedOff(WHITE);
LedOff(PURPLE);
LedOff(BLUE);
LedOff(CYAN);
LedToggle(GREEN);

```

```
LedToggle(YELLOW);
LedToggle(ORANGE);
LedToggle(RED);
```

Hint: if the LEDs are not behaving as expected, check (and adjust) the `PIOB_SODR_INIT` value used in `GpioSetup`. We chose to initialize the LEDs to the ON state with the setup value, but now that we have an LED driver, it makes more sense to initialize to the OFF state during `GpioSetup` and let the LED driver bring up the LEDs.

## 5.9 • Blinking

Adding the blinking functionality involves a public API function to set up the blinking parameters, and then the LED Idle state to check each LED to see if anything needs to be updated. Look at the function header below and then write the code to implement it.

```
/*!-----
@fn void LedBlink(LedNameType eLED_, LedRateType eBlinkRate_)

@brief Sets eLED_ to LED_BLINK_MODE with the rate given.

Requires:
@param eLED_ is a valid LED index
@param eBlinkRate_ is an allowed blinking rate from LedRateType

Promises:
- eLED_ is set to LED_BLINK_MODE at the blink rate specified

*/
```

The code simply sets up the `Led_asControl` members for the LED specified in the API call.

```
void LedBlink(LedNumberType eLED_, LedRateType eBlinkRate_)
{
    Led_asControl[(u8)eLED_].eMode = LED_BLINK_MODE;
    Led_asControl[(u8)eLED_].eRate = eBlinkRate_;
    Led_asControl[(u8)eLED_].u16Count = eBlinkRate_;
} /* end LedBlink() */
```

The Idle state to blink the LED should use a loop to parse through `Led_asControl` to find any LEDs that are in `LED_BLINK_MODE`. For each one it finds, the LED's counter value should be decremented. If the counter has reached 0 it's time to toggle the LED. `LedToggle()` cannot be used here because it will change the LED back to `LED_NORMAL_MODE`. You can, however, just copy that part of the code from `LedToggle()` into the Idle state. Give it a try and check your solution below.

```
/*!-----
@fn static void LedSM_Idle(void)

@brief Run through all the LEDs to check for blinking updates.
*/
static void LedSM_Idle(void)
{
    u32 *pu32Address;

    /* Loop through each LED to check for blinkers */
    for(u8 i = 0; i < U8_TOTAL_LEDS; i++)
```

```

{
    /* Check if LED is in LED_BLINK_MODE mode */
    if(Led_asControl[(LedNumberType)i].eMode == LED_BLINK_MODE)
    {
        /* Decrement counter and check for 0 */
        if( --Led_asControl[(LedNumberType)i].u16Count == 0)
        {
            /* Toggle and reload the LED */
            pu32Address = (u32*)&(AT91C_BASE_PIOA->PIO_ODSR) +
                G_asBspLedConfigurations[(u8)eLED_].ePort;
            *pu32Address ^= G_asBspLedConfigurations[(u8)eLED_].u32BitPosition;
            Led_asControl[(LedNumberType)i].u16Count =
                Led_asControl[(LedNumberType)i].eRate;
        }
    }
} /* end for(u8 i = 0; i < U8_TOTAL_LEDS; i++) */
} /* end LedSM_Idle() */

```



To test the blinking mode, write a short piece of code in `UserApp1SM_Idle` to cycle through and turn on your favorite LED color at different blinking rates every 2 seconds. Start at `LED_8HZ` and go to `LED_1HZ`. Repeat forever.

```

static void UserApp1SM_Idle(void)
{
    static u32 u32Counter = 0;

    u32Counter++;

    if(u32Counter == 2000)
    {
        LedBlink(PURPLE, LED_8HZ);
    }

    if(u32Counter == 4000)
    {
        LedBlink(PURPLE, LED_4HZ);
    }

    if(u32Counter == 6000)
    {
        LedBlink(PURPLE, LED_2HZ);
    }

    if(u32Counter == 8000)
    {
        LedBlink(PURPLE, LED_1HZ);
        u32Counter = 0;
    }
} /* end UserApp1SM_Idle() */

```

The counter variable that tracks the time must be static so it persists between calls to the Idle state. You must be careful to not call the Blinking function every loop cycle. Every time it is called, the control values are reset. If it is called every time the Idle state runs (every 1ms), the counters will never move off their 0 values and the LEDs won't blink. Understanding the system timing will always be critical in EIE!

### 5.9.1 • Map File



The last thing to cover in this chapter is a special file that is output by the linker whenever the code is built. You can find this in the “Output” folder in the Workspace window. If it’s not there, make sure that “Generate linker map file” is enabled inside Project Options > Linker > List. Make sure your code has been built and the .map file should appear. Double-click to open it.

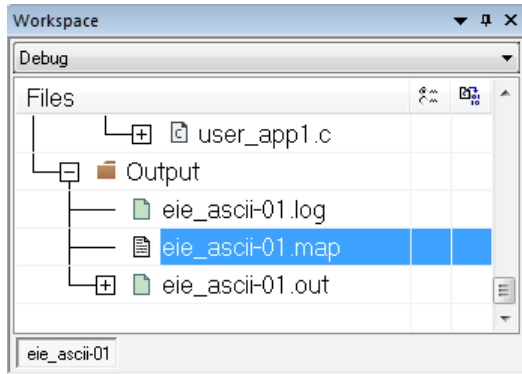


Figure 5-17 Map file location

The .map file is a summary of all of the objects, symbols, code sections, etc. that the Linker is aware of in putting together the final binary file that will be flashed to your processor. Not only can this be a powerful debugging tool, it is generally informative in that it tells you what resources all of the different parts of your program are using. The “MODULE SUMMARY” is in the middle of the file somewhere. It shows the flash space used (ro code and ro data where “ro” is read-only), and the RAM used (rw data where “rw” is read/write). Below is the summary from all the source code that we have built so far in this chapter.

```

*** MODULE SUMMARY
***

```

Module	ro code	ro data	rw data
D:\EiE\EiE_Git\eiobook\firmware_ascii\iar_8_10_1\Debug\Obj: [1]			
board_cstartup_iar.o	20	188	
eief1-pcb-01.o	720	88	
exceptions.o	78		
kill_x_cycles.o	14		
leds.o	596		72
main.o	88		12
user_app1.o	124		8
-----			
Total:	1 640	276	92
command line: [2]			
-----			
Total:			
dl7M_tln.a: [3]			
exit.o	4		
-----			
Total:	4		

```

rt7M_tl.a: [4]
  cexit.o           10
  cmain.o           26
  cstartup_M.o      12
  data_init.o       40
  zero_init3.o      64
-----
Total:              152

shb_l.a: [5]
  exit.o            20
-----
Total:              20

Linker created                      16    4 096
-----
Grand Total:          1 816    292    4 188

```



The first section covers the code that we have added to the project. It is a total of 1640 bytes of flash memory for code + 276 bytes of flash for data (like the `G_asBspLedConfigurations` array in `eief1-pcb-01.c`). 92 bytes of RAM is being used. Try removing the “const” part of the `G_asBspLedConfigurations` declaration and see how the resources change.

The other code usage is from the C-runtime library functions that end up getting compiled and linked in. If you included libraries like `string.h`, you would start to see those object files taking memory, also. The linker is smart enough to strip out any functions or object files that are not used, so you are not wasting flash or RAM. The 4k “Linker created” chunk of RAM at the end is from the Stack declaration in the linker file. Since we are not using any dynamic memory allocation yet, the linker is not allocating any Heap space.

If you have worked on simple 8-bit micros like PICs that sometimes only have 1k or 2k of flash, consuming all of that for startup code and a little light blinking probably seems excessive. Perhaps the appropriate proverb at this time is that to fire a bigger bullet, you need a bigger gun. A complicated 32-bit embedded system will indeed take more memory to initialize and get started. We are also being very pragmatic in what we are doing. And of course, every line of assembly code in a 32-bit system requires 4 bytes of memory, although Cortex processors have many 2-byte instructions.

As your programs get more complicated, your resource usage will indeed rise as well. If doing nothing costs 1640 bytes, then doing something must cost infinite bytes? That, of course, is not true. What you will see as you write more complicated firmware is that there is almost a standard amount of memory usage involved with any new object file, but the file will grow more slowly than you might expect. Startup code is brute force and takes space. Clever solutions to problems with careful thought and good coding practices in a high-level language will assemble down into something much smaller.

All this becomes critically important when you try to build code and get a linker error that says the code size exceeds the available resources! Embedded designers do not like this message, but guaranteed you will see it one day. Having the `.map` file available to see what pieces of code are perhaps hogging all the memory can help you target efforts to shrink your code and make it fit. It is often very easy to free up some RAM by (carefully) shrinking buffer sizes or even (very carefully) reducing the stack size. Freeing up flash space can be surprisingly easy in some cases if certain functions have been very poorly written perhaps due to a lack of knowledge about the processor and assembly language.

### 5.10 • Chapter Exercise

You now know a lot about GPIO and have an example of how you might build an LED driver. Obviously, there is a lot of work underneath the API and just to set up the processor. If you are going to be an embedded systems engineer, the information presented here is what really matters. We now have a code base and API that completely abstracts the hardware from any application that needs to use the LEDs. If you have ever used Arduino or other hobbyist platforms, only now is the point at which you would normally first start writing code that makes use of libraries that might look similar to the design of this code.

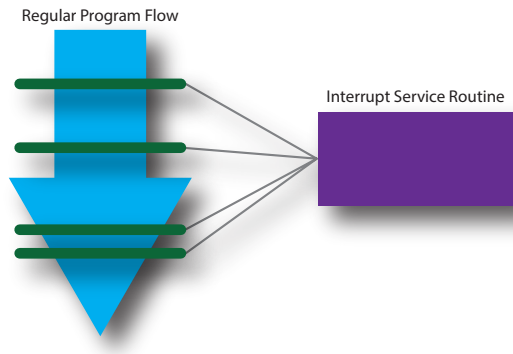
You should still have some fun at the application level! Try building a 4-bit counter using the discrete LEDs on the development board. A walk-through of this exercise is available online. What other fun LED projects can you build?

## Chapter 6 • Interrupts & Button Drivers

### 6.1 • Interrupts

Interrupts are a fundamental part of an embedded system. Interrupts are something you must be comfortable with if you are going to be serious about embedded design. The concept is identical to any interruption you experience in real life. Imagine you're watching a movie and the phone rings. Your "task" of watching a movie is interrupted by the phone. You pause your movie task in the middle of whatever it was doing, change to your "phone answering task" and run that until the conversation is over. Then you resume the movie. The movie has no idea it was paused and picks up exactly where it left off.

The idea in code is the same: an event that needs immediate attention triggers the processor to execute a special function called an interrupt service routine (ISR). To do this, the processor must stop what it is doing, save any important context information about what it was working on, and jump to the ISR. When the ISR is finished, the processor restores the saved context and continues running the code where it was interrupted in the first place as if nothing ever happened.



**Figure 6-1** An interrupt occurring 4 times during program executions

The interrupt capability of a processor is much like any other peripheral on the microcontroller. It has bits and bytes for configuration and status and will provide signals to the MCU about events that occur. Solid understanding of the interrupt system on a processor is absolutely critical to creating successful designs. The embedded engineer can define an entire system driven by interrupts and optimize system performance and power consumption by using interrupts. You could literally have a main program that is just a Sleep function call and let interrupts and ISRs do everything.

```
while (1)
{
    Sleep();
}
```

Though each processor core will have a unique set of rules that define how the interrupt hardware operates, the fundamentals apply across all platforms. It's even easier in the ARM Cortex-M world because the main interrupt functionality is part of the core, thus once you learn it the knowledge applies to any Cortex-M processor. Once you get past the syntax and understand the critical aspects of the interrupt engine, then you are free to build your system.



As you work through the chapters in the book you will see that interrupts are a fundamental part of the EiE operating system. They have been carefully designed as part of the overall system to work seamlessly with the main super loop so there are very few dependencies between the main code and the interrupt handlers. By the time you reach the end of this book, the system will be full of regular tasks that run in the main loop, and nearly a dozen interrupt handlers.

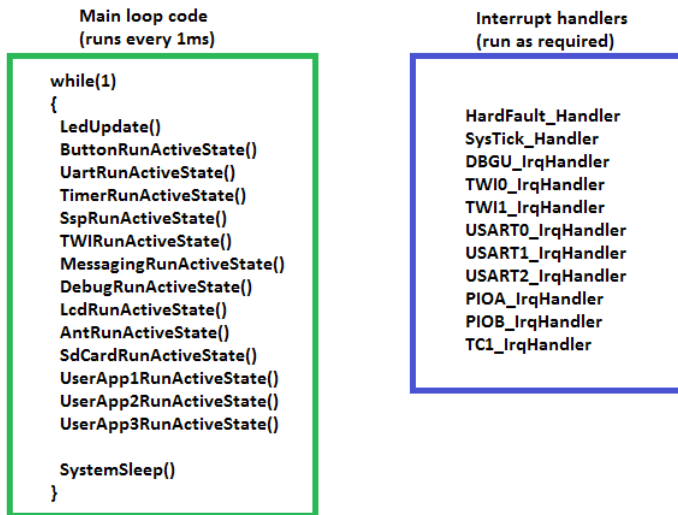


Figure 6-2 Main loop and interrupt handlers

Many interrupts occur for communication peripherals which allow the system to efficiently handle high-speed communications with other hardware while not disturbing the 1ms loop time. Even if the processor is sleeping, an interrupt can wake it up to do what it needs to do such as receive a byte of data, and then the processor will go back to sleep. The SysTick interrupt will be used to wake up the processor for the next 1ms loop, just like an alarm clock waking a person up in the morning to start another day. The tick interrupt will replace the call to kill\_x\_cycles and automatically adjust for the dynamic amount of time that each task will take.

## 6.2 • Interrupts on the SAM3U2

Every microcontroller and microprocessor will have at least a few interrupt sources that provide control to the firmware system. Not only do interrupts allow signals to be prioritized to ensure that critical events are handled in a timely fashion, but they also allow access to unique capabilities of the processor and its peripherals.

When we say, “interrupt source” we are referring to simple binary inputs that the processor can detect, stop what it is currently working on, address the signal source, and then resume what it was doing. In the last chapter, we alluded to interrupts from the PIO controller like a button press. Interrupts can also be enabled from timers when they expire, from communication peripherals that let you know when new data has arrived, and almost every peripheral on the microcontroller.

For ARM processors, Interrupts are a subset of Exceptions. Sometimes these two terms are taken as the same thing which is not totally correct. Technically speaking, an interrupt is a signal external to the core and one that you can decide to worry about or not. This is referred to as “maskable.” Exceptions are generated internally by the core when

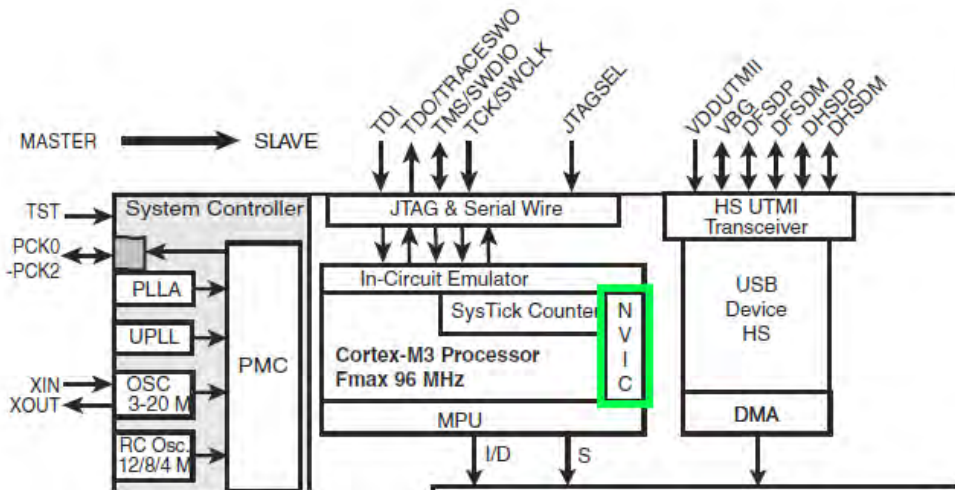
something has really broken like a division by zero or an invalid memory access. The processor must do something in this case, and it doesn't make sense to try and turn them off. They are referred to as "non-maskable."

The key concepts surrounding interrupts on any processor are quite general, but they will be discussed in the context of the Cortex-M3. Make sure you understand the following points.

### 6.2.1 • Interrupts depend on hardware

There is some sort of interrupt control hardware associated with interrupts. It may be with the core, a peripheral in the microcontroller, or a combination of both. For Cortex-M3 processors, the main interrupt hardware is called the “Nested Vector Interrupt Controller” (NVIC). Though the NVIC is the same across Cortex-M vendors, the microcontroller peripherals are hooked up in different ways so you still have to learn a little bit with every processor. You will learn a lot about the NVIC in this chapter.

The NVIC block can be seen in the SAM3Ux main block diagram.



**Figure 6-3 SAM3Ux main block diagram showing NVIC**

### 6.2.2 • Interrupts need to be configured by firmware

An interrupt source must be configured and enabled in order to provide an interrupt signal that the interrupt controller will recognize. If the interrupt is not enabled, the interrupt signal (called an interrupt 'flag') will still be asserted with the event, but the signal will be ignored by the interrupt controller and will thus not cause the processor to do anything. This interrupt flag can be polled instead so you can use the interrupt hardware without actually interrupting the processor – occasionally this is useful.

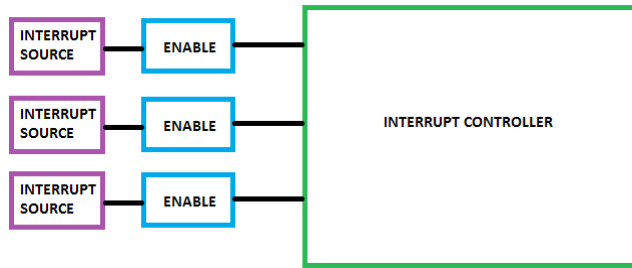


Figure 6-4 Interrupt sources into the controller

### 6.2.3 • Interrupts can be enabled and disabled globally

There is usually some sort of global interrupt enable (GIE) bit that must also be set to allow any interrupts to be active. If the GIE is disabled, the individual interrupt signal will still be present and an interrupt will be triggered if the GIE is later enabled as long as the interrupt flag has not been cleared. If there is no GIE, then there will be individual interrupt bits to enable the various sources.

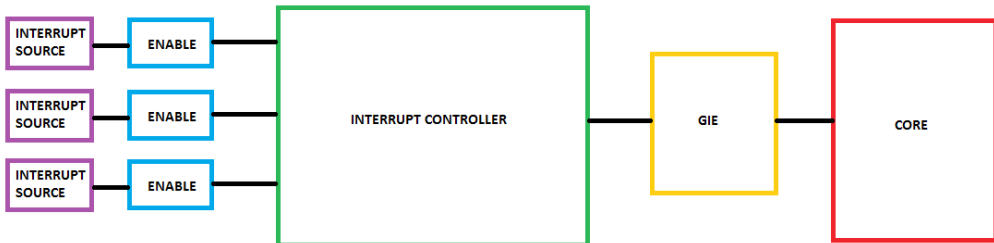


Figure 6-5 Global Interrupt Enable (GIE) diagram

For Cortex-M3, Global control of interrupts from C is handled by intrinsic functions provided by `core_cm3.h`:

```

_enable_irq();
_disable_irq();
_enable_fault_irq();
_disable_fault_irq();
    
```

The first two functions most reflect the action of the GIE bit described above because they globally control all the interrupt sources that can be enabled and disabled. The second pair of functions controls the entire exception hardware. These would not typically be used as turning off the exceptions would have bizarre results if the processor experienced a fault that would normally be handled by an exception. Both are enabled on reset, but none of the regular interrupt sources are enabled automatically.

### 6.2.4 • An interrupt forces the processor to run an Interrupt Service Routine

When an interrupt occurs, a function called an Interrupt Service Routine (ISR) is executed. The ISR that is selected to run is based on the source of the interrupt. Every interrupt source that is enabled will have its own unique ISR. The initial selection of the ISR is done in hardware using the Vector Table. ISRs are not “called” by a line of code like a regular function call.

For Cortex-M3 processors, an ISR function definition looks no different than any other function definition. An ISR cannot take function parameters and never returns a value. A compiler directive that explicitly identifies a function as an ISR is NOT needed for a Cortex-M processor. This is a big difference from many microcontrollers including the earlier ARM7 and ARM9 cores before the Cortex-M was released. This is because the compiler does not need to know what functions are ISRs as it does not have to add any additional instructions around ISRs to save or restore the system context when the interrupt occurs. It's still necessary to save context, but all of it is automatically done in hardware.

### 6.2.5 • Interrupts have priorities

Interrupts often have priorities so one will execute before the other. Many processors have multiple priority levels and allow a lower priority interrupt to be interrupted by a higher priority interrupt. This is called nesting. The NVIC on Cortex-M3 processors has 16 priority levels of interrupts that may be assigned to different peripherals as required. There are additional exception priorities that are set in hardware for system-type interrupts. For Cortex, the lower the number, the higher the priority. Figure 6-6 lists the 16 processor exceptions (1 through 0xF) along with their names, supported cores and defined priority levels.


Exception	Vector	Core	Priority
1	Reset	M0/M4/M3/M7	-3
2	NMI	M0/M4/M3/M7	-2
3	HardFault	M0/M3/M4/M7	-1
4	MemManageFault	M3/M4/M7	User
5	BusFault	M3/M4/M7	User
6	UsageFault	M3/M4/M7	User
7-A	Reserved		
B	SVCall	M0/M3/M4/M7	User
C	Debug Monitor	M3/M4/M7	User
D	Reserved		
E	PendSV	M0/MM3/M4/M7	User
F	SysTick	M0(optional)/M3/M4/M7	User
10-...	IRQ0, IRQ1, ... (Vendor)	M0: 0x10-0x47 M4/M7: IRQ0-239	User

Figure 6-6 Cortex-M exceptions

Active interrupts of the same hardware priority will not interrupt each other. The later one must wait and will execute immediately after the earlier one finishes. If an ISR is running and a higher priority interrupt occurs, then the lower priority ISR will be paused while the higher priority ISR executes. There may be several interrupt signals within each peripheral. If more than one interrupt source triggers the overall interrupt, then the ISR firmware can decide which to service first.

### 6.2.6 • Interrupts can (and will) occur anytime, anywhere

An interrupt can occur at any time and in any place in the code as long as the required interrupt enable bits are set. This means the program must be able to survive being interrupted at any point. If there are places where the program simply cannot survive if it is interrupted, then interrupts must be disabled during that part of the code. This can be done globally, or just the specific interrupts that could be a problem may be disabled temporarily.

 This is probably the most significant cause of impossible-to-find bugs in a system because it might be very difficult to repeat the problem. Customers in the field tend to be very good at doing it, but by the time the device gets back to the lab, everything will be working properly. The difficulty of the problem is compounded because many programmers do not realize how many instructions a line of C code may involve. Interrupts occur at the instruction level, not the high-level programming language level. If you ever think that you don't have to disable interrupts because you're just writing one line of C code, go back and read the Assembly chapter to properly learn how a processor works.

### 6.2.7 • Interrupts require context preservation

When an interrupt occurs, the program “context” is saved so that the core resources can be used to execute the ISR code without destroying values in those locations that may be overwritten during the ISR. All the context saving and restoring is done in hardware. When the processor interrupts, it vectors (moves the program counter) to the ISR and automatically pushes an “exception stack frame” on to the stack with no instruction cycle overhead. The exception frame is simply a group of registers to save that are in a particular order. These values include:

- R0-R3, and R12
- Return address
- PSR and LR

The stack frame is the minimum required context saving for interrupts. By saving this on the stack, multiple interrupts can occur where each context is saved and restored as the interrupt is serviced.

In some cases, the programmer may need to save additional context in the ISR which can be done at the start of the ISR as soon as the processor vectors there. Also at the same time, a special instruction is written to the link register which will properly trigger popping the stack frame back to restore the pre-exception context. This replaces the more conventional method of inserting some type of “return from interrupt” instruction in code.

Some processors do not automatically save registers and leave it up to the programmer and/or the compiler to decide what needs to be saved. If you use other resources that are shared in the main code and ISR, then those resources must be saved as well. For example, a scratch register “temp” that is used all the time for temporary storage. If the ISR uses “temp” as well, then it must be saved first and restored on exit, so the original value is preserved. Not likely a concern for a 32-bit processor with lots of resources, but it could be relevant to resource-limited MCU.

If you have global variables that are accessed in interrupt service routines, you need to be very careful on how these are managed. These should be declared “volatile” since the compiler cannot properly determine when they will be accessed and may optimize them incorrectly. Don't forget that peripheral registers are global.

When the ISR is complete, the system context must be restored. For Cortex cores, the context that is automatically saved is also automatically restored. The ISR returns and re-enables interrupts that had been disabled (equal or lower priority). If another interrupt is waiting to run, it will start executing immediately.

#### 6.2.8 • Interrupts set flags that need to be cleared

Triggering an interrupt happens when hardware sets a flag bit somewhere. This is just a logic one in a register. If the flag is not cleared in the ISR by the time it exits, then the processor will immediately vector to the ISR again. Every ISR must take the appropriate action to clear the interrupt flag that was set in the peripheral to get it there. Very often the first time you start working with a new interrupt source, you will find that you get stuck in the ISR because it turns out you are not clearing the flag properly. For Cortex-M, there is a peripheral flag and the NVIC flag.



Clearing an interrupt flag might be done by explicitly writing to a register, or by just reading or writing some other register. If it's done by reading a register, be aware that if you are looking at a register in the debugger, every time the debugger is halted it effectively reads the registers that are set up for display in the debug window. A debugger read has the same effect as a regular instruction reading the register, which can cause unexpected behavior.

#### 6.2.9 • ISRs should be short and fast

An ISR should execute as quickly as possible, especially if it can occur very frequently. If an interrupt is used to trigger a larger piece of code, a good strategy is to use the ISR to set a flag, and then poll this flag during normal operation of the task scheduler or main loop.

### 6.3 • Interrupt User Guide Resources

How are interrupts configured and used on a processor? First, we need to locate the details in the user guide. The information about using interrupts on the SAM3U2 is oddly scattered around several places. Some of the information contains a lot of details that you don't need to worry about. The information you do need is summarized here. We will reference the March 2015 version of the guide (Atmel-6430G-ATARM-SAM3U-Series-Datasheet\_31-Mar-15) as that is the latest release at the time of writing. Previous or subsequent versions will have similar sections but different section numbers.

#### Section 12.5 Exception model

This section gives detailed information about the structure of the exception system. This is really low-level information and not necessary to know to use interrupts with the processor. However, it becomes necessary as you architect more complicated systems especially if you are adding an operating system, memory protection, and coding the most robust system possible.

The Cortex interrupt capability is widely considered exceptional (no pun intended) amongst processors in the industry. It is remarkably efficient and if you have worked with other processors and understand interrupts, this section should show that the Cortex interrupt capability possess some very impressive features. A few sub-sections are highlighted below.

### Section 12.5.4 Vector table

The first low-level detail you should know about is the Vector Table which was introduced in the Assembly chapter. The term “Vector” is synonymous with “Address” or, more accurately, “moving to an address.” This is a specific set of addresses in flash memory that is programmed with the addresses of all the exception handlers. It is accessed by hardware when the corresponding exception happens and the addresses stored in these locations are loaded into the program counter – essentially the same thing that happens with a function call but this is initiated by hardware. You can see the implementation of the Vector Table in the `board_cstartup_iar.c` file. The list in firmware is bottom to top, while the vector table in the user guide is shown top to bottom.

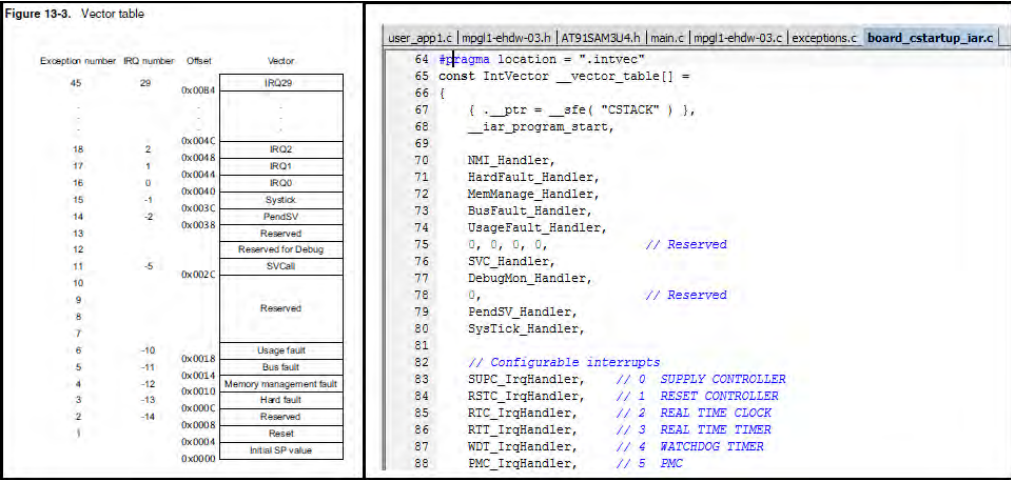


Figure 6-7 Vector table in user guide (left) and firmware (right)

### Section 12.5.5 Exception priorities

The concept of priorities with respect to interrupts in a microcontroller is no different than your own priorities. You always do the highest priority activity first, and the processor does the same. If something of higher priority comes along, you stop what you’re working on, do the higher priority task, then resume what you were doing before. Lower priorities are ignored until you are finished higher priority work. Think of cooking dinner when suddenly your child falls and hits their head. You drop everything because your child is a very high priority. But then if the stove catches fire, that’s even higher priority. Put out the fire, help the child feel better, then resume cooking dinner.

Exception number <sup>(1)</sup>	IRQ number <sup>(1)</sup>	Exception type	Priority	Vector address or offset <sup>(2)</sup>	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Hard fault	-1	0x0000000C	-
4	-12	Memory management fault	Configurable <sup>(3)</sup>	0x00000010	Synchronous
5	-11	Bus fault	Configurable <sup>(3)</sup>	0x00000014	Synchronous when precise, asynchronous when imprecise
6	-10	Usage fault	Configurable <sup>(3)</sup>	0x00000018	Synchronous
7-10	-	-	-	Reserved	-
11	-5	SVCall	Configurable <sup>(3)</sup>	0x0000002C	Synchronous
12-13	-	-	-	Reserved	-
14	-2	PendSV	Configurable <sup>(3)</sup>	0x00000038	Asynchronous
15	-1	SysTick	Configurable <sup>(3)</sup>	0x0000003C	Asynchronous
16 and above	0 and above <sup>(4)</sup>	Interrupt (IRQ)	Configurable <sup>(5)</sup>	0x00000040 and above <sup>(6)</sup>	Asynchronous

Figure 6-8 Properties of the different exception types

#### Section 12.5.7.5 Exception entry

This is when an interrupt first happens. In many processors, the compiler must pad any ISR with certain context handling instructions, but all required context saving in a Cortex processor is handled in hardware so there is actually no instruction overhead and almost no latency (that's amazing, by the way). This is also why interrupt service routines coded in the system do not require a `#pragma` to identify them as interrupt service routines which usually triggers the compiler to add the correct context saving code.

#### Section 12.9 Intrinsic functions



CMSIS provides standard headers for functions that run assembly code where there is no equivalent C code. These are called Intrinsic functions and there are several specific to interrupts. This section has the complete list that you should look at.

#### Section 12.19 Nested Vectored Interrupt Controller

The details of the NVIC are described here. The description is confusing because of the way the registers are organized. All of the registers are listed with their bit definitions here. In most cases, we will use intrinsic functions provided in the CMSIS header file to manage these bits and that's essentially all you need to know. A nice summary table of all CMSIS NVIC control functions is provided.



CMSIS interrupt control function	Description
void NVIC_SetPriorityGrouping(uint32_t priority_grouping)	Set the priority grouping
void NVIC_EnableIRQ(IRQn_t IRQn)	Enable IRQn
void NVIC_DisableIRQ(IRQn_t IRQn)	Disable IRQn
uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)	Return true if IRQn is pending
void NVIC_SetPendingIRQ (IRQn_t IRQn)	Set IRQn pending
void NVIC_ClearPendingIRQ (IRQn_t IRQn)	Clear IRQn pending status
uint32_t NVIC_GetActive (IRQn_t IRQn)	Return the IRQ number of the active interrupt
void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)	Set priority for IRQn
uint32_t NVIC_GetPriority (IRQn_t IRQn)	Read priority of IRQn
void NVIC_SystemReset (void)	Reset the system

Figure 6-9 CMSIS interrupt control functions

### Section 11.1 Peripheral identifiers

The very useful table to determine peripheral numbers that are required for various interrupt configuration registers. Print this table because you can never find it when you need it.

Instance ID	Instance Name	NVIC Interrupt	PMC Clock Control	Instance Description
0	SUPC	X		Supply Controller
1	RSTC	X		Reset Controller
2	RTC	X		Real-time Clock
3	RTT	X		Real-time Timer
4	WDT	X		Watchdog Timer
5	PMC	X		Power Management Controller
6	EEFC0	X		Enhanced Embedded Flash Controller 0
7	EEFC1	X		Enhanced Embedded Flash Controller 1
8	UART	X	X	Universal Asynchronous Receiver Transmitter
9	SMC	X	X	Static Memory Controller
10	PIOA	X	X	Parallel I/O Controller A
11	PIOB	X	X	Parallel I/O Controller B
12	PIOC	X	X	Parallel I/O Controller C
13	USART0	X	X	Universal Synchronous Asynchronous Receiver Transmitter 0
14	USART1	X	X	Universal Synchronous Asynchronous Receiver Transmitter 1
15	USART2	X	X	Universal Synchronous Asynchronous Receiver Transmitter 2
16	USART3	X	X	Universal Synchronous Asynchronous Receiver Transmitter 3
17	HSMCI	X	X	High Speed Multimedia Card Interface
18	TWI0	X	X	Two-Wire Interface 0
19	TWI1	X	X	Two-Wire Interface 1
20	SPI	X	X	Serial Peripheral Interface
21	SSC	X	X	Synchronous Serial Controller
22	TC0	X	X	Timer Counter 0
23	TC1	X	X	Timer Counter 1
24	TC2	X	X	Timer Counter 2
25	PWM	X	X	Pulse Width Modulation Controller
26	ADC12B	X	X	12-bit Analog-to-Digital Converter
27	ADC	X	X	10-bit Analog-to-Digital Converter
28	DMAC	X	X	DMA Controller
29	UDPHS	X	X	USB High Speed Device Port

Figure 6-10 Peripheral identifiers

## 6.4 • Interrupts and C

The C programming language itself does not specifically care about interrupts, nor does it have any interrupt-specific special features. However, each compiler will have various mechanisms to allow the developer to code interrupt functionality for the target processor. The compiler usually needs to know something about interrupt service routines, and it is also important to consider the stack, heap, and RAM. Remember that an interrupt can happen ANYWHERE in your code unless you have interrupts disabled for some or all your program. Understanding the interrupt system on the microcontroller you are using is extremely important.

To use interrupts for the Cortex-M3, we need to use a combination of the following elements. Some of this information has been covered previously but is repeated for emphasis.

### 6.4.1 • Vector Table

The Vector table is the address map of all the supported exceptions. Each of these locations is essentially physically attached to the interrupt signaling hardware. The interrupt signals cause one of the addresses stored in the vector table to be loaded to the program counter. This is how an interrupt service routine is “called” by hardware.



Open the IAR project for this chapter and find the vector table in `board_startup_iar.c`. Also add `interrupts.c`, `buttons.c`, and `buttons.h` from `\firmware_common\drivers` to the “Drivers” sub-groups in the project. `#include` the button header file in `configuration.h`.

Every line in the vector table corresponds to a “Handler” function. In `exceptions.c` there is a definition for all the handlers referenced in the vector table. There must be code somewhere in the system for these functions or else the code won’t compile since they are referenced in the vector table. The default handlers are all just `while(1)` infinite loops because they are not intended to run. If you purposely enable an interrupt, it is assumed you will write a proper ISR. If the code seems to be frozen and halting the debugger shows that it is in one of these loops, then you know that the associated interrupt was enabled, occurred, and you need to do something about that.

The “WEAK” keyword is supposed to allow the developer to re-define the function somewhere else and the compiler will automatically use that definition instead of the WEAK function. This sometimes doesn’t work for us, so you might see several of the default handlers removed explicitly using `#if 0 / #endif`.

### 6.4.2 • Priorities

Interrupt priority levels are decided by the designer and assigned in firmware. The needs of the system will determine what priority you use and there are some general rules and best practices. Timing and/or time-critical interrupts should be a high priority. Interrupts that occur very frequently might be lower in priority unless they deal with high-speed incoming data where failing to detect an incoming byte could result in missing the message. As the system gets more complicated, you might end up calculating how quickly different interrupts can occur to help decide priorities.

There are 30 peripherals in the SAM3U2 which are connected to the NVIC. The NVIC has 16 priority levels available, so if you use all the peripherals with interrupts then some will have to share priority levels. There are no limits to how many peripherals can be assigned to a level. Priorities are assigned by writing priority bits to the register location that corresponds to the peripheral you’re setting. 8 bits are used but only the top 4 are used to select an interrupt priority 0-15.

For the SAM3U2, there are eight 32-bit priority registers in total, with each one used to configure 4 peripheral priorities. These registers are arranged as an array, `NVIC_IPR[0...7]`. This is where the peripheral number table is required so you know which interrupt corresponds to the groups of 8 bits in the array. It's a bit confusing, so below shows how the first 6 peripherals map into the first two `NVIC_IPR` array elements:

```
NVIC_IPR[0] bits 0-7: Peripheral 0 SUPC Supply Controller
NVIC_IPR[0] bits 8-15: Peripheral 1 RSTC Reset Controller
NVIC_IPR[0] bits 16-23: Peripheral 2 RTC Real-time Clock
NVIC_IPR[0] bits 24-31: Peripheral 3 RTT Real-time Timer
NVIC_IPR[1] bits 0-7: Peripheral 4 WDT Watchdog Timer
NVIC_IPR[1] bits 8-15: Peripheral 5 PMC Power Management Controller
```

For any Cortex processor, the vendor will likely provide a low-level function library to access the priority levels, but we prefer to define `INIT` values in a header file and load all the `IPR` registers directly just like we do for other peripheral setup registers. This keeps all the priority information in one place and can easily be changed while seeing what the other registers are set at.



Open `interrupts.h` and find the priority level initializations that have been set already. Unused peripherals have been assigned as the lowest priority even though that shouldn't matter as they will never be enabled. The priority levels for interrupts that will be used have all been assigned and those choices will be described in their respective chapter discussions.

#### 6.4.3 • Enabling and Disabling Peripheral Interrupt Sources

The maskable (selectable) interrupts are always disabled when the processor first starts up. As peripherals are configured, their interrupt behavior is set up locally to the peripheral using the peripheral's user interface registers. When the local registers are ready, the final step is to enable that peripheral interrupt source in the NVIC by calling an NVIC function. A peripheral source may be individually enabled or disabled at any time in the program.

CMSIS provides two functions for this purpose:

- `NVIC_EnableIRQ( (IRQn_Type)x)` to enable specific interrupts
- `NVIC_DisableIRQ( (IRQn_Type)x)` to disable specific interrupts

The argument of the function is the peripheral number. An enum is provided from the vendor which captures the peripheral numbers for the SAM3U2. For EiE this is stored in the Type Definitions section of `interrupts.h`. The typedef is called `IRQn_Type` and the values are the peripheral ID numbers captured in a format that will work with the CMSIS calls.

Peripherals that offer interrupt capability have a single, general interrupt line into the NVIC, even though the peripheral itself might have many sources of interrupts within the peripheral that can be individually enabled or disabled. All of those get ORed together into the single line for the NVIC. The interrupt handler for the peripheral can parse the individual peripheral flags once the NVIC processes the general flag and vectors the processor to the handler. The general interrupt flag is typically automatically cleared once all the peripheral flags are clear. If an NVIC flag needs to be explicitly cleared, there is another CMSIS-provided function to do this.

- `NVIC_ClearPendingIRQ( (IRQn_Type)x)` clears an interrupt flag

If a peripheral flag is still set and the interrupt is enabled, the NVIC flag will remain set even if you try to clear it. That means the processor will keep vectoring to that interrupt

handler. The handler must ensure that all interrupts are processed properly and clear their peripheral flags so the NVIC flag will clear.



As with any peripheral, the initialization code should set it up and ensure that it is in the known state we want. The function `InterruptSetup()` will be used to load the priorities and make sure that all interrupt sources are turned off when the processor is booting. This is done in `interrupts.c` and is called immediately after `GpioSetup()` in `main()`'s initialization code. The header for `InterruptSetup()` is shown.

```

/*!-----
@fn void InterruptSetup(void)

@brief Disables and clears all NVIC interrupts and sets up interrupt priorities.

Requires:
- IRQn_Type enum is the sequentially ordered interrupt values starting at 0

Promises:
- Interrupt priorities are set
- All NVIC interrupts are disabled and all pending flags are cleared
*/

```



The `IRQn_Type` enum is sequential so again we'll abuse enums and use a loop to run through all the interrupts to disable and clear them. A second loop is used to load in the INIT values which need to be in an array for this to work.

```

void InterruptSetup(void)
{
    const u32 au32PriorityConfig[PRIORITY_REGISTERS] = {IPR0_INIT, IPR1_INIT,
        IPR2_INIT, IPR3_INIT, IPR4_INIT, IPR5_INIT, IPR6_INIT, IPR7_INIT};

    /* Disable all interrupts and ensure pending bits are clear */
    for(u8 i = 0; i < SAM3U2_INTERRUPT_SOURCES; i++)
    {
        NVIC_DisableIRQ( (IRQn_Type)i );
        NVIC_ClearPendingIRQ( (IRQn_Type) i);
    }

    /* Set interrupt priorities */
    for(u8 i = 0; i < PRIORITY_REGISTERS; i++)
    {
        ((u32*)(AT91C_BASE_NVIC->NVIC_IPR))[i] = au32PriorityConfig[i];
    }
} /* end InterruptSetup(void) */

```



Add a call to `InterruptSetup()` immediately after `GpioSetup()` in `main`. Build the code and start the debugger. Activate the "Nested Vectored Interrupt Controller" group in a Registers window. Put a breakpoint on `InterruptSetup()` and run the code.

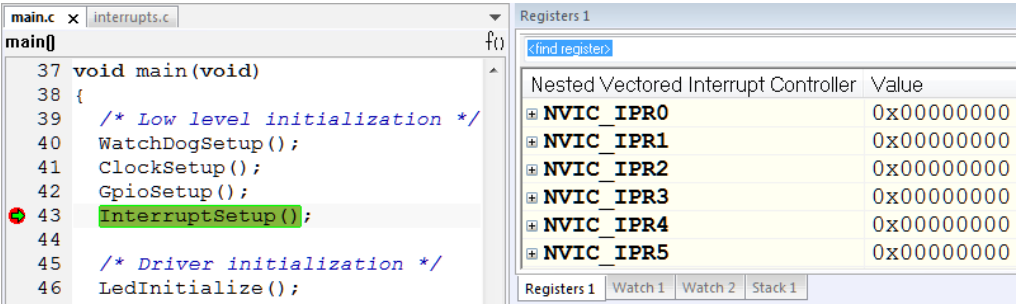



Figure 6-11 Examining NVIC initialization

 Step into `InterruptSetup()` which will start at the array initialization. Open a Stack view and the Disassembly window and look at what happens when you step over the array init. A runtime library `memcpy` function is called and all the values get loaded to the stack. Though this has nothing to do with configuring interrupts, it is a good example to demonstrate what happens behind the C code. It also is a good demonstration of the stack usage and debug windows that are available.

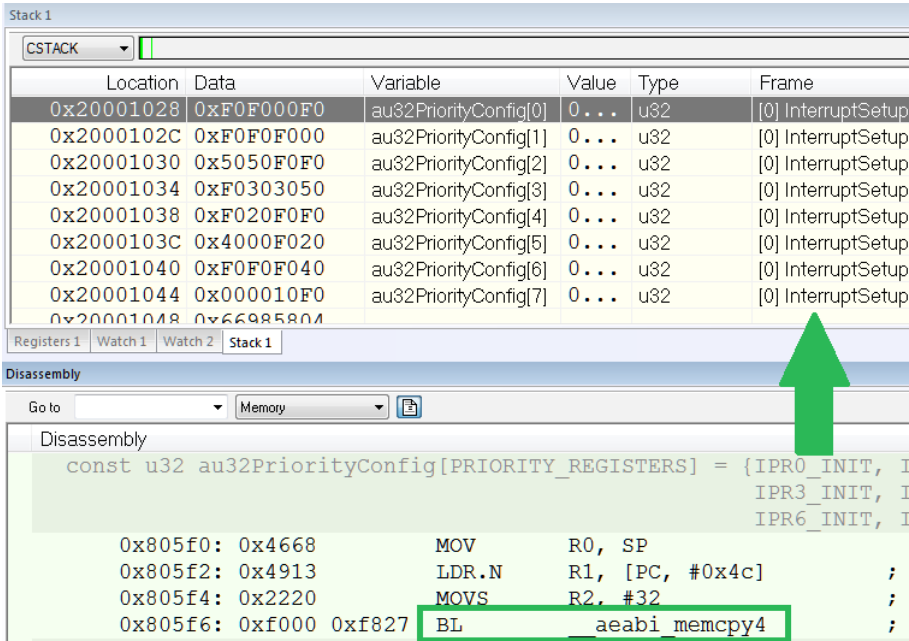



Figure 6-12 Stack usage and debug window

 Stop the debugger and change the `au32PriorityConfig` initialization to “static const.” Rebuild the code and see the difference. In both cases, the INIT values are still stored in flash memory somewhere as all tables of values must be. But with “static” the compiler knows that it can reference the table directly in flash instead of copying it to the stack. This saves a few lines of assembler code and reduces stack usage.

Finish stepping through the code and observe the `NVIC_IPRx` values updating.

Nested Vectored Interrupt Controller	Value
± NVIC_IPR0	0xF0F000F0
± NVIC_IPR1	0xF0F0F000
± NVIC_IPR2	0x5050F0F0
± NVIC_IPR3	0xF0303050
± NVIC_IPR4	0xF020F0F0
± NVIC_IPR5	0x4000F020
± NVIC_IPR6	0xF0F0F040
± NVIC_IPR7	0x000010F0

Figure 6-13 Nested Vectored Interrupt Controller values

## 6.5 • Peripheral Interrupts

There are many different interrupt sources on the SAM3Ux family of processors. Each can be enabled individually, and you can have many different interrupt sources active. Every peripheral that is connected to the NVIC has details about how various sources within the peripheral are enabled, disabled, set, cleared, and multiplexed into the single peripheral interrupt signal to the NVIC. No matter what interrupt source you are working with, the same rules apply to the behavior of that source.

### 6.5.1 • GPIO Interrupts

Interrupts that come from sources external to the microcontroller are often a key element in low-power design. The processor can be in sleep mode and use power only for the interrupt hardware until it is woken up by an external event interrupt. Power consumption can be sub-microamp in deep sleep on some processors.

Buttons are a simple example. Using a button interrupt source saves you polling for the button signal and allows user input to be sensed instantly. It also essentially eliminates the chance of missing an input signal since the signal is detected in hardware and not cleared until the processor sees it and acknowledges it even if the original signal source has long-since vanished. Input sources can be extended beyond buttons to things like encoders, where fast signals can be read from other circuits. Other devices in the system can provide wake-up or status signals as well on GPIOs.

### 6.5.2 • Timer / Counter Interrupts

Timer peripherals can be interrupt sources and cause interrupts for various reasons. The most common use is interrupting when the timer reaches a certain value or overflows. Though that might not sound like a big deal, timer interrupts enable some of the most powerful features of an embedded system. For example, a timer interrupt can provide a system tick that will maintain a very accurate representation of elapsed time in the system regardless of how busy the system is.

In addition to keeping great time, timer interrupts facilitate low power operation of embedded systems. Many embedded systems spend most of their time doing nothing except sleeping. Often a timer peripheral can run independently from the core and by itself consume much less power. A timer interrupt is going to cure our variable 1ms system tick problem and allow us to sleep the microcontroller.

### 6.5.3 • Communication Peripheral Interrupts

Communication peripherals provide interrupt signals to do things like indicating when receive buffers are full or transmit buffers are empty. They may also indicate various

errors or warnings that the system can respond to and make sure data flows properly. Each peripheral has its own interrupts, so multiple communications peripherals can be connected to the NVIC. This can provide a nice interrupt-driven system that can manage many data streams in and out of the processor without any code in the main loop. Interrupt-driven communications allow high data rates to be maintained without ever missing a byte. In these cases, it becomes important to ensure ISRs execute very quickly and are properly prioritized to handle the highest speed data.

#### **6.5.4 • Other Peripheral Interrupts**

There are many other interrupt sources available on a microcontroller. Though timers, communications, and GPIO typically make up most interrupt sources, there are always a few unique interrupts available that can be helpful. Whenever you begin working with a microcontroller that you have not had experience with, reading and understanding the available interrupts and how the processor manages those interrupts is extremely important. Developing a good interrupt strategy as part of the overall system design is a critical planning step before starting to code.

### **6.6 • Button Driver Overview and Setup**

This chapter focuses on the GPIO interrupts from the buttons, so we need to start talking about the button driver to determine how interrupts will help us. The goal of the button driver is to provide easy control over the momentary switches on the board. To ensure fast response without missing any signals, button inputs will trigger interrupts to detect press and release. The driver will provide much more than just indicating if a button is pressed or not. From experience, we know some additional features would be helpful.

#### **6.6.1 • Debouncing**

Debouncing is required for all mechanical switches to eliminate multiple signals that happen very quickly just as the button contacts make or break their connection. Some buttons are very good and rarely bounce, but others can turn on and off several times when they are pressed and released before the signal is stable. Interrupt-driven button drivers would register all these transitions if not for debouncing.

The worst-case contact bounce period is typically specified on the button datasheet so you know how long to debounce for. If in doubt, 5ms is a good rule of thumb. The scope trace shown is from releasing a button on the EiE development board. There are a total of 4 signal edges over 300us.

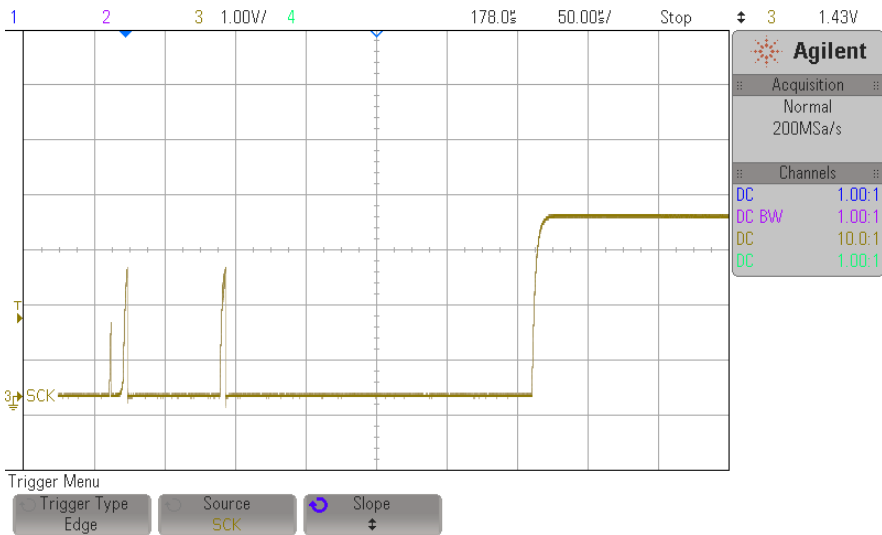


Figure 6-14 Contact bounce when a button is released

### 6.6.2 • Button history or edge detection

Very often a task is interested in the single transition of a button state. In other words, edge detection. Imagine a task running in the 1ms system loop. If the task checked if the button was currently pressed and acted in some way, then that action would be taken again every time the task ran since the task has no way of determining if the button was released and pressed again since last time it checked. Therefore, it would be up to the task to monitor the button to wait until the button was released and then pressed again. Add debouncing to this and it can be a lot of code to manage.

If you have a long state in your program or just a fast button press that might be missed by a polling function, it would be nice to be able to check if a button had been pressed since last time you checked it even if it is no longer pressed. Multiple tasks in the system may also need to access this kind of button information.

### 6.6.3 • Button held

An interrupt indicates when a button is first pressed, and after it is debounced a flag can be set to indicate the button is currently pressed or was pressed since last time it was checked. But what if you need to determine if the button is being held down? Long-press checking is very common especially when there are a limited number of inputs to a system. Long presses can be used to accept conditions, choose different options, or even access special functionality.

Timing out a button press takes resources and may be difficult across states of a task. If several tasks are waiting for buttons to be held, those duplicated resources are wasted. Having this information provided automatically is a great feature.

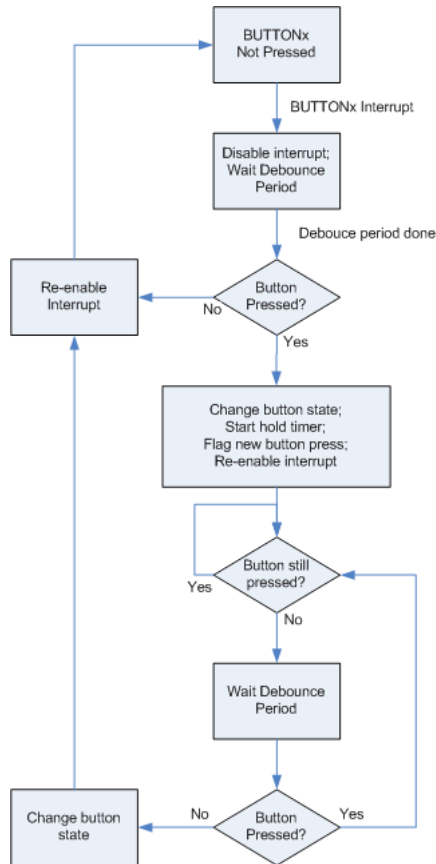
## 6.7 • Button Operation

We know all the services that we want to provide with the button application, now we must implement the firmware to do it. There are two main problems to solve:



1. Hardware connection to buttons (interrupt handlers and debouncing).
2. API functions to work with buttons that have been debounced and are either on or off.

The design will be a hybrid between interrupt-driven and a regular state machine. Below is a flowchart that shows how the button application will work for each button in the system to sense the signal interrupt and debounce the input.



**Figure 6-15 Button operation flowchart**

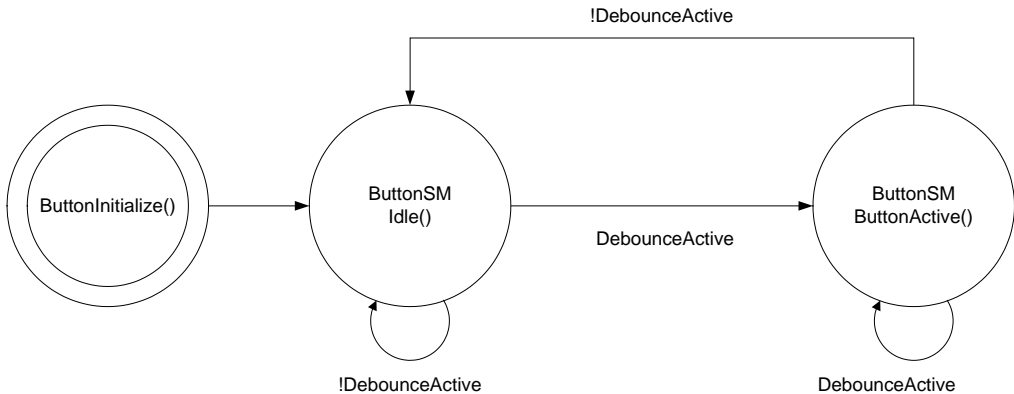
In words, the button system works like this:

1. Assume that buttons always start off. Even if a button is held when the system starts up, the application will work as soon as the button is released and pressed again. During this time, the button interrupts are enabled waiting to see the falling edge when a button press occurs.
2. As soon as a button is pressed and the interrupt occurs, the interrupt is disabled so bouncing does not cause multiple interrupts and the system time is captured for the particular button.
3. The button application waits for the debounce time to expire.
4. The state of the button is checked again by reading the PIO line associated

with the button. If it is still low after the debounce time, then we conclude that a valid button press has occurred. The button state is changed in memory so we know it is currently pressed, the current system time is logged so we know when the button press started, and the Boolean flag to indicate a new button press is set. If the button state is high, that means it was not a valid button press, so we go back to state 1. In both cases, the interrupt is re-enabled so we can detect the next transition.

5. If a pressed button is released, then a new button debounce period is started (release requires debouncing, too) and the process repeats.

When there are no buttons pressed, the button application code can execute quite quickly as all it needs to do is check the debounce flags. As more buttons get pressed, the application will need to do more work to monitor all the buttons, though it is still relatively efficient in the way it is coded. The state diagram is shown.



**Figure 6-16** Button state diagram

Each button will need to be tracked individually. If at least one button is active then the button task will be in the Active state. As soon as no buttons are debouncing, the SM can return to Idle. With multiple buttons doing the same thing, an array of data will be used so it can easily change for different platforms, and all code can index through with the same function.

### 6.7.1 • Button Typedefs

Before we do the interrupts, the bones of the button task need to be set up. The first part is almost identical to the LEDs. Type definitions for `PortOffsetType`, `GpioActiveType`, and `PinConfigurationType` are already in `typedefs.h` from the LED task. We just need a few new button-specific types.



Open `eief1-pcb-01.h` and add similar information that was done for the LEDs. Define an enum `ButtonNameType` where the list is `BUTTON0` through `BUTTON3`. There are four total buttons defined as `U8_TOTAL_BUTTONS`. For interrupt parsing, create two symbols `GPIOA_BUTTONS` and `GPIOB_BUTTONS` that are the button pin locations on each port ORed together. You will see why shortly.

```


/*!
@enum ButtonNameType
@brief Logical names for buttons in the system.

```


```
The order of the buttons in ButtonNameType must match the order of the definition
in G_asBspButtonConfigurations from eief1-pcb-01.c
*/
typedef enum {BUTTON0 = 0, BUTTON1, BUTTON2, BUTTON3} ButtonNameType;

#define U8_TOTAL_BUTTONS    (u8)4    /*!< Total number of Buttons in the system */

/*! All buttons on each port must be 0Red together here: set to 0 if no buttons */
#define GPIOA_BUTTONS      (u32)( PA_17_BUTTON0 )
#define GPIOB_BUTTONS      (u32)( PB_00_BUTTON1 | PB_01_BUTTON2 | PB_02_BUTTON3 )
```

 A global variable for the button configurations is required in eief1-pcb-01.c just like the LED definition.

```
/*! Button locations: order must correspond to ButtonNameType in the header file. */
const PinConfigurationType G_asBspButtonConfigurations[U8_TOTAL_BUTTONS] =
{
    {PA_17_BUTTON0, PORTA, ACTIVE_LOW},
    {PB_00_BUTTON1, PORTB, ACTIVE_LOW},
    {PB_01_BUTTON2, PORTB, ACTIVE_LOW},
    {PB_02_BUTTON3, PORTB, ACTIVE_LOW},
};
```

 A button could be pressed or released at any given time, and we want to be able to time how long a button is held for. First, create an enum called ButtonStateType with RELEASED and PRESSED as options in buttons.h. Create a status structure type that will hold the information necessary for each button: the current state, a new state, timestamp for the button hold, and a flag to indicate if the press is “new” to the system.

```
/*!
@enum ButtonStateType
@brief Self-documenting button state type */
typedef enum {RELEASED, PRESSED} ButtonStateType;

/*!
@struct ButtonStatusType
@brief Required parameters for the task to track what each button is doing.
*/
typedef struct
{
    ButtonStateType eCurrentState; /*!< @brief Current state of the button */
    ButtonStateType eNewState;     /*!< @brief New state of the button */
    u32 u32TimeStamp;             /*!< @brief Time when button was pressed */
    bool bNewPressFlag;           /*!< @brief TRUE if press not acknowledged */
} ButtonStatusType;
```

Create a local global array of ButtonStatusType in buttons.c.



```
static ButtonStatusType Button_asStatus[U8_TOTAL_BUTTONS];
```

Setup the calls to ButtonInitialize() and ButtonRunActiveState() in main.c, and make sure buttons.h is included in configuration.h. Build the code to check for any input errors. Ignore warnings about unused variables.

## 6.8 • PIO Interrupts

PIO interrupts from buttons are a good place to demonstrate interrupts since they will only occur as the result of the physical act of pushing a button. Starting with an interrupt

signal that is much faster like the 1ms SysTick timer can be difficult because the interrupt happens continuously and very quickly.

The basic interrupt behavior of the button inputs can be defined without discussing the details of the button driver yet. All we need to know is that we want to configure the system to provide an interrupt whenever a button is pressed and released.



The interrupt handlers will be kept in `interrupts.c` because they are general to the PIO peripheral and could service PIO interrupts from any of the hardware attached to the PIO controllers. Since there are buttons on PIOA and PIOB, we will set up both handlers. Start by adding the PIOA handler function in `interrupts.c` using the following example. The only code so far is to clear the IRQ flag. This handler will be completed entirely and then can be copied for PIOB.

```

/*!-----
@fn ISR void PIOA_IrqHandler(void)

@brief Parses the PORTA GPIO interrupts and handles them appropriately.

Note that all PORTA GPIO interrupts are ORed and will trigger this handler,
so any expected interrupt that is enabled must be parsed out and handled.

Requires:
- The button IO bits match the interrupt flag locations

Promises:
- Buttons: sets the active button's debouncing flag, clears the interrupt
  and initializes the button's debounce timer.
*/
void PIOA_IrqHandler(void)
{

    /* Clear the PIOA pending flag and exit */
    NVIC_ClearPendingIRQ(IRQn_PIOA);

} /* end PIOA_IrqHandler() */

```



Interrupt configuration for the button signals will take place in `ButtonInitialize()` since only the button task needs to worry about these signals. First, initialize the `Button_asStatus` array.

```

/* Setup default data for all of the buttons in the system */
for(u8 i = 0; i < U8_TOTAL_BUTTONS; i++)
{
    Button_asStatus[i].bNewPressFlag = FALSE;
    Button_asStatus[i].eCurrentState = RELEASED;
    Button_asStatus[i].eNewState     = RELEASED;
    Button_asStatus[i].u32TimeStamp  = 0;
}

```




Enable the PIO peripheral interrupts in the PIO. They are not enabled yet in the NVIC. The two button masks `GPIOx_BUTTONS` can be written to the PIO controller's `PIO_IER` (interrupt enable) register. This will set just those bits. To make sure no interrupts are currently set, read the `PIO_ISR` registers which is how the PIO peripheral documentation indicates to clear the interrupt flags (remember that). For this operation, you can do anything that causes the processor to read the values in the registers and it does not need to really do anything. It can be tricky to do this without getting a compiler warning or having the code removed by an optimizer. The solution below works.

```

/* Enable PIO interrupts */
AT91C_BASE_PIOA->PIO_IER = GPIOA_BUTTONS;
AT91C_BASE_PIOB->PIO_IER = GPIOB_BUTTONS;

/* Dummy code to read the ISR registers and clear the flags */
u32Dummy = AT91C_BASE_PIOA->PIO_ISR;
u32Dummy |= AT91C_BASE_PIOB->PIO_ISR;

```

 Lastly, clear the NVIC PIO flags and enable the PIOA and PIOB interrupts. Set the Button task state pointer to the Idle state. Build the code and fix any errors.

```

/* Configure the NVIC to ensure the PIOA and PIOB interrupts are active */
NVIC_ClearPendingIRQ(IRQn_PIOA);
NVIC_ClearPendingIRQ(IRQn_PIOB);
NVIC_EnableIRQ(IRQn_PIOA);
NVIC_EnableIRQ(IRQn_PIOB);

/* Init complete: set function pointer and application flag */
Button_pfnStateMachine = ButtonSM_Idle;


```

An important problem to solve is how data will be linked between the interrupt handler and the button task while maintaining abstraction between the two. Interrupts can be a big problem and major bug contributor due to their unpredictable nature. One of the most common problems are variables that are accessed both by an ISR and another piece of code. The odds are quite high of regular code being in the process of writing to a variable when an interrupt occurs and accesses the same variable to change it. This can sometimes be fatal to an embedded system.

In the case of the button task, the interrupt service routine will trigger when a button is pressed. This is the general PIO interrupt. The ISR then needs to determine which button was pressed by looking for the pin-specific interrupt flags and then tell the button task that the button is debouncing. That means the ISR needs to know where the buttons are, and then have a way to report the information.

If we treat this in the Object-Oriented world, button-specific information should be private to the button task and public member functions would be made available to access any of the private data. A function would be required for the ISR to query if a particular bit belonged to the buttons. If that returned true, then another function could be called to flag the debouncing information. Alternatively, we could use globally accessible information that the ISR could read and write.

Both methods require the PIO ISRs to be written specifically for the system, but the OO approach provides more flexibility and is truer to the goal of this system to be as OO-esque as possible. For the longest time, the button task code used global variables, but for the final version of the code for this book, it was changed to the way it really should be. To make things slightly less complicated, the ISR will be allowed to use the GPIOA\_BUTTONS and GPIOB\_BUTTONS to mask out PIO interrupt flags that are not due to buttons. Then we only need a single function to queue the debounce information.

 To make this work, add two additional members to the ButtonStatusType definition. The first is a Boolean flag that will be set if a button's debounce time is active, and the second is the start time of that debounce period. Go to ButtonInitialize() and add initialization to the two new parameters.

```

typedef struct
{
    bool bDebounceActive;           /* TRUE by ISR if a button interrupt occurs */
    bool bNewPressFlag;            /* TRUE if the press has not been acknowledged */
    ButtonStateType eCurrentState; /* Current state of the button */

```

```

    ButtonStateType eNewState;          /* New state of the button */
    u32 u32DebounceTimeStart;          /* Time loaded by ISR when interrupt occurs */
    u32 u32TimeStamp;                  /* System time when the button was pressed */
}ButtonStatusType;

```

The order of the members in the typedef is reorganized for 4-byte alignment. We have not talked about byte-alignment, but data structures like structs should be built by members where the total bytes used is evenly divisible by 4 bytes. This gets complicated with enumerated types because the compiler will decide what native type they are. The enums and Booleans (which in C are just enums) should be single bytes, so four of them together are 4-byte aligned. Alignment is maintained even if the enums are 16 or 32 bits.



The function to update the debounce information from the ISR will be placed in the “protected” section of buttons.c since it’s not something that any regular task would use, but it is called externally. The ISR needs to indicate what bit position it is working with, and what port the pin is at. Write the function based on this header.

```

/*!-----
@fn void ButtonStartDebounce(u32 u32BitPosition_, PortOffsetType ePort_)

@brief Called only from ISR: sets the "debounce active" flag and debounce start time

Requires:
- Only the PIOA or PIOB ISR should call this function

@param u32BitPosition_ is a SINGLE bit for the button pin to start debouncing
@param ePort_ is the port on which the button is located

Promises:
- If the indicated button is found in G_asBspButtonConfigurations, then the
  corresponding interrupt is disabled and debounce information is set in Button_as-
  Status
*/

```



When implementing the code, we found it necessary to add a “NOBUTTON” entry to the ButtonNameType enum. This was placed at the end of the enum list so it would not interfere with the indexing functionality of the enum. The function code ends up being straight-forward. Using “break” in a for-loop is debatable. It adds extra code and in this case, there are not very many buttons, so the processor time is negligible. It was kept in, though.

```

void ButtonStartDebounce(u32 u32BitPosition_, PortOffsetType ePort_)
{
    ButtonNameType eButton = NOBUTTON;

    /* Parse through to find the button */
    for(u8 i = 0; i < U8_TOTAL_BUTTONS; i++)
    {
        if( (G_asBspButtonConfigurations[i].u32BitPosition == u32BitPosition_) &&
            (G_asBspButtonConfigurations[i].ePort == ePort_) )
        {
            eButton = (ButtonNameType)i;
            break;
        }
    }
}

/* If the button has been found, disable the interrupt and update debounce status */
if(eButton != NOBUTTON)

```

```

{
    AT91C_BASE_PIOA->PIO_IDR |= u32BitPosition_;
    Button_asStatus[(u8)eButton].bDebounceActive = TRUE;
    Button_asStatus[(u8)eButton].u32DebounceTimeStart = G_u32SystemTime1ms;
}

} /* end ButtonStartDebounce() */

```

The button ISR and the button task are now reasonably abstracted from each other. We have not directly addressed the possibility of issues arising from the ISR accessing `Button_asStatus` at the same time as the regular button code. Even though the member-access function is being used, there is still a potential conflict. In this case, though, the interrupt is turned off before the button task access, and it will not be turned on until the button task has finished writing to the shared variable. It is safe to say there is nothing to worry about here.

Write the ISR next. The key things that must be done:

- Mask out only the button PIO interrupts
- For each flag set, call `ButtonStartDebounce()`

The biggest trick is reading the PIO interrupt flags from `PIO_ISR`. As soon as this is done, the PIO flags are cleared as long as the source of the interrupt does not continue to set the flag. Since the button interrupts are edge-triggered, that should be the case.



The trickier part is that the flag values read from `PIO_ISR` needs to be used a few times during the ISR, so `PIO_ISR` needs to be saved in a different variable. You can use `GPIOA_BUTTONS` to mask out the flags of interest. Remember that halting the debugger when the PIO registers are open in a Register window constitutes reading the registers, so don't ever put a break at the start of the ISR before your code reads `PIO_ISR`. Checking the button flags can be done with a bit mask to look at each bit and call `ButtonStartDebounce` using the current bit mask if a set bit is found. Write the function and then compare your solution.

```

PIOA_IrqHandler()
109 void PIOA_IrqHandler(void)
110 {
111     u32 u32GPIOInterruptSources;
112     u32 u32ButtonInterrupts;
113     u32 u32CurrentButtonLocation;
114
115     /* Grab a snapshot of the current PORTA status flags (clears all flags) */
116     u32GPIOInterruptSources = AT91C_BASE_PIOA->PIO_ISR;
117
118     /****** DO NOT set a breakpoint before this line of the ISR because
119     will "read" PIO_ISR and clear the flags. *****/
120
121     /* Examine button interrupts */
122     u32ButtonInterrupts = u32GPIOInterruptSources & GPIOA_BUTTONS;
123

```

Figure 6-17 Button interrupt code



Build the code and start the debugger. Even though the rest of the button driver is not finished, the interrupt function and `ButtonStartDebounce()` can be verified. Put a breakpoint inside the PIOA ISR after reading the interrupt flags. Run the code. Wait for a few seconds to make sure the board finishes booting. During this time the ISR should not run. Then press `BUTTON0` which should trigger the interrupt and stop at the breakpoint.

Open a Registers window with the PIOA group and find the PIOA\_IMR and PIOA\_ISR registers. Open a watch window and look at u32GPIOInterruptSources and u32ButtonInterrupts. Make sure you view them in “hex” format. Single step the code so u32ButtonInterrupts gets loaded and then stop to understand everything that is going on.

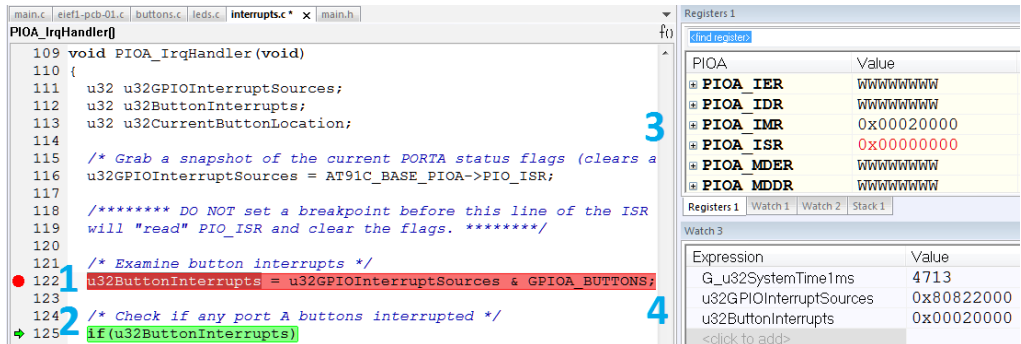


Figure 6-18 Stepping through the ButtonInterrupts code

The screenshot above is numbered for the following explanation:

1. Breakpoint is here. This is the earliest place a breakpoint can be set if the PIOA\_ISR register is in view. The u32GPIOInterruptSources variable has safely captured the interrupt flags. When the code is here, PIO\_ISR will not be cleared yet as there is one instruction latency before the register updates in the debug window.
2. u32ButtonInterrupts now holds just the flag bits for button interrupts. By this time, PIOA\_ISR will clear any button-related interrupts.
3. PIOA\_IMR shows the PORTA interrupts that are currently active. A bit in PIO\_ISR should have been set that lines up with the bit set in PIO\_IMR when the breakpoint was hit. Since the code has just been stepped, PIO\_ISR now clears.
4. u32ButtonInterrupts is ready to be processed. A single bit is set as expected. This bit lines up to the BUTTON0 pin position, PA\_17\_BUTTON0. You can see that quite a few PIOA interrupt flag bits are set in u32GPIOInterruptSources. These are for other peripherals that we may use later but other than the button interrupts, these interrupt flags are not enabled so they do not cause an interrupt.



The essence of interrupts is shown here. If you don't understand this, you must go back and review the notes to this point. It is assumed that this part of interrupt functionality is understood going forward.



Interrupt functionality generally does not work very well if you are single-stepping in code, especially for catching an interrupt on the way into the ISR. Once in an ISR, it is probably ok, and stepping out of an ISR usually works. There are ways to use breakpoints and other variables to debug interrupts if necessary. Remember this point if you are trying to debug interrupts and single-stepping is producing unexpected results.

Continue single-stepping through the code to see how it works including the bitmask to parse the interrupt flags. If you understand what is happening, set a breakpoint on the call to ButtonStartDebounce since the button is in bit 17 and single stepping all the way is tedious.



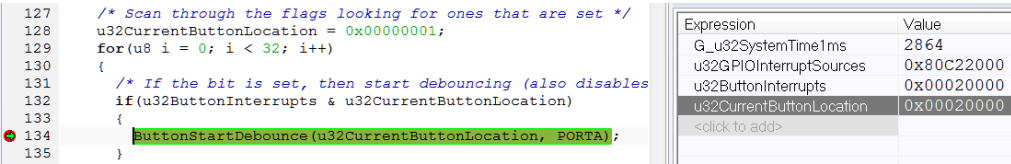


Figure 6-19 Debounce code highlighted

Step into ButtonStartDebounce and watch the array get updated. You can change to a new watch window and add eButton, G\_asBspButtonConfigurations, and Button\_asStatus. Expand the [0] position in the two arrays to see BUTTON0 information. In the figure below, the following should be visible once the button information has been updated. Be careful you don't step out of the function yet as that will clear the button local variable information as it goes out of scope.

1. The BUTTON0 interrupt has been disabled (bit is no longer set in PIOA\_IMR)
2. The button of interest has correctly been identified as BUTTON0.
3. G\_asBspButtonConfigurations has the expected BUTTON0 configuration information which was done at initialization.
4. Button\_asStatus has been updated so bDebounceActive is now TRUE and the debounce start time is the current system time.

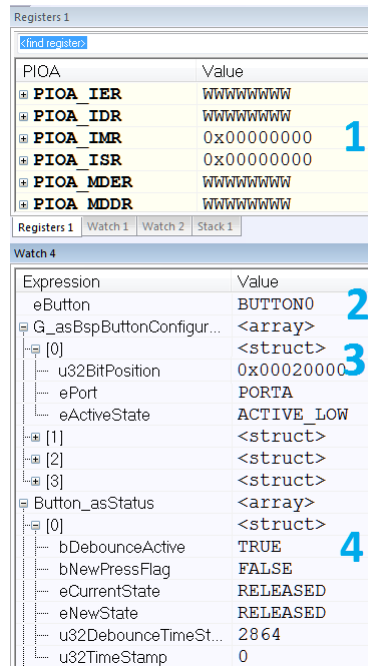


Figure 6-20 Button debugger information

It looks like the functions and ISR are working well. Notice that the ISR does not distinguish between a rising edge or falling edge interrupt. The state of the button will be handled by the button task.

The other buttons don't work yet because they are on PIOB. Copy and paste the whole PIOA ISR and update the information for PIOB. Always be extremely careful when copying

and pasting code as this is most often where updating comments can be missed that can lead to confusion. It is less likely but more harmful to miss updating a line of code. In most cases, the resulting bug would be found quickly, but there are instances where a bug like this could make it through to a release since the code may accidentally work all the time on the bench. It's a good idea to use "find and replace" and double check that any reference to PORTA data has been updated to PORTB in the copied code.



The rest of the button task can be written now. The job of the Idle state is to check if any of the buttons have started debouncing since the button task will manage that. All that's required is looping through `Button_asStatus` to check for debounce flags. If at least one flag is set, then the state machine should advance to the `ButtonSM_ButtonActive` state.

```

/*!-----
@fn static void ButtonSM_Idle(void)

@brief Look for at least one button to enter debouncing state
*/
static void ButtonSM_Idle(void)
{
    for(u8 i = 0; i < U8_TOTAL_BUTTONS; i++)
    {
        if(Button_asStatus[i].bDebounceActive)
        {
            Button_pfnStateMachine = ButtonSM_ButtonActive;
            break;
        }
    }
}

} /* end ButtonSM_Idle(void) */

```

The job of `ButtonSM_ButtonActive` is to monitor the buttons that are debouncing. Until the debounce period is over, the button level cannot be determined. All button statuses are initialized to `RELEASED`, but even if the button is held on power up, this will get sorted out when the button is released the first time. In this case, the button will not be registered as `PRESSED` when the code boots, so the button task cannot be used to look at buttons on power-up. If you need to detect if a button is being held when the board powers on, you need to use direct GPIO access.

Like all the button task functionality, `ButtonActive` will loop through and run the same code on each individual button. All it needs to do is check if the debounce time is up and then verify if the new state of the button is still different than the old state of the button. If the old state was `RELEASED` and the new state is `PRESSED`, then the "new press" flag can be set. Once a button changes to the `PRESSED` state, the system time is logged to the button to facilitate the button hold function.

On a quick side note: Very often a programmer is interested in the amount of time that has passed in the system. When a system tick is available, the process of checking how much time has passed is easily done by saving the tick time at the start of an event and then subtracting that from the current time. If you are looking for a certain amount of time to elapse, then the subtracted value is compared to the value in time of interest.

This happens a lot in a system. One of the things to watch out for is if the system tick timer overflows. While this doesn't happen very often, it could happen often enough that there could be big problems when it does. Always check for that overflow. If the total tick period was very, very short, then a different strategy would have to be used since the timer could overflow many times. If the longest period of time is always less than the tick period, it should be fine.

The 1ms counter overflows every 49 days, so any period that the EiE system needs to time is safe from multiple rollover problems but still should handle single rollovers. There is enough code there to justify writing a function for it, so a “utility” function called `IsTimeUp()` is added that can be used to safely check periods of time. This function is in a source file called `utilities.c` where other useful, general functions that don’t really belong anywhere else will be added.



Add `utilities.c` and `utilities.h` to the project. Put the `#include` in the “Common header files” area of `configuration.h`. Review the function description for `IsTimeUp()`. A pointer is used to pass the saved time variable as this function used to have the option of automatically resetting that variable.

We can resume coding the button task and will make use of `IsTimeUp()`. The `ButtonActive` state is what implements the lower part of the flowchart that was introduced at the start of the chapter. The code for each button will do the following:

- Check if the current button is debouncing.
- Check if the debounce period is over.
- Read `PIO_PDSR` register to get the new state of the button. The code must handle the `ACTIVE_LOW` and `ACTIVE_HIGH` states separately here.
- Check if the new state is different than the saved state and update the saved states accordingly.
- If the new state is `PRESSED`, then the new press flag must also be set.
- Re-enable the interrupt for the button which could be on `PIOA` or `PIOB`.



A lot of array indexing of `G_asBspButtonConfigurations` and `Button_asStatus` will be required. Write the outline of the code with a comment about each section.

```
static void ButtonSM_ButtonActive(void)
{
    /* Check for buttons that are debouncing */
    for(u8 i = 0; i < U8_TOTAL_BUTTONS; i++)
    {
        /* Check if the current button is debouncing */
        if( Button_asStatus[i].bDebounceActive )
        {
            /* Check if debounce period is over */
            if( IsTimeUp(&Button_asStatus[i].u32DebounceTimeStart, U32_DEBOUNCE_TIME) )
            {
                /* Active low */
                if(G_asBspButtonConfigurations[i].eActiveState == ACTIVE_LOW)
                {
                    /* Read PIO_PDSR to get the actual input signal (new button state) */
                }
                /* Active high */
            }
            else
            {
                /* Read PIO_PDSR to get the actual input signal (new button state) */
            }
        }

        /* Update if the button state has changed */
        if( Button_asStatus[i].eNewState != Button_asStatus[i].eCurrentState )
        {
            /* If the new state is PRESSED, update the new press flag */
            if(Button_asStatus[i].eCurrentState == PRESSED)
            {
                {
            }
        }
    }
}
```

```

    /* Regardless of a good press or not, clear the debounce active flag
       and re-enable the interrupts */

    } /* end if( IsTimeUp...) */
  } /* end if(Button_asStatus[i].bDebounceActive) */
} /* end for (u8 i = 0; i < U8_TOTAL_BUTTONS; i++) */

} /* end ButtonSM_ButtonActive() */

```



Code to return to the Idle state is also needed. An easy way to update the state machine is to assume that the code will go back to Idle, but change that if a button is found to be debouncing still. Perhaps the trickiest part is accessing the PIO\_PDSR and PIO\_IER register bits generically in a loop since buttons can be on PIOA or PIOB. To make this work, define two variables `pu32PortAddress` and `pu32InterruptAddress` and use them to save the address of these registers based on the current button. With the state machine update and the address loading, the top part of the state code looks like this:

```

static void ButtonSM_ButtonActive(void)
{
    u32 *pu32PortAddress;
    u32 *pu32InterruptAddress;

    /* Start by resetting back to Idle in case no buttons are active */
    Button_pfnStateMachine = ButtonSM_Idle;

    /* Check for buttons that are debouncing */
    for(u8 i = 0; i < U8_TOTAL_BUTTONS; i++)
    {
        /* Load address offsets for the current button */
        pu32PortAddress = (u32*)&(AT91C_BASE_PIOA->PIO_PDSR) +
            G_asBspButtonConfigurations[i].ePort);
        pu32InterruptAddress = (u32*)&(AT91C_BASE_PIOA->PIO_IER) +
            G_asBspButtonConfigurations[i].ePort);

        /* Check if the current button is debouncing */
        if( Button_asStatus[i].bDebounceActive )
        {
            /* Still have an active button */
            Button_pfnStateMachine = ButtonSM_ButtonActive;

```



The code inside the `/* Active low */` branch should read PDSR and check if the corresponding button bit in `G_asBspButtonConfigurations` is set. Since this is the `ACTIVE_LOW` case, the PDSR bit will be 0 if the button is pressed. The value read from PDSR can be inverted with the `~` operator and then bit-wise ANDed with the button's bit position. The `ACTIVE_HIGH` case is identical but without the inversion of PDSR.

```

    /* Active low */
    if(G_asBspButtonConfigurations[i].eActiveState == ACTIVE_LOW)
    {
        /* Read PIO_PDSR to get the actual input signal (new button state) */
        if( ~( *pu32PortAddress ) & G_asBspButtonConfigurations[i].u32BitPosition )
        {
            Button_asStatus[i].eNewState = PRESSED;
        }
        else
        {
            Button_asStatus[i].eNewState = RELEASED;
        }
    }
}

```



The button state change code simply compares the saved new and current states (by this time, “current” is old). Once the update is made, the new press information is updated if the button has changed to PRESSED.

```
if( Button_asStatus[i].eNewState != Button_asStatus[i].eCurrentState )
{
    Button_asStatus[i].eCurrentState = Button_asStatus[i].eNewState;

    /* If the new state is PRESSED, update the new press flag */
    if(Button_asStatus[i].eCurrentState == PRESSED)
    {
        Button_asStatus[i].bNewPressFlag = TRUE;
        Button_asStatus[i].u32TimeStamp = G_u32SystemTime1ms;
    }
}
```



The last step is to clear the debounce active flag for the button and re-enable the interrupt. Dereference pu32InterruptAddress like any regular pointer and write the bit location.

```
/* Regardless of a good press or not, clear the debounce active flag and re-enable
the interrupts */
Button_asStatus[i].bDebounceActive = FALSE;
*pu32InterruptAddress = G_asBspButtonConfigurations[i].u32BitPosition;
```



Build the code. There should be a warning about the unused ButtonSM\_Error state. The button task does not require an error state, so use a #if 0 and #endif directive to remove it. That way if it is ever needed it is easy to put back. Take some time to test the operation of the code to make sure everything works. If there are parts you don’t understand, step through them carefully. Always try to predict what you think will happen before stepping through each line of code. If you predict incorrectly, figure out why.

## 6.9 • Button API

Now that the button application is working, API functions can be provided so that other applications can use the button app. All the hard work is already done. These functions simply provide easy access to the data that is already present within the button task. We want a few simple functions:

1. **IsButtonPressed():** returns TRUE if a button is currently pressed (reads Button\_asStatus.eCurrentState)
2. **WasButtonPressed():** returns TRUE if a button was pressed since last time it was checked, even if the button is no longer pressed at that moment (reads Button\_asStatus.bNewPressFlag). If so, we need a function to acknowledge that we have received the information and clear the flag. ButtonAcknowledge() will do this.
3. **IsButtonHeld()** – returns TRUE if a button has been held for a certain amount of time. This function must check if the button is currently pressed, and then access Button\_asStatus.u32DebounceTimeStart and compare it with the current system time. Remember we have the IsTimeUp() function to do this easily.



The full code for IsButtonPressed() is shown below. Try to write the function yourself and compare with the solution shown. If you need a hint, read the function header. It is up to you to implement the other three functions. They will require just a few lines of code

each. Only use existing parameters of the button data structures and return the correct Boolean value where applicable. You do not need to write any code outside of these functions, but you do need to understand how button information is stored.

```

/*!-----
@fn bool IsButtonPressed(ButtonNameType eButton_)

@brief Determine if a particular button is currently pressed at the moment in time
when the function is called.

Requires:
- Button_asStatus[eButton_] is a valid index

@param eButton_ is a valid button

Promises:
- Returns TRUE if Button_asStatus[eButton_].eCurrentState is PRESSED
- Otherwise returns FALSE
*/
bool IsButtonPressed(ButtonNameType eButton_)
{
    if( Button_asStatus[(u8)eButton_].eCurrentState == PRESSED)
    {
        return(TRUE);
    }
    else
    {
        return(FALSE);
    }
}

} /* end IsButtonPressed() */

```



Make sure your code builds without errors. In `user_app1.c`, write some quick functions to test the new button API functions.

```

static void UserApp1SM_Idle(void)
{
    if(IsButtonPressed(BUTTON0))
    {
        LedOn(WHITE);
    }
    else
    {
        LedOff(WHITE);
    }

    if(WasButtonPressed(BUTTON1))
    {
        ButtonAcknowledge(BUTTON1);
        LedBlink(PURPLE, LED_2HZ);
    }

    if(IsButtonHeld(BUTTON2, 2000))
    {
        LedOn(BLUE);
    }
    else
    {
        LedOff(BLUE);
    }
}

} /* end UserApp1SM_Idle() */

```

Again, we have shown that adding seemingly simple functionality requires a lot of thought, design, and careful implementation. The button task now gives easy access to the button hardware and all board-specific details have been abstracted for the end user.

#### **6.10 • Chapter Exercise**

The online chapter exercise is all about using the button (and LED) APIs to solidify your understanding of the functionality that is available on the development board at this point. These functions will be used extensively so you must be comfortable with what has been covered up to now. Understanding the difference between `IsButtonPressed` and `WasButtonPressed` is fundamental, especially when it comes to conditioning code execution based on user input from the buttons.

## Chapter 7 • Sleep, System Tick and Timer Peripheral

“It’s all about timing.” You have likely heard that expression before, though maybe not in the context of electronics. A huge part of embedded system design has to do with timing. We have already looked at clocking the processor and the EiE development board is equipped with both a high-speed and low-speed crystal source. Basic clock configuration has been completed, and we have a 1ms system tick albeit with a brute-force, instruction-consuming loop. Timing concepts in embedded systems extend into power optimization, using sleep modes, and taking advantage of timer peripherals for convenience and/or further system optimization.

Production devices require further attention to how they will regularly operate to optimize system performance in the intended application. It may be that a product is meant to run full speed all the time to maximize throughput and processing. However, microcontroller-based embedded systems tend to be more involved in applications where low-power is a concern. This is especially true in battery-powered devices, including portable consumer electronics, commercial and industrial equipment, and IoT applications. Even if a product is not required to be ultra-low power, as engineers we are ethically responsible for minimizing the energy footprint of our devices. Considering that embedded system products are very likely mass-produced, being careful to minimize power consumption in designs can have a real impact.

Development boards are not typically optimized for power consumption because of their intended general application, but that doesn’t mean you can’t get average current down to negligible levels. With two processors on board, the EiE development board has numerous options for optimizing power consumption while still maintaining what appears to be full system performance. The base firmware we are developing is not focused on power consumption, but we will improve it a lot with the code in this chapter and discuss some options that could be implemented to further reduce power usage.

### 7.1 • Sleep

Just as sleep is an essential part of a person’s everyday life, sleep is also very important for an embedded system. Unlike humans, a processor can run forever without sleeping, but that doesn’t mean it should. The duty cycle of a system – the time where the processor is actively doing something versus the total time it is on – is often very low. Even when all the tasks are completed in the EiE firmware system if those tasks are in their idle state the system duty cycle is less than 5%. Imagine if 95% of the time the system could be almost completely shut down consuming no power. In the simplest of examples, a battery-powered device that lasted one day if the processor was running continuously, could be made to run for almost a month.

Every processor has at least one low power or sleep mode where parts of the microcontroller can be turned off so they stop consuming power. This includes the core. Other parts of the micro can continue running and wake-up the sleeping parts when it is time for them to do some work.

The sleep mode used for the EiE system is not a very low power mode compared to some of the modes available. Sleep mode turns off the core but not the main oscillator that generates the MCK clock, and all the enabled peripherals keep clocking. This saves almost 20mA but the microcontroller is still consuming about 10mA – but still significant savings. The table shows the average currents expected from each of the available low power modes.



**Table 7-1 SAM3U2 Low Power Modes**

3.3V, 25°C, 48MHz	
Mode	Current
Backup	3 uA
Wait	27 uA
Sleep	12 mA
Active	31 mA

To wake up from sleep, some sort of hardware signal must tell the core to run again. For EiE, the “System Tick” (SysTick) clock peripheral that is built into the core will be used as this wake-up signal. The system tick timer runs completely independently from the rest of the code and hardware in the system, so it never needs to be adjusted and is never affected by other lines of code that the processor has to execute. The timer also continues to run while the core is sleeping as long as the MCK clock is running.

Right now, the firmware system is using explicit instructions to time out the 1ms period of the loop so any additional code that must run in the loop extends that time. Using the timer ensures that exactly 1ms is clocked out. If the processor is not busy, it doesn’t have to waste time in `kill_x_cycles` – it just goes to sleep. Regardless of what the system is doing, the System Tick timer generates an interrupt every 1ms that stops the processor from doing whatever it was doing and increments `G_u32SystemTime1ms`. If the processor was sleeping, it is woken up to run another system loop. This should always be the case if all the tasks are following the 1ms execution rule. However, even if the code in the system is violating the 1ms rule, `u32SystemTime1ms` is never disrupted because the timer is a separate peripheral with an exception that will always run at exactly 1ms intervals.

To get down to microamp current consumption, the main oscillator must be turned off. In that case, the wake-up strategy would need to be different since the SysTick would not be clocked. The firmware design would also need to change to ensure that any signal not generated by a system task would wake up the processor. This could include a button press, incoming message from the ANT radio, or insertion of an SD card. In the absence of these signals, the processor could continue sleeping, or the low power internal or external oscillator could be used to provide an alarm clock signal.

A typical application could use Wait or Backup mode and wake up every second to update the LCD. The LCD itself does not draw a lot of current when it is just displaying as long as the backlight is off. If an external signal woke up the processor, the processor could switch to Sleep or Active mode to take care of the LCD data transfer, then change back to a better low power mode. We have designed devices like this that appear to provide continuous LCD operation for three years on just a tiny lithium battery.

## 7.2 • System Tick Configuration

The concept of a system tick is widely used in embedded systems and is the main timing source for these systems. Having a base time source allows all the tasks to synchronize themselves or be managed by an operating system using a common reference. This does not have to be “real time” like a day, month and year, though that is possible and many systems (like GPS) utilize this. Systems with GPS receivers can be synchronized to within microseconds even if they’re on opposite sides of the world.

The EiE system time was introduced in the last chapter. Two counters, `G_u32SystemTime1ms` and `G_u32SystemTime1s` are incremented and together can time

out over 100 years of operation with 1ms accuracy. Currently, `G_u32SystemTime1ms` is incremented by explicit calls to `SystemSleep`, but that will be updated to run from the SysTick timer ISR.

All Cortex-M3 cores come with the SysTick timer, which assists in portability across different processors and different vendors since the timer hardware remains the same. The SysTick exception is one priority above the maskable interrupts, so it will always be prioritized above any peripheral interrupt configured.

The SysTick configuration registers are introduced in the ARM Cortex-M3 section of the user guide.

Address	Name	Type	Required privilege	Reset value
0xE000E010	CTRL	RW	Privileged	0x00000004
0xE000E014	LOAD	RW	Privileged	0x00000000
0xE000E018	VAL	RW	Privileged	0x00000000
0xE000E01C	CALIB	RO	Privileged	0x0002904 <sup>(1)</sup>

Figure 7-1 SysTick configuration registers

There are only four registers, so it is easy to determine how to use the SysTick timer. What is not obvious is where these registers are located. We're not sure what happened with the processor documentation here, but the register names in `AT91SAM3U4.h` do not match the user guide at all. All the SysTick registers are in the `AT91PS_NVIC` struct with the names as shown in the following descriptions.

**SysTick Control and Status Register (CTRL, NVIC\_STICKCSR):** the main configuration register for the SysTick timer. The clock source is selected which is always MCK, but an 8x prescaler can be applied to extend the available timer period. Both the SysTick interrupt and the timer itself are enabled in this register.

**SysTick Reload Value Register (LOAD, NVIC\_STICKRVR):** the 24-bit value that the timer counts down from. This value is reloaded automatically once the timer reaches 0. Note the comment about how to use the reload value correctly. To continually count the given period, you must reduce the number of ticks in the reload register by 1. This would account for the extra tick where the timer is 0.

**SysTick Current Value Register (VAL, NVIC\_STICKCVR):** the current timer count. Writing 0 here resets the timer and clears the counter flag.

**SysTick Calibration Register (CALIB, NVIC\_STICKCALVR):** a fixed value that, with a clock source of 10.5MHz, will yield a 1ms SysTick period. This could be used to check the accuracy of the hardware if that was necessary, though we are not quite sure the exact usage. EiE will not use this register.

### 7.2.1 • Tick Time and CTRL INIT value


The slowest SysTick period that can be generated with the EiE clock configuration of 48MHz would be to use the 8x prescaler with the maximum 24-bit value. A prescaler is an additional counter in the signal chain. It is just a 3-bit counter, so it provides one output clock tick for every 8 clocks on the input.

$48\text{MHz} / 8 = 6\text{ MHz}$

$1/f = T = 16.7\mu\text{s}$  per tick

$224 = 16777216 \times 16.7\mu\text{s} = 2.7962\text{ seconds}$


If for some reason a longer period was needed, a different timer would have to be used. The SysTick timer is designed for an operating system tick, which is typically on the order of milliseconds. Since the SysTick timer does not run when the MCK clock source is turned off, it cannot be used for long, deep sleep.

 To generate the reload value, the clock constants in `eief1-pcb-01.h` can be used. The default 8x prescaler will be used, so first `#define` the value `SYSTICK_DIVIDER` to 8 just after the other clock constants. Then document the calculation for the 1ms tick.

```
#define SYSTICK_DIVIDER      (u32)8    /*!< System tick scaling value */

/* To get 1 ms tick, need SYSTICK_COUNT to be 0.001 * SysTick Clock.
Should be 6000 for 48MHz MCK. */
#define U32_SYSTICK_COUNT    (u32)(0.001 * (MCK / SYSTICK_DIVIDER) )
```

Decimal values like 0.001 can be used for preprocessor definitions because the preprocessor does this calculation when the code is compiled. Only the resulting number is put into the code and used directly by the assembler. This is true for any `#define` where the arguments can be predetermined without having to run any code to know the values used. This is cast to an integer so even if the result was not a whole number it would be truncated to an integer. In this case, the result should be 6000 without any rounding. When using preprocessor calculations, make sure any rounding that takes place is reasonable.


 The INIT value for the SysTick control register should ensure the prescaler is on and can enable both the timer and the timer interrupt. Code this value below the PMC INIT values in `eief1-pcb-01.h`.

```
#define SYSTICK_CTRL_INIT (u32)0x00000003
/* Bit Set Description
   31:20 Reserved

   19 [0] Reserved
   18 [0] "
   17 [0] "
   16 [0] Countflag (read only)

   15-04 [0] Reserved

   03 [0] "
   02 [0] Clock source is CPU clock / 8
   01 [1] System tick interrupt on
   00 [1] System tick is enabled
*/
```

 A function call to `SysTickSetup()` should be added to `main` after `InterruptSetup()`. Define the function in `eief1-pcb-01.c` and make sure it resets the two global timer values since they will be connected to the SysTick interrupt in a moment. Also, load `NVIC_STICKRVR` using `SYSTICK_COUNT - 1` per the register definition. Zero `STICKCVR` to reset the timer, and then load `STICKCSR` with `SYSTICK_CTRL_INIT` to enable the timer and the interrupt. Build the code to check for errors but don't run it yet.

```

/*!-----
@fn void SysTickSetup(void)

@brief Initializes the 1ms and 1s System Ticks off the core timer.

Requires:
- NVIC is setup and SysTick handler is installed

Promises:
- Both global system timers are reset and the SysTick core timer is configured
*/
void SysTickSetup(void)
{
    G_u32SystemTime1ms = 0;
    G_u32SystemTime1s = 0;

    /* Load the SysTick Counter Value */
    AT91C_BASE_NVIC->NVIC_STICKRVR = U32_SYSTICK_COUNT - 1;
    AT91C_BASE_NVIC->NVIC_STICKCVR = (0x00);
    AT91C_BASE_NVIC->NVIC_STICKCSR = SYSTICK_CTRL_INIT;
} /* end SysTickSetup() */

```

There is a default SysTickHandler function in exceptions.c that would run and trap the code when the SysTick interrupt happens. Write a new SysTick\_Handler function in interrupts.c that will take over as the SysTick handler function. You already wrote the code for this function! Go to SystemSleep() in eieft1-pcb-01.c and cut the part of the function that clears the \_SYSTEM\_SLEEPING flag and updates the timers. Paste this in to SysTick\_Handler().

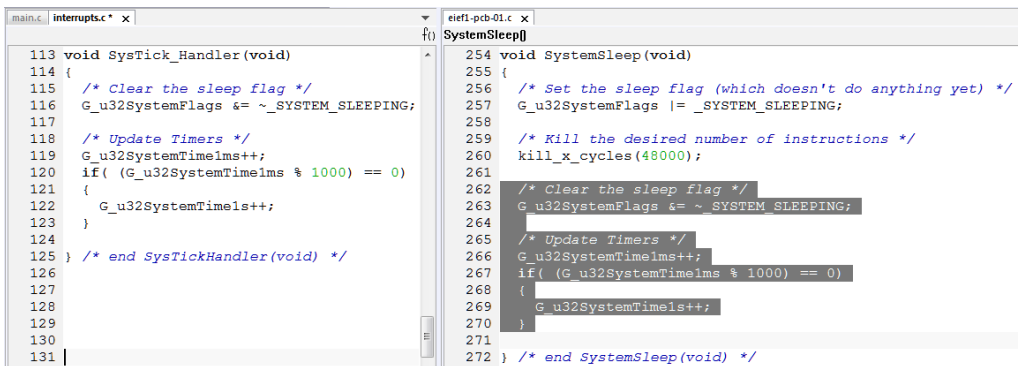


Figure 7-2 Paste code into SysTick\_Handler()



In SystemSleep(), delete the call to kill\_x\_cycles and replace it by the code shown below that configures the sleep mode for the processor and then puts the processor to sleep with a call to an intrinsic function. Delete the “extern kill\_x\_cycles” at the top of the file, too, since this file will no longer call that function. Keep kill\_x\_cycles.s in the project, though, because it will be used elsewhere.

```

void SystemSleep(void)
{
    /* Set the system control register for Sleep (but not Deep Sleep) */
    AT91C_BASE_PMC->PMC_FSMR &= ~AT91C_PMC_LPM;
    AT91C_BASE_NVIC->NVIC_SCR &= ~AT91C_NVIC_SLEEPDEEP;
}

```

```

/* Set the sleep flag (cleared only in SysTick ISR) */
G_u32SystemFlags |= _SYSTEM_SLEEPING;

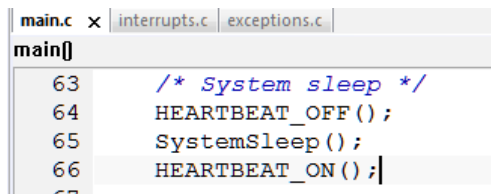
/* Now enter the selected LPM */
while(G_u32SystemFlags & _SYSTEM_SLEEPING)
{
    __WFI();
}

```

Make sure `SystemSleep()` matches what is shown here. `PMC_FSMR` and `NVIC_SCR` are used to tell the processor what low power mode should be used and what to do after it wakes up (e.g. stay awake, or go back to sleep). The `__WFI()` call is an intrinsic function to invoke the WFI instruction that stands for “Wait for interrupt.” This is what causes the power mode change to sleep mode. This is all described in the “Power management” section of the user guide, but it can be confusing the first few times you work with it. Getting the processor to sleep and wake up exactly how you want can take some trial and error. The debugger is not much help, since debug action either wakes up the processor or doesn’t work at all during sleep.

*Debugging probes and emulators rarely function properly when it comes to demonstrating sleep mode. Debuggers rely on the processor core to clock out information, so if the core clock is sleeping, the debugger cannot get any information. Power consumption is also completely different, so if you are designing ultra-low power devices and are trying to measure operating current, you need to do so by running and measuring a real, programmed device without any debug probe connected.*

The reason the `__WFI` call is in a loop is so other interrupts can wake up the processor, but the system will go back to sleep once those ISRs run if the SysTick interrupt has not yet occurred to clear `_SYSTEM_SLEEPING`. This ensures 1ms loop timing even when other interrupts occur and must be serviced. It makes more sense now to have the loop here, so the loop that was in main around `SystemSleep()` can be removed.



```

main.c x interrupts.c exceptions.c
main()
63     /* System sleep */
64     HEARTBEAT_OFF();
65     SystemSleep();
66     HEARTBEAT_ON();
67

```

Figure 7-3 `SystemSleep()` function

Since the SysTick timer is counting independently of what is happening in the main super loop, the 1ms period is always exact. Even if the main loop takes too long, the SysTick timer will increment `G_u32SystemTime1ms` every 1ms since SysTick is above the highest priority interrupt level. In the debug chapter we will add a check to ensure that only 1ms elapses each loop and can print a warning if too much time has gone by.

Add a simple test to make sure the loop timing is working. In `UserApp1_Initialize()`, add one line of code:

```

void UserApp1Initialize(void)
{
    LedBlink(BLUE, LED_1HZ);
}

```

Build the code and start the debugger but do not run yet. Open the “Breakpoints” debug window and set a breakpoint on the first line of code in `SysTick_Handler`. You will see the

breakpoint description in the Breakpoints window. Uncheck the box in the breakpoint window. This disables the breakpoint but does not remove it from its location in the code.

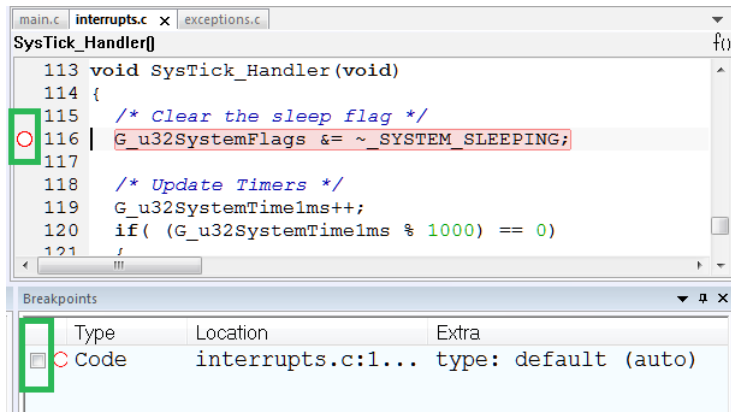


Figure 7-4 Breakpoint description window

Disabling breakpoints like this is a very powerful debugging feature. Sometimes you will have dozens of breakpoints in a system to find problems. It might take hours if not days to set up a sequence of breakpoints. Often you will work backward from when a problem occurs and will add more and more breakpoints to find the spot where things are still working before the system breaks. It takes a lot of time to set those up, and you do not always want them all turned on as you step through and make changes. If you just disable them, you can re-enable them easily and jump to that point of the code just by double-clicking the breakpoint. You can also re-enable them while the code is running in real time inside the debugger.



Add `G_u32SystemTime1ms` and `G_u32SystemTime1s` to a watch window. Run the code and wait for the blue LED to start blinking so you know the main loop is running. Re-enable the SysTick breakpoint in the Breakpoints window and the code should stop right away. The value of `G_u32SystemTime1ms` should be updated. In the example below, 24.7 seconds have elapsed. You can see that `G_u32SystemTime1s` will always be `G_u32SystemTime1ms` divided by 1000.

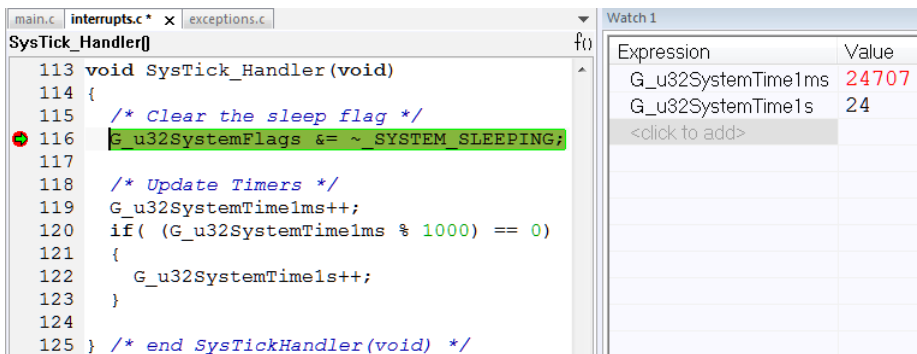
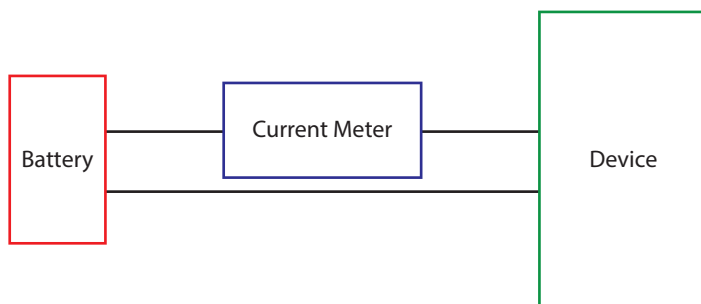


Figure 7-5 Re-enable the SysTick breakpoint

Our system timing is now complete and will function perfectly for the remainder of development on the board. Don't forget that you have the two system timers available and that the processor is now sleeping and being woken up by the SysTick timer every

1ms. The HEARTBEAT LED will also now properly reflect how busy the system is. Right now, it should be quite dim because the processor is sleeping most of the time.

Optimizing power consumption in an embedded system takes time and patience. When designing very low power systems, use a programmed real device with a current meter in series with the main power source to measure the actual total current consumption is what you expect. For microamp levels, you need a good multimeter for the measurement to be accurate. You probably need to measure sleep current and operating current separately because they can be many orders of magnitude different and good low-current measurements can't usually be done without switching scales in the multimeter to measure the higher current modes. That does not always work.



**Figure 7-6 Measure sleep current and operating current**

Even if the measurement system is perfect, the current you measure will most likely be higher than you expect! This is where great embedded designers can use their skills to get power consumption down to as low a level as possible. There are many things to check:

- Overall system duty cycle – is the system really sleeping as much as expected?
- Sleep entry and exit optimization – have you accounted for the time it takes to shut down the clock(s), sleep, and then wake up and restart?
- Code optimization – are you running the minimum instruction cycles during wake time?
- Peripheral optimization – is every peripheral shut down when you don't need it?
- Clock optimization – is every clock signal shut down when you don't need it? Can you use a slower clock in some cases? The slower the clock, the less power consumption. For example, waking up due to a button press could be done on the slow but low-power RC oscillator.
- Hardware optimization – every active circuit in a design will consume some power. Every IC has a "quiescent" current that is required just to make the IC active even if it is not doing anything. This includes power supplies. Many ICs have enable lines that can be controlled by the main processor to shut down the IC when it's not in use. Transistor switches can also be used to remove power from circuits that don't need to be on all the time. A classic example is a battery measuring circuit that needs a resistor divider to scale the battery voltage down for the processor to read it. That resistor divider will always leak some current. Many devices might only need to check their battery once a day or once a week, so if the power can be turned off to the resistor divider, power consumption can be reduced significantly. Obviously, any hardware required for power saving needs to be designed in at the beginning.

### 7.3 • Timer Peripheral

The built-in timer on Cortex-M3 cores is great, but having more than just one timer and having timers that have more features is important. Timers are ubiquitous peripherals on microcontrollers. At the most basic level, a timer peripheral is a counter that increments with some signal source. It can be configured to trigger an interrupt or some other system action when it reaches a certain value or overflows. Some timers count up, some count down. Some are as small as 8-bit, others are 32-bits or more.

Timer peripherals can often function as square wave generators (PWM – pulse width modulation) by toggling output lines at set intervals. This functionality is supported by the Timer Counter peripheral on the SAM3U2 microcontrollers but there is also a separate PWM controller that we will look at in the next chapter that offers even more control. This feature is very useful because the Timer peripheral can do this without ever interrupting the core. The core can be busy processing other tasks, interrupts, or sleeping. The timer-driven output is very accurate and can be generated without consuming a lot of power.

Another very powerful feature of timers is that they can be clocked by signals from external pins. This is considered the “counting” mode of a timer peripheral and explains why the peripherals are usually called timer-counters. There are a lot of applications for counting mode with a favorite being reading quadrature encoders on motor shafts to determine revolutions and/or speed of a motor.

The SAM3U2 Timer Counter has a hardware-selectable quadrature encoder function that will automatically determine the direction and provide that information in a flag bit. This is excellent for an application like a rotary user input where an external user is controlling the mechanical action. The direction bit is less necessary for monitoring a motor that is also being controlled by the MCU since the MCU will be controlling the direction already. In this case, a single channel of the encoder output could be used to count revolution steps and only a small firmware routine would be necessary to clear the counter upon direction change, although there are ways to get errors here so counting both channels is recommended.

We’ll focus on configuring the Timer Counter as a timer that can interrupt and run a “callback” function whenever the timer reaches the set value. Learning how to use the Timer Counter peripheral on the SAM3U2 follows the same steps required to learn any peripheral.

1. Read the whole peripheral description section in the user guide making sure to note any apparently relevant features.
2. Examine the block diagrams to get an idea about the hardware and logical relationships.
3. Study the user interface to determine the registers required for configuration and usage of the peripheral.
4. Code the low-level driver and test thoroughly.

Be careful with the Timer Counter documentation because there are some confusing parts. It is stated that there are three channels in the module, but in the Embedded Characteristics, it states there are a total of 9 channels. The SAM3U2 only has one timer module, TC0, which has 3 channels. These registers are accessed in the peripheral register struct `AT91PS_TC`. The base addresses for the three channels in `AT91SAM3U4.h` are confusingly referenced as TC0, TC1, and TC2. Perhaps TC00, TC01, and TC02 would be more intuitive.

There is also something called the timer control “block.” This provides some connectivity between the three independent timers channels within the TC0 module. The registers here are accessed in the register struct `AT91PS_TCB`. This is confusing because of the



types used. The type of the first three members is “AT91S\_TC” which means they are complete structs for a Timer Counter channel. If you count the four Reserved words, that’s a total of 0x40 bytes between the timer channels. The addressing works out such that the two TCB registers ((TCB\_BCR and TCB\_BMR) are correctly located.

```

AT91SAM3U4.h x
2922 // *****
2923 //          SOFTWARE API DEFINITION FOR Timer Counter Interface
2924 // *****
2925 #ifndef __ASSEMBLY__
2926 typedef struct _AT91S_TCB {
2927     AT91S_TC TCB_TC0; // TC Channel 0 TC0 Channel 0 struct
2928     AT91_REG Reserved0[4]; //
2929     AT91S_TC TCB_TC1; // TC Channel 1 TC0 Channel 1 struct
2930     AT91_REG Reserved1[4]; //
2931     AT91S_TC TCB_TC2; // TC Channel 2 TC0 Channel 2 struct
2932     AT91_REG Reserved2[4]; //
2933     AT91_REG TCB_BCR; // TC Block Control Register
2934     AT91_REG TCB_BMR; // TC Block Mode Register
2935     AT91_REG Reserved3[9]; //
2936     AT91_REG TCB_ADDR_SIZE; // TC ADDR_SIZE REGISTER
2937     AT91_REG TCB_IPNAME1; // TC IPNAME1 REGISTER
2938     AT91_REG TCB_IPNAME2; // TC IPNAME2 REGISTER
2939     AT91_REG TCB_FEATURES; // TC FEATURES REGISTER
2940     AT91_REG TCB_VER; // Version Register
2941 } AT91S_TCB, *AT91PS_TCB;

```

Figure 7-7 Type of the first three members is “AT91S\_TC”

What is wrong, however, are the header file definitions for the TCB blocks. Since there is only one Timer Counter, there is only one TCB and using the TCB1 or TCB2 base addresses would result in erroneous addressing.

```

AT91SAM3U4.h x
6719 #define AT91C_BASE_TC0 (AT91_CAST(AT91PS_TC) 0x40080000) // (TC0) Base Address
6720 #define AT91C_BASE_TC1 (AT91_CAST(AT91PS_TC) 0x40080040) // (TC1) Base Address
6721 #define AT91C_BASE_TC2 (AT91_CAST(AT91PS_TC) 0x40080080) // (TC2) Base Address
6722 #define AT91C_BASE_TCB0 (AT91_CAST(AT91PS_TCB) 0x40080000) // (TCB0) Base Address
6723 #define AT91C_BASE_TCB1 (AT91_CAST(AT91PS_TCB) 0x40080040) // (TCB1) Base Address
6724 #define AT91C_BASE_TCB2 (AT91_CAST(AT91PS_TCB) 0x40080080) // (TCB2) Base Address

```

Figure 7-8 Header file definitions are incorrect

We could just ignore this or correct the header file by deleting the TCB1 and TCB2 references, but leaving it as-is and pointing it out helps to emphasize that mistakes are often present in vendor-sourced documents and source code. Even microcontrollers themselves can have bugs (remember the Intel Pentium fiasco?). We have found silicon errata in microcontrollers and worked with the vendor to confirm and document it. No doubt there will be errors in this book. People do their best to provide perfection, but mistakes will always happen. Part of being an engineer is to constructively contribute to improving the resources and technology available. It is certainly extremely frustrating and sometimes very expensive to find an error or bug, but the best way forward is positive understanding and cooperation to fix the issue.

## 7.4 • Timer Counter Highlights



Each timer channel can be clocked from any of 5 sources based on MCK or SLCK which is selected as TCLKx. These are multiplexed with other signals TIOAx that can also be used to clock the timers in sequence which could effectively create a 48-bit timer in hardware. Try to work through the block diagram and understand the clock signaling available.

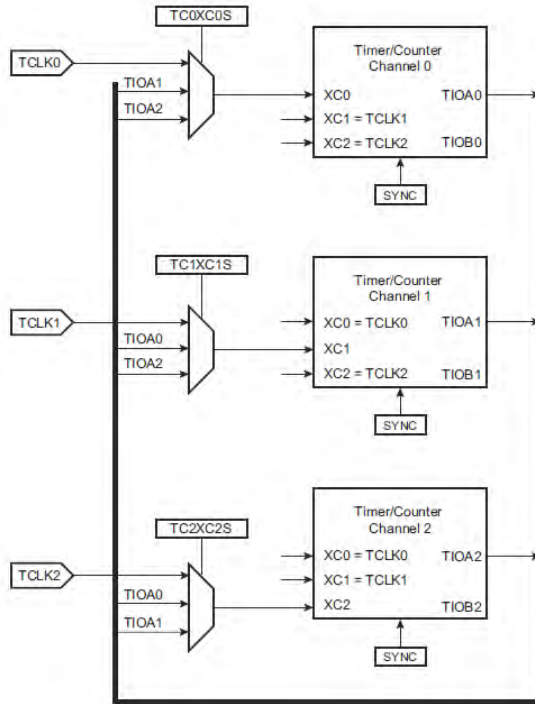


Figure 7-9 Timer channel block diagram

There are quite a few IO lines associated with TC0. The EiE development board is not optimized for the Timer Counter, but several of the signals are available on unused test points or could be borrowed from their official function with little effort. For example, the SD card write-protect input (SD\_WP, PA1) uses the TIOA0 pin. If there is no SD card in use, this line is available and could be wired in with TIOA1 and TIOB1 (PB5 and PB6 test points) for the TC0 quadrature decoder.

Once a clock source is selected, it needs to be triggered on. This is a critical step that isn't documented that well and caused us 3 hours of debugging the first time the Timer driver task was written. In hindsight, it was a simple problem involving a single bit, but things like this happen all the time when working with a new peripheral. Understanding a problem and knowing how to find and use the resources available to solve it are critically important.

The Clock Control diagram (Figure 7-10 on page 236) turned out to be the key to solving the issue, though we should have discovered it sooner from the documentation. To read the diagram, you need to know what an SR (set-reset) latch is. Working from left to right, the counter clock is ANDed with the SR output, so the SR output must be 1. For that to happen, the clock must not be disabled or stopped so R is at 0. S must be 1, which comes from an AND gate driven by the clock start bit (CLKSTA) and (this is what we missed) a trigger input.

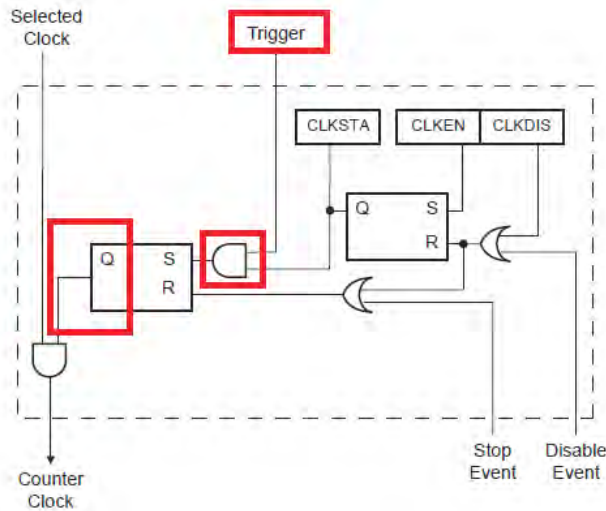


Figure 7-10 Clock control trigger input

Configuring the TC peripheral properly requires a lot of reading. The first decision to make is what mode to run the peripheral in: capture, or waveform. Since the goal of the EiE Timer driver is to provide an alarm-clock style of function, the peripheral must be run in Waveform mode. The pin output feature won't be used. In Waveform mode, the timer can be set up to count to a value loaded in the TC\_RC register and then automatically restart, which is what we want. This corresponds to the WAVSEL bits = 10 with the behavior shown:

**WAVSEL = 10 without Trigger**

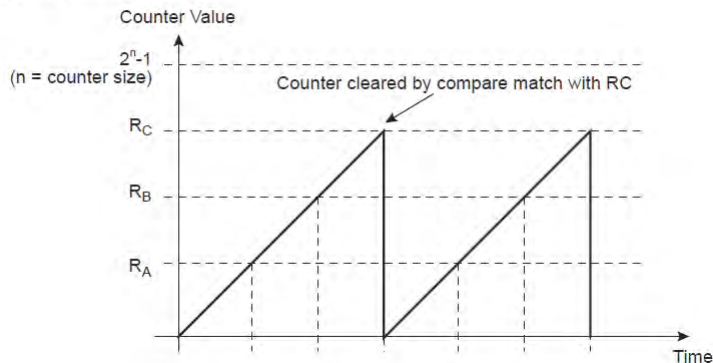


Figure 7-11 WAVSEL without trigger

A single timer channel is only 16 bits, so we will use  $MCK/128$  as the clock source which gives  $2.67\mu s$  per tick for a total available time of 174ms. That might not seem like much, but having this available provides a perfect resource for system timing at smaller resolutions than the 1ms system tick. The smallest resolution would be with  $MCK/2$  that would yield 41.7ns ticks with a total time of 2.7ms – still potentially very useful depending on the application requirements.

For the planned application, the timer does not need to have any external triggers, nor does it need to manage any output pins. It does need to generate an interrupt, though, since the intent is for a task to use this timer to clock out precise intervals outside of the

1ms program loop. For the precision we're looking to provide, the interrupt latency is negligible. If more precision was needed, there are ways to achieve this.

The quadrature encoder functionality is very cool. This would be a great project to test out and we will be sure to have an example on the EiE website. The QDEC will not be enabled for this task, but the section in the user guide is an interesting read to get an idea of everything that is involved in reading an encoder.

## 7.5 • Timer Counter Registers

We now have a good idea about how the timer works and what we need to configure in the peripheral registers. Going through the register bit descriptions and setting up INIT values should fill in any gaps about the operation of the timer. If the purpose of a register or bits in a register is unclear, you can search for that specific information in the user guide. This type of iterative process to learning a new peripheral is common since you first need to learn what you need to learn.

There are three instances of each TC register (one per channel), but only a single instance of the TCB block registers. Notice the address notation in the register description pages, e.g. TC\_CCRx 0x40080000 (0)[0] means the address is for Timer Counter (0) channel [0]. Search for the TC register struct AT91PS\_TC and you should come up with the base addresses for each channel.

```
#define AT91C_BASE_TC0 (AT91_CAST(AT91PS_TC) 0x40080000) // (TC0) Base Address
#define AT91C_BASE_TC1 (AT91_CAST(AT91PS_TC) 0x40080040) // (TC1) Base Address
#define AT91C_BASE_TC2 (AT91_CAST(AT91PS_TC) 0x40080080) // (TC2) Base Address
#define AT91C_BASE_TCB (AT91_CAST(AT91PS_TCB) 0x40080000) // (TCB0) Base Address
```

Just as we did for the PIO registers, we'll go through the important TC registers to highlight the details and provide INIT values where applicable. The same style of defining each bit in the header file is used in timer.h which is provided in the start-up code for this chapter.

**Channel Control Register (TC\_CCR):** Three important bits are here: one each to enable or disable the counter's clock (remember the Clock Control diagram), and the very important software trigger bit, SWTRG. Control registers don't usually require setup values, but for this, we will define a setup value to initialize the clock to be disabled.

```
#define TC1_CCR_INIT (u32)0x00000002
```

**Channel Mode Register (TC\_CMR) WAVEFORM MODE:** There are two separate definitions for the TC\_CMR register that depend on which mode the Timer Counter is run in. Ignore the "Capture Mode" bit descriptions and look at the "Waveform Mode" bit descriptions. The majority of configuration for the Timer Counter is done here.

Clock selection is set with the three LSBs plus one more bit for polarity. The clock signal can be gated by one of the XC sources. The behavior of the timer when it reaches registers TC\_RA, TC\_RB and TC\_RC are all configured here as well as some external triggering and Capture / Waveform mode selection. Since there are so many options, the full INIT value description is shown. Note that in addition to restarting the timer when it reaches RC, TIOA is toggled. This can be used to verify the timer signal and potentially drive a circuit if required.

```
#define TC1_CMR_INIT (u32)0x000CC403
/*
    31 [0] BSWTRG no software trigger effect on TIOB
```

```

30 [0] "
29 [0] BEEVT no external event effect on TIOB
28 [0] "

27 [0] BCPC no RC compare effect on TIOB
26 [0] "
25 [0] BCPB no RB compare effect on TIOB
24 [0] "

23 [0] ASWTRG no TIOA software trigger effect
22 [0] "
21 [0] AEEVT no TIOA effect on external compare
20 [0] "

19 [1] ACPC Toggle TIOA on RC compare
18 [1] "
17 [0] ACPA No RA compare effect on TIOA
16 [0] "

15 [1] WAVE Waveform Mode is enabled
14 [1] WAVSEL Up to RC mode
13 [0] "
12 [0] ENETRГ external event has no effect

11 [0] EEVT external event assigned to XC0
10 [1] "
09 [0] EEVTEDG no external event trigger
08 [0] "

07 [0] CPCDIS clock is NOT disabled when reaches RC
06 [0] CPCSTOP clock is NOT stopped when reaches RC
05 [0] BURST not gated
04 [0] "

03 [0] CLKI Counter incremented on rising edge
02 [0] TCCLKS TIMER_CLOCK4 (MCK/128 = 2.67us / tick)
01 [1] "
00 [1] "

```

```
*/
```

**Counter Value (TC\_CV):** The readable value of the counter. This register would not typically be used since the main point of the TC driver is to count to a particular value. However, reading the intermediate value may be useful at least for debugging to confirm the timer is counting and resetting properly.

**TC Register A / B / C (TC\_RA / TC\_RB / TC\_RC):** These three registers hold target value(s) that the timer will trigger when it reaches them. They are 32-bit registers, but only 16 bits apply since the timer itself is 16 bits. Only TC\_RC will be used, and it will be initialized to give a 100us timer count by default. Remember for a continuously running timer, the value should be value - 1.

```
/* Default Timer 1 interrupt period of about 100us (1 tick = 2.67us); max 65535 */
#define TC1_RC_INIT (u32)37
```

**Status Register (TC\_SR):** The Timer Counter's status register provides information about the timer/counter operation that may be useful to the application. This includes absolute overflow information as well as information on status relative to the three TC\_Rx registers. TC\_SR also provides pin and interrupt flag information. For the EiE application of the Timer Counter, only the RC compare interrupt flag will be needed.

**Interrupt Enable / Disable / Mask Register (TC\_IER / TC\_IDR / TC\_IMR):** All interrupt functionality is managed here. The TC\_IER register will be used to set up the RC Compare interrupt. No other interrupts are required for this application.

```
#define TC1_IER_INIT (u32)0x00000010
/*
    31-08 [0] Reserved

    07 [0] ETRGS RC Load interrupt not enabled
    06 [0] LDRBS RB Load interrupt not enabled
    05 [0] LDRAS RA Load interrupt not enabled
    04 [1] CPCS RC compare interrupt is enabled

    03 [0] CPBS RB compare interrupt not enabled
    02 [0] CPAS RA Compare Interrupt enabled
    01 [0] LOVRS Lover's bit? Load Overrun interrupt not enabled
    00 [0] COVFS Counter Overflow interrupt not enabled
*/
```

**Block Control Register (TC\_BCR):** A register that defines only a single bit that can be used to simultaneously trigger all three of the channels in the Timer Counter. This is important for precise timing or control applications, especially when interrupts are active in the system. It is not required for EiE.

**Block Mode Register (TC\_BMR):** This register has a bunch of configuration bits. The lowest 6 bits assign clock sources and the remaining bits configure the quadrature decoder. Since the QDEC is not used, the most important bit to correctly configure is QDEN which disables the decoder. The other QDEC bits are not relevant. Though we don't intend to use the XCx clock sources, we set them up to connect to their respective TCLKx signals.

```
#define TCB_BMR_INIT (u32)0x00100800
```

**QDEC Interrupt Enable / Disable / Mask / Status Register (TC\_QIER / TC\_QIDR / TC\_QIMR / TC\_QISR):** If using the QDEC, you may want to use the interrupt features which are managed with these registers. No configuration required for EiE.

## 7.6 • Timer Driver

All configuration, operation, and API will be contained in timer.c and timer.h. Make sure the files are in the project, timer.h is in configuration.h, and the required initialization and state machine calls are in main.c. The timer driver can be used to manage all timers, but we will only setup channel 1 (chosen arbitrarily).

Add code to TimerInitialize() to load all the INIT values defined for channel 1 as well as the TCB\_BMR value. The timer will use the RC compare interrupt, so use the NVIC functions to clear IRQn\_TC1 and then enable those interrupts. The code would build and run even though we have not written the TC1 ISR because the WEAK default handler would be invoked by default. The callback function will be added later.

```
void TimerInitialize(void)
{
    /* Load the block configuration registers */
    AT91C_BASE_TCB0->TCB_BMR = TCB_BMR_INIT;

    /* Load Channel 1 settings */
    AT91C_BASE_TC1->TC_CMR = TC1_CMR_INIT;
```

```

AT91C_BASE_TC1->TC_RC = TC1_RC_INIT;
AT91C_BASE_TC1->TC_IER = TC1_IER_INIT;
AT91C_BASE_TC1->TC_IDR = TC1_IDR_INIT;
AT91C_BASE_TC1->TC_CCR = TC1_CCR_INIT;

/* If good initialization, set state to Idle */
if( 1 )
{
    /* Enable required interrupts */
    NVIC_ClearPendingIRQ(IRQn_TC1);
    NVIC_EnableIRQ(IRQn_TC1);
    Timer_StateMachine = TimerSM_Idle;
}
else
{
    /* The task isn't properly initialized, so shut it down and don't run */
    Timer_StateMachine = TimerSM_Error;
}
} /* end TimerInitialize() */

```

Everything else will be API or interrupt driven, so this is really a simple driver. The empty Idle state can remain in place for future advancements.

## 7.7 • Timer API

The intended application for the timer driver is to provide a simple interface to use the timer like an alarm clock. Take a moment to consider what functions would be useful to a user. To do that, consider any real-life application where you use a clock or timer. How do you use it? What features do you need? Obviously, you need to be able to set the timer value. Starting and stopping the timer is essential. You might want to know the time while it is running to see how much time has elapsed. Lastly, you need to take some sort of action when the time has expired. For this driver, that will be done by running a callback function so there must be a utility for the user to assign their own function here. This is the mechanism by which any user task that uses the API can connect independently to another task.

A choice needs to be made about how the API will be implemented with respect to the individual channels. A function for each channel could be written (e.g. Timer1Start, Timer2Start, etc), or the channel of interest could be passed as a parameter (e.g. TimerStart(Channel) ). Since we don't know how the code will be implemented yet, we will choose to use generic functions and pass the channel as a parameter. This makes the most sense from a scalability and maintenance point of view.

In timer.h, define an enum type called TimerChannelType for the available channels in the system. The intent is to use address offsets to access the registers for each channel, so the enum values are the address offsets between groups of channel registers. These are determined from the base address offsets for TC0, TC1, and TC2.

```

/*!
@enum TimerChannelType
@brief Controlled list of available timer channels used in the member functions.
*/
typedef enum {TIMER0_CHANNEL0 = 0, TIMER0_CHANNEL1 = 0x40, TIMER0_CHANNEL2 = 0x80}
TimerChannelType;

```

All the API functions will need the TimerChannelType parameter. Setting the timer requires the 16-bit value to which it will be set. The callback function assignment can

be done with a function pointer exactly like the function pointers used for all the system tasks.

The public functions can now be defined from the requirements.

#### PUBLIC FUNCTIONS

```
- void TimerSet(TimerChannelType eTimerChannel_, u16 u16TimerValue_)
- void TimerStart(TimerChannelType eTimerChannel_)
- void TimerStop(TimerChannelType eTimerChannel_)
- u16 TimerGetTime(TimerChannelType eTimerChannel_)
- void TimerAssignCallback(TimerChannelType eTimerChannel_, fnCode_type
fpUserCallback_)
```



Write the first four API functions. The full header for `TimerSet()` is shown below if you need help getting started. If you need more help, read further to the code explanation. Once you write the first function, the others are mostly copy-paste with slight adjustments to bits and registers.

```
@fn void TimerSet(TimerChannelType eTimerChannel_, u16 u16TimerValue_)

@brief Sets the timer tick period (interrupt rate).

Requires:
- TimerStop should be called before, and TimerStart should be called after
this function to reset the counter and avoid glitches.
@param eTimerChannel_ holds a valid channel
@param u16TimerValue_ x in ticks

Promises:
- Updates register TC_RC value with u16TimerValue_
*/
```

To address the correct timer channel, build the 32-bit address from `AT91C_BASE_TC0` and the channel offset value.

```
void TimerSet(TimerChannelType eTimerChannel_, u16 u16TimerValue_)
{
    /* Build the offset to the selected peripheral */
    u32 u32TimerBaseAddress = (u32)AT91C_BASE_TC0;
    u32TimerBaseAddress += (u32)eTimerChannel_;
```

The `TC_RC` register is used as the “count-to” value. There are no restrictions about loading this register, so a potential error is loading a lower value than what is currently loaded while the actual timer count is already above the value that is being loaded. Without any action, the timer would then have to overflow and then count up again to the new value. The timer could be reset inside `TimerSet()`, but that might not be desired, so it will be left as a note to the user to manage how they want.

Use the address that was set up for the channel of interest to load the correct channel’s `TC_RC`. Recall from the LED driver that the 32-bit address that was calculated needs to be cast back to the special register type used in `AT91SAM3U4.h`. Since `TC_RC` is a 32-bit value, safely cast the timer value function parameter up to 32-bits. Whenever type casts are used, be very careful to ensure that the resulting values are correct.

```
/* Load the new timer value */
(AT91_CAST(AT91PS_TC)u32TimerBaseAddress)->TC_RC =
    (u32)(u16TimerValue_) & 0x0000FFFF;
} /* end TimerSet() */
```



TimerStart() is nearly identical. The channel address offset is calculated, then it uses the TC\_CCR register which is written to enable the timer clock and provide the software trigger which resets and starts the clock.

```
/* Ensure clock is enabled and triggered */
(AT91_CAST(AT91PS_TC)u32TimerBaseAddress)->TC_CCR |= (AT91C_TC_CLKEN |
                                                         AT91C_TC_SWTRG);
```

TimerStop() just disables the clock in the channel's TC\_CCR register.

```
/* Ensure clock is disabled */
(AT91_CAST(AT91PS_TC)u32TimerBaseAddress)->TC_CCR |= AT91C_TC_CLKDIS;
```

Notice there is no way to just “pause” the timer with the implementation of TimerStart() and TimerStop(), since TimerStart() always resets the timer. Pausing does not make sense in this application since the timer period is so short. That type of functionality can be handled at the user level. If it were ever an issue, a TimerRestart() function could be added to just re-enable the clock without resetting the timer though you'd have to confirm that the clock is still indeed triggered.


Getting the timer value means returning the TC\_CV value of the specified channel. Only 16-bits are provided.

```
/* Read and format the timer count */
return ((u16)( (AT91_CAST(AT91PS_TC)u32TimerBaseAddress)->TC_CV & 0x0000FFFF));
```

These four API functions are very simple. As it turns out, the choice of using generic functions and passing the channel as a parameter might not have been the best. As they exist now, the four functions only require a single line of code to do what they really need to do if you took out the channel-specific address adjustments. Therefore, these could all be in-lined or replaced by macros that would not be function calls at all which would save some instruction cycles. However, we do not know what the future holds for this timer driver, and to maximize portability the API functions as they are written work best. If you were optimizing the code for a specific, ultra-low power device this is a place where a few dozen instruction cycles could be saved.

The last function is also easy but is described here as some syntax is new. TimerAssignCallback() loads a function pointer with the address of the function that the user wants to run whenever the timer interrupt occurs. The declaration for this pointer is made immediately after the task function pointer declaration.

```
/******
Global variable definitions with scope limited to this local application.
Variable names shall start with "Timer_<type>" and be declared as static.
******/
static fnCode_type Timer_fpStateMachine;      /* State machine function pointer */
static fnCode_type Timer_fpTimer1Callback;    /* Timer1 ISR callback pointer */
```

 To make sure there are no unassigned pointers, we will write a private default callback function. This ensures that the pointer is always valid even if the user forgets or does not want to assign a callback function.

```
/*!-----
@fn static void TimerDefaultCallback(void)

@brief An empty function that the Timer Callback points to by default. Expected
```

that the user will set their own.

Requires:

- NONE

Promises:

- NONE

```
*/
static void TimerDefaultCallback(void)
{
}
/* end TimerDefaultCallback() */
```

Initialize the callback pointer to this function in `TimerInitialize()`.

```
Timer_fpTimer1Callback = TimerDefaultCallback;
```



The last step is to write the interrupt handler for the timer. Each channel in the Timer Counter has its own ISR, so just add `TC1_IrqHandler()` at this time. The ISR should verify that the RC interrupt occurred and then invoke the callback function. The RC bit is bit 4 in the status register and the interrupt registers and is named `CPSC`. In most cases, the bit name descriptions in the user guide have a corresponding definition in the processor's main header file. If you search `AT91SAM3U4.h` for `CPSC` you will find it. The full name definition is `AT91C_TC_CPSC`. Atmel does not use a naming convention to distinguish bit names from other constant names like `EiE` does with a leading underscore.

```
AT91SAM3U4.h x timer.c main.c
2906 // ----- TC_SR : (TC Offset: 0x20) TC Channel Status Register ---
2907 #define AT91C_TC_COVFS      (0x1 << 0) // (TC) Counter Overflow
2908 #define AT91C_TC_LOVRS      (0x1 << 1) // (TC) Load Overrun
2909 #define AT91C_TC_CPAS       (0x1 << 2) // (TC) RA Compare
2910 #define AT91C_TC_CPBS       (0x1 << 3) // (TC) RB Compare
2911 #define AT91C_TC_CPSC       (0x1 << 4) // (TC) RC Compare
```


Figure 7-12 Bit definition for `CPSC` using left shift

Notice the way the bit location value is defined (`0x1 << 4`). This is a common way to define bit positions or the positions of groups of bits. It takes a set bit in bit position 0 (`0x1`) and left shifts it to bit 4. The result is no different than directly writing `0x10` or `b'0001 0000` or any other way it could be expressed. Since it is a preprocessor definition, wherever the `AT91C_TC_CPSC` symbol is used the actual number will be inserted before the code is compiled so there is no code usage penalty. There are advantages and disadvantages to making definitions this way, but for sure it is widely used in processor documentation. This became common a while ago and may be the product of an automated source code generator tool that writes the lengthy header files for all the processors.

Regardless of how it is defined, `AT91C_TC_CPSC` is the bit that needs to be checked. From the Timer Counter user guide, the corresponding interrupt bit is in the status register, `TC_SR`. The act of reading `TC_SR` clears the flag. Since our Timer Counter implementation only uses a single interrupt, a copy of `TC_SR` does not have to be made. Check the bit directly and simultaneously clear the flag like this:

```
if(AT91C_BASE_TC1->TC_SR & AT91C_TC_CPSC)
```

If you look at the assembly language for that line of C code, you will see the TC\_SR register get read (which clears the flag) and loaded into a core register so it can be bit-wise logically ANDed with CPCS. The result flags are tested to make the decision of the “if” statement. If that is confusing, please reference the Assembly chapter.

 Write the rest of the ISR. Remember to clear the NVIC IRQn\_TC1 flag. Add a debug counter Timer\_u32Timer1IntCounter to track how many times this occurs which will be useful for verifying everything is running correctly. The solution is shown.

```

/*!-----
@fn void TC1_IrqHandler(void)

@brief Parses the TC1 interrupts and handles them appropriately.

Note that all enabled TC1 interrupts are ORed and will trigger this handler,
therefore, any expected interrupt that is enabled must be parsed out
and handled.


Requires:
- NONE


Promises:
- If Channel1 RC: Timer Channel 1 is reset and automatically

*/
void TC1_IrqHandler(void)
{
    /* Check for RC compare interrupt - reading TC_SR clears the bit if set */
    if(AT91C_BASE_TC1->TC_SR & AT91C_TC_CPCS)
    {
        Timer_u32Timer1Counter++;
        Timer_fpTimer1Callback();
    }

    /* Clear the TC1 pending flag and exit */
    NVIC_ClearPendingIRQ(IRQn_TC1);
} /* end TC1_IrqHandler() */

```

 Build the code and fix any errors. The Timer Counter API is now complete. Currently, there is no requirement to run a state machine but that could easily change as features are added so the task will be kept in the system and run an empty Idle state.

 To test the API, write a user app that uses the timer API to blink an LED at 4 Hz (toggle every 125ms). Setup the timer in UserApp\_Initialize(). Use LedToggle() in the callback function. As a quick way to check the timing, configure an LED physically adjacent to the LED the timer will blink and call LedBlink(LED\_4HZ) after the call to TimerStart() during initialize. Even though the LedBlink() function does not start running until the main loop is running, it is fast enough that both LEDs will appear to start blinking simultaneously. Both LEDs should continue to blink at identical rates since 125ms is evenly divisible into the timer period resolution.

The local callback function:

```

/*!-----
@fn void UserApp1TimerCallback(void)

@brief Toggles LED at Timer1 interrupt.

```

```

Requires:
- Automatically called from Timer interrupt

Promises:
- WHITE LED is toggled

*/
void UserApp1TimerCallback(void)
{
    LedToggle(WHITE);
} /* end UserApp1TimerCallback */

```

The new code in UserApp1Initialize():

```

void UserApp1Initialize(void)
{
    /* Initialize LEDs and queue PURPLE to blink */
    LedOff(WHITE);
    LedOff(PURPLE);
    LedBlink(PURPLE, LED_4HZ);

    /* Setup Timer1 to clock out 125ms periods. 125ms / 2.66666us = 46875 */
    TimerAssignCallback(TIMER0_CHANNEL1, UserApp1TimerCallback);
    TimerSet(TIMER0_CHANNEL1, 46875);
    TimerStart(TIMER0_CHANNEL1);
}

```



Try changing the timer count value by 100 and see how long it takes before you notice that the blinking is no longer synchronized. Load the correct value and leave the code running overnight to see if the LEDs still appear to be synchronized in the morning. Since both the SysTick and the Timer Counter are being clocked from MCK, there are no division errors, and both timers are running on interrupts, the LEDs should remain synchronized forever. If the 1ms system time is being violated, then the LedBlink() function will slow down and the LEDs will be noticeably out of sync.



*Running overnight tests is a simple way to extensively increase your productivity. A great rule for yourself as a developer is to always start an overnight test before you go home. Even if you only have an existing product with no new code, set it up to do something and make sure it works when you get back in the morning. To make things more exciting, put the device in the freezer. If random testing ever produces unexpected results, you can conduct more specific controlled testing to debug the problem.*

## 7.8 • Chapter Exercise

The online chapter exercise combines functions of the Timer, Buttons, and LED APIs. It is highly recommended to experiment with the other capabilities of the Timer Counter module, particularly in counter mode. Consider adding API functions to set up a counter channel on one of the available test points.



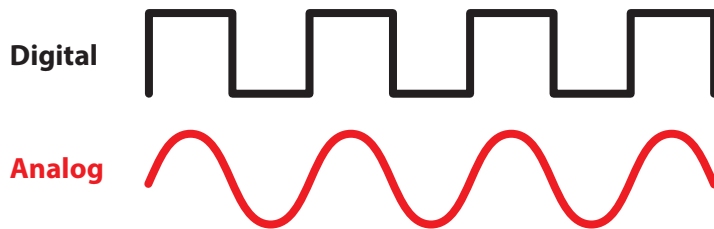
## Chapter 8 • Pulse Width Modulation

This chapter examines yet another very important and fundamental function that an embedded system often provides: Pulse Width Modulation (PWM). We will look at what PWM means, some applications, and how to implement it on the development board both using the PWM peripheral and by brute force techniques. An audio driver will be written so you can start producing ridiculously irritating noise from your buzzer, and the LED application will be updated so it can produce some beautiful lighting effects that will let you brag to your friends, “I know how to do that” when you spot these effects in everyday life.

### 8.1 • PWM Concepts

Digital systems are great because there are only two voltage states, high and low, that carry any information. The potential difference between the two states provides a lot of noise immunity making for very robust systems. However, since the world is analog, digital systems must have a way to step out of the discrete realm and into the infinite continuous space of the real analog world. There is an infinite number of voltage amplitudes and infinite frequencies of time-varying signals. Digital systems can approximate analog output through digital to analog conversion, but still only at discrete levels. Hardware that explicitly does analog to digital and digital to analog conversion of voltage levels is a topic of another chapter.

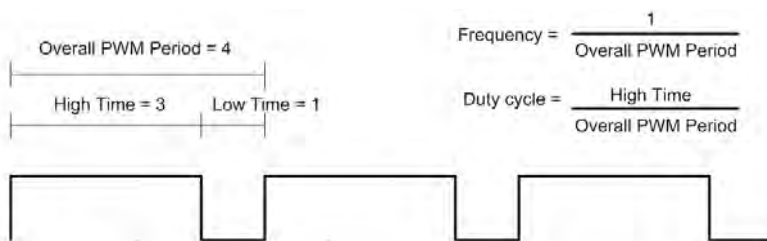
In this chapter, we are mostly interested in PWM generating periodic signals or “waves.” The only signal wave that a digital processor can produce directly is a square wave since the processor can only toggle a digital output high or low. To satisfy the purists, we note that a square wave is not truly a square wave due to finite rise and fall times in any real system. Let’s borrow a strategy from academia to eliminate all those real-world issues and declare these “ideal” square waves for this discussion.



**Figure 8-1** Digital (top) and analog (bottom) periodic signals with equal frequency

Despite the imperfections and approximations, it turns out that a square wave is a very powerful tool when it comes to achieving analog capability in a digital system. Through frequency and duty cycle adjustments of the digital pulses, external systems can be modulated to give the appearance of analog voltages and signals. Lighting and audio are two examples that very tangibly demonstrate these concepts, and will thus be the examples on which we focus.

When a microcontroller is programmed to blink an LED at 1Hz, it is generating a square wave, albeit a very slow one. The LED is on for 500ms and off for 500ms. Since the signal is high for half of the time, we call this “50% duty cycle.” Duty cycle is the ratio of the high time during the period to the total period. Remember that the “period” of a wave is the time between repetitions of the signal and is measured in units of time like seconds, milliseconds, etc. The inverse of period is the signal’s frequency and is measured in Hz, kHz, etc. Duty cycle can change independently of frequency. Figure 8-2 on page 248 shows some details of a 75% duty cycle PWM.



*PWM signal details (75% duty cycle)*

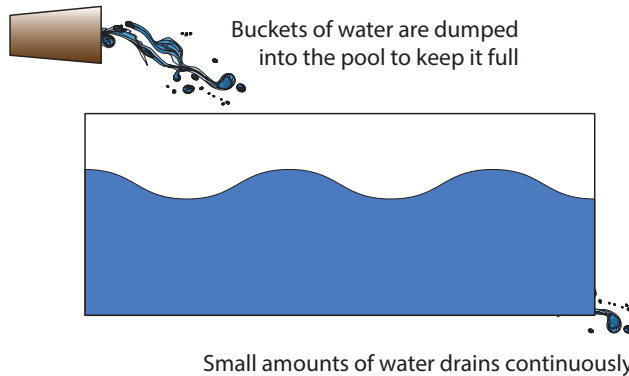
**Figure 8-2 PWM signal details (75% duty cycle)**

If you increased the frequency to 1 kHz, the digital behavior of the LED is still 50% duty cycle, but now the LED is on and off for just 500us each. Since LEDs are very fast at reacting to changes in signal voltage, the LED is, in fact, going from fully-on to fully-off at the drive rate, which you can verify by attaching an oscilloscope and looking at the waveform. However, if you are looking at the LED with your eyeballs while this is happening, you will not see it blinking because your eye cannot respond that quickly. In fact, your eye can only distinguish changes at about 10Hz, though you might see some flicker as fast as 20-30Hz.

The result of looking at the LED being driven by such a fast signal, is that you will see the LED as if it is running at about half the voltage of the supply, so it will appear to be half as bright. We can generalize that a 50% duty cycle results in an effective 50% supply voltage/light intensity if the frequency is fast enough that your eye cannot detect the transitions.

If you measure the effective luminous output it probably is not exactly half, but as a rule of thumb, the apparent LED brightness is equal to the duty cycle of the PWM signal driving it. If you change your driving signal to be on for 250us but off for 750us, the frequency is still 1kHz, period of the signal remains 1ms, but now you are running "25% duty cycle" and the LED will appear about one-quarter of its full brightness. If you maintain the frequency and just vary the duty cycle (on time vs. off time), then you can make the LED any brightness that you want.

Not only does this work for varying LED brightness in a digital system, it has many other applications as well. Switching step-down power supplies use integrated circuits that convert a high input voltage down to a lower voltage using a fast switch and varying duty cycle. This is why switching power supplies are much more efficient than linear regulators – power is only "sampled" as it is needed from the input. There is always a big capacitor at the output of switcher that acts like a tank that keeps getting topped up with a burst of electrons from the switch while a much smaller but steady stream of electrons is trickling out of the capacitor to power the rest of the circuit. A good analogy is a pool of water with a hole. Someone keeps dumping buckets of water into the pool to keep it full.



**Figure 8-3 Filling a pool to compensate for a leak**

Switching power supplies require careful calculation of their output capacitors so that an acceptable amount of “ripple” can be attained (you will always see some evidence of the switching signal). The capacitor, in this case, has the same effect as your eyeball in the LED dimming case as far as filtering out the discrete on/off transitions.

PWM is also used in motor controllers and essentially works the same way to apply an effective voltage to the motor. Varying the effective voltage gives you control over the motor speed. This applies in both AC and DC systems and many microcontrollers have built-in PWM peripherals to target applications like this.

The concept of PWM is everywhere and it is arguably the easiest way to do digital to analog conversion. It is thus essential to know how to program an embedded system to do it. The last chapter introduced the Timer Counter peripheral that could be set up to drive external outputs and generate a PWM signal, and we mentioned the dedicated PWM peripheral that does the same with some more options that we will look at now. This chapter will also look at how to “bit-bash” – or generate manually – a PWM signal which might be necessary if there are not enough peripheral resources available.



If you need PWM outputs in a design, make sure you correctly assign the hardware that requires PWM to a pin where PWM output is available. Also, check that there are no conflicts with the PWM peripheral output for the pin you have chosen.

## 8.2 • PWM the Easy Way: SAM3U2 PWM Peripheral

A PWM peripheral is pretty much just a timer peripheral with some specific PWM features. The most notable difference between the two is that a PWM peripheral can be set up so that different events can occur at different times in the waveform generation period and thus easily allows for varying duty cycles. Many timers have only one event per period, so it is slightly more difficult to vary duty cycle. On some processors, PWM functionality is built into all timer peripherals rather than having discrete PWMs and timers.



Using the PWM peripheral on the SAM3U2 processor is straightforward once you get past what appears to be a massive number of configuration registers. Open the User Guide to the Pulse Width Modulation (PWM) peripheral. We are very purposely repeating the instructions on how to learn a peripheral to drive the point home:

1. Read the whole peripheral description section in the user guide making sure to note any apparently relevant features.
2. Examine the block diagrams to get an idea about the hardware and logical relationships.



3. Study the user interface to determine the registers required for configuration and usage of the peripheral.
4. Code the low-level driver and test thoroughly.

### 8.3 • Peripheral Highlights

The PWM peripheral offers 4 channels, each with two outputs that are compliments of each other. As expected, the signals can be adjusted for frequency and duty cycle, as well as some less common parameters. There are more clock sources available than in the Timer Counter peripheral and there is great emphasis on channel independence and even fault protection. This should hint at the importance of synchronization and glitch-free adjustment of PWM waveforms, which can be extremely important for PWM applications like motor control.

The PWM block diagram looks a bit overwhelming but the peripheral is straightforward to set up and use. Part of the complexity comes from the fact that there are many channels within the peripheral, but each channel is identical. Spend some time tracing through the diagram to make sense of what is going on for one of the channels. There are a lot of IO lines associated with the PWM module and many can be output through the daughterboard connector.

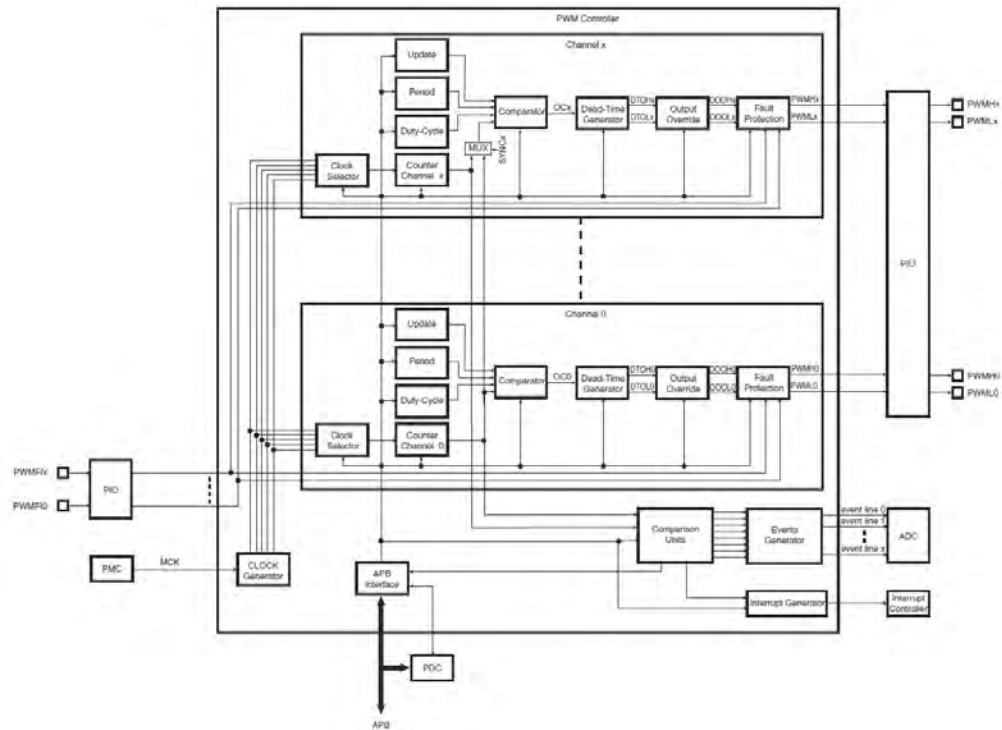


Figure 8-4 PWM peripheral controller hardware

There are 13 different clocking options including MCK scaling from 1 to 1024 and two arbitrary generators that can be programmed. There are two clock behaviors, “left

justified” and “center justified” that control how exactly the channel counter behaves. The center justification can further extend the period by a factor of two, but can also trigger an interrupt at the mid-way point of the signal.

The dead time and fault protection can help with motor controllers and ensure safe operating conditions if certain things go wrong. For example, when switching motor directions or changing speed. The PWM module has external fault input pins where an outside monitor could tell the microcontroller that something is wrong. An external motor controller might have a “coast” and/or “break” function, either of which could be desired or catastrophic if applied at the wrong time. If a fault condition occurred as determined by an external monitor, the hardware can be configured to go to a known state. None of this is required for EiE, but this knowledge helps build a foundation for future applications.

Changing duty cycles is prone to error because of making transitions on the fly. This was pointed out in the last chapter when it came to resetting the timer TC\_RC value that could result in adjusting the end time to a point before the current time and thus having to wait for the timer to overflow. Not a big deal for an alarm clock timer application with a relatively short period, but potentially a huge deal for a motor controller that is clocking very slowly.

The user guide mentions that the PWM module is connected to the “PDC” which stands for “Peripheral DMA Controller.” DMA, in turn, stands for Direct Memory Access. Next chapter you will learn all about the PDC and DMA – for now it can be ignored.

There are extensive instructions for initialization and updating parameters in the PWM module. Scan through these, but don’t worry too much as the steps needed for the EiE application will be detailed later. If you are designing a critical motor-control application to be as robust as possible, be sure to consider all this information carefully. This holds true for interrupt configuration as well as using the various write-protect registers available in the module.

#### 8.4 • PWM and Audio

Generating a square wave to drive a beeper/buzzer is a great way to take advantage of a microcontroller’s PWM peripheral, even though the use of the peripheral in this way is not the way it really is intended to be used. PWM is typically used with constant frequency and varying duty cycle. Using the PWM for audio will keep a 50% duty cycle and just change the frequency. This also is much less a digital to analog application of PWM since we are not looking at creating voltages between the supply rails, but rather just emulating the sinusoidal signals that make up audio.

Speaker output is typically a combination of periodic driving signals with variable amplitude (louder and softer volume) and variable frequency (higher and lower tones). Traditional speakers have coils that work like an electromagnet to physically move the speaker in and out at the signal frequency which then produces sound waves that propagate through the air and rattle our eardrums. It takes considerable current to drive speakers like this and requires some type of amplifier to provide that current. You cannot drive a magnetic coil speaker directly from a processor because a processor cannot source enough current. Even a small speaker coil requires tens if not hundreds of milliamps of drive current. A big sub-woofer can pull amps of current.

Piezoelectric (pronounced pee-EH-zoh-electric) buzzers like the ones that are on the development board are not traditional speakers. They are just a metal disk that uses the piezoelectric effect to bend the disk with applied voltage. There is almost no current involved, so they are suitable to drive directly from a microcontroller and respond very nicely to a square-wave drive and they can usually be driven directly from the processor pin. This is highly convenient for little digital systems, but the tradeoffs include having

just a single volume and very annoying sound quality. Piezo buzzers usually work best between 1kHz and 4kHz which are quite high frequencies.

**!** Regardless of the type of speaker/buzzer you have, never leave a DC voltage applied to it. If you are not driving the speaker, the output pin to the speaker should be at 0 volts or changed to input high-impedance.

The trade-offs are well worth it to provide low-cost, easy, basic audio. Indicator beeps in devices, sounds in toys, alarms, etc. can all be added to systems with very little effort. The microcontroller can be programmed to provide the output directly through bit-bashing, or a PWM peripheral output can be configured to set the audio. A PWM peripheral is the ideal way to do it. You might be slightly impressed at the outputs that can be achieved as you will see in the examples.

## 8.5 • EiE Audio Hardware

The EiE development board uses the PWM peripheral to drive the two piezoelectric buzzers. Channel 0 drives the right buzzer (PA\_28\_BUZZER1), Channel 1 (PA\_29\_BUZZER2) drives the left buzzer. In both cases, only the “high” output signal is configured for the hardware.

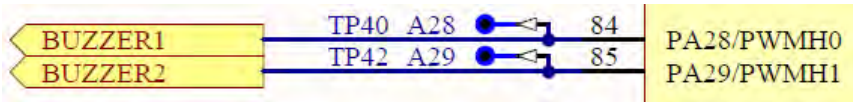


Figure 8-5 Two piezoelectric buzzers driven by PWM

Since the function of the pins is assigned to a peripheral, the relevant PIO registers are set to take the pin out of standard PIO mode and connect it to the peripheral. This has already been in place since we did the PIO module. However, since this is the first time we are using peripheral functionality that depends on it, it’s a good place to draw some attention to it.

The pins must be unassigned from the PIO controller and assigned to the correct peripheral output. The peripheral selection is determined from the I/O lines table in the PWM user guide description. The code below is from `eief1-pcb-01.h` and you should see clearly how it lines up with the SAM3U user guide table.

```
#define PIOA_PER_INIT (u32)0x84030007
/*
    29 [0] PA_29_BUZZER2 PIO control not enabled
    28 [0] PA_28_BUZZER1 PIO control not enabled

#define PIOA_ABSR_INIT (u32)0x7B000000
/*
    29 [1] PA_29_BUZZER2 PERIPHERAL B
    28 [1] PA_28_BUZZER1 PERIPHERAL B
```

Instance	Signal	I/O Line	Peripheral
PWM	PWMF0	PA11	B
PWM	PWMF1	PA12	B
PWM	PWMF2	PA18	B
PWM	PWMH0	PA4	B
PWM	PWMH0	PA28	B
PWM	PWMH0	PB0	A
PWM	PWMH0	PB13	B
PWM	PWMH0	PC24	B
PWM	PWMH1	PA5	B
PWM	PWMH1	PA29	B

Figure 8-6 PWM signals and I/O

Other PIO register setup values ensure the buzzer pins are set as outputs and start at 0 in case they are repurposed back to the PIO controller.

### 8.6 • PWM Registers

There are a ton of registers needed to configure and use the PWM module listed in the User Interface table in the user guide. A lot have the same function just for different channels. There is a common base address for common PWM registers, and there are channel-specific registers at their own base address.

```
#define AT91C_BASE_PPMC_CH0 (AT91_CAST(AT91PS_PPMC_CH) 0x4008C200) // (PPMC_CH0)
#define AT91C_BASE_PPMC_CH1 (AT91_CAST(AT91PS_PPMC_CH) 0x4008C220) // (PPMC_CH1)
#define AT91C_BASE_PPMC_CH2 (AT91_CAST(AT91PS_PPMC_CH) 0x4008C240) // (PPMC_CH2)
#define AT91C_BASE_PPMC_CH3 (AT91_CAST(AT91PS_PPMC_CH) 0x4008C260) // (PPMC_CH3)
#define AT91C_BASE_PPMC (AT91_CAST(AT91PS_PPMC) 0x4008C000) // (PPMC)
```

Note that to be able to write to certain PWM registers, various protection bits must be cleared. This is to make sure that any erroneous address writes in other parts of the code don't accidentally change the PWM configuration values. The protection bits are off by default.

The starting point for the code is in the `pwm_audio` repository and includes the initialization values in the `eief1-pcb-01.h` header file.

**PWM Clock Register (PWM\_CLK):** As the name suggests, this register is used to set up the main clock signals to the PWM peripheral. MCK is always the input clock source and it is always available in one of 11 scaled outputs as part of the PWM peripheral hardware. PWM\_CLK can configure additional clock signals CLKA and CLKB that are derived from the available MCK values. The four channels can then independently select any of these clocks so not all the PWM channels need to have the same clock. The EiE audio functions do not need the CLKx clocks, so they can be turned off to save power.

```
#define PWM_CLK_INIT (u32)0x00000000
```

**PWM Enable / Disable / Status Registers (PWM\_ENA / PWM\_DIS / PWM\_SR):** A simple set of registers to control and monitor the state of each of the four channels. We won't bother initializing this as the channels will be enabled only when they are needed to turn on the buzzer(s).

**PWM Interrupt Enable / Disable / Mask / Status Register 1 (PWM\_IER1 / PWM\_IDR1 / PWM\_IMR1 / PWM\_ISR1):** Channel-specific interrupt control and status for counter events and fault protection triggers. Reading the PWM\_ISR1 status register clears the flags. This is the first of two interrupt register sets as there are many interrupt sources available. Remember that all interrupts for a peripheral are ORED together for a single interrupt into the NVIC. The PWM ISR could end up being quite long if all the available interrupts were enabled. Make a copy of PWM\_ISR1 at the start of the ISR to preserve all the flags and then start parsing through. The EiE firmware will not use PWM interrupts.

**PWM Synchronization Registers (PWM\_SCM, PWM\_SCUC, PWM\_SCUP, PWM\_SCUPUPD):** These four registers control the PWM peripheral to ensure that channels are updated precisely in situations where this matters. They are very application-specific and not required for EiE, so they are left as an exercise to explore further.

**PWM Interrupt Enable / Disable / Mask / Status Register 2 (PWM\_IER2 / PWM\_IDR2 / PWM\_IMR2 / PWM\_ISR2):** The second set of channel-specific interrupt control and status registers. These interrupts provide status for events where the PWM counters have reached certain levels or for other synchronization signals. Most of the flags are cleared by reading PWM\_ISR2 so again, be sure to make a copy of the register in the PWM ISR before starting to parse it out. These registers will not be used for EiE.

**PWM Output Override / Deadtime Registers (PWM\_OOV, PWM\_OS, PWM\_OSS, PWM):** This group of related registers specify the value latched to the PWM outputs if an override function is active. There are two choices, either the “Output override value or the “Dead-time generator” value. The registers work together to configure and select the desired signal if invoked. They are channel-specific and apply separately to the “high” and “low” signals within each channel. Not used for EiE.

**PWM Fault Mode / Status / Clear / Protection Value / Protection Enable Registers (PWM\_FMR / PWM\_FSR / PWM\_FCR / PWM\_FPV / PWM\_FPE):** This group of related registers set up and monitor fault signals and specify the value latched to the PWM outputs if a fault condition is active. Faults are one of four hardware signals (3 pins and the oscillator) that could be inputs to the PWM peripheral. The register bits correspond to the hardware bits. It’s confusing that there are 8 bits for each setting, but only 4 bits are effective. The PWM\_FPE register bytes are the channels and the bits are the fault signals. If using fault protection, be sure to experiment in the safety of a lab to ensure that the behavior is what you expect. These are not used in EiE.

**PWM Event Line Registers (PWM\_ELMR0, PWM\_ELMR1):** There are two internal “event line” connections to the ADC peripheral that can provide triggers to the ADC to run conversions when the PWM channel counters reach certain values. There are two copies of this logic in the processor. Each has the same 8 available events ORED together so PWM\_ELMRx is used to select which of those 8 events are enabled. This is not used in EiE.

**PWM Write Protect Control Register (PWM\_WPCR):** The protection features built into the PWM peripheral are managed here. There are 6 separate bits that are used to protect different registers. Each can be set or cleared. There are also two bits that control the protection overall including locking out the PIO controller from changing the purpose of the pins. Most interestingly, the top 24 bits of the register must be written with a special key value (0x50574D) when writing any of the other bits. If the key is not sent as part of the configuration value, the write operation will be ignored. If you ever doubted that engineers had a sense of humor, work out what 0x50, 0x57 and 0x4D are in ASCII codes.

**PWM Write Protect Status Register (PWM\_WPSR):** This register reports what register protection bits are currently active and even flags any violations that have occurred. A robust system would monitor this register and make sure that a violation never occurs. If it does occur, the register offset address that was attempted to be written and caused the

flag is captured in the top 16 bits. Very cool.

The last group of registers are straightforward to understand as they are responsible for directly configuring the output signal of each PWM channel. Every one of these registers that affects the output waveform has both a value register and a separate “update” register. This ensures that glitches on the output are not created since any changes are “buffered” in the update registers, and only change at the end of the channel’s current cycle.

**PWM Comparison x Value Registers (PWM\_CMPV0... PWM\_CMPV7 /**

**PWM\_CMPVUPD0... PWM\_CMPVUPD7):** The values in these registers are used in triggering the two PWM event lines. The counter to which they are compared is always channel 0. Bit 24 in the register determines if the comparison happens while the counter is incrementing or decrementing. Not used for EiE.

**PWM Comparison x Mode / Update Registers (PWM\_CMPM0...PWM\_CMPM7 /**

**PWM\_CMPMUPD0...PWM\_CMPMUPD7):** For each of the 8 comparison units, the comparisons can be enabled or disabled, and the logic can report for 1-16 comparisons. For example, if the system needed to detect every 4th cycle. Not used for EiE.

**PWM Channel Mode Registers (PWM\_CMR0...PWM\_CMR3):** Critical configuration registers to select the clock that will be used for each channel. The 4 LSBs select the input clock. Each channel is configured to either count up and roll back to 0 (left aligned), or count up and then back down (center aligned). The output can start high or low. Center-aligned channels can have events in the middle or just at the end. The “dead-time” generator is also configured here.

For both channels in EiE, MCK/8 clock is used which gives sufficient resolution for all the audio frequencies that need to be generated. This calculation is captured in the code at the start of the PWM configuration section in `eief1-pcb-01.h` so anyone reading the setup values knows why they were chosen.

```
To achieve the full range of audio we want from 100Hz to 20kHz, we must be able to
set periods of 10ms to 50us.
- 10ms at 48MHz clock is 480,000 ticks
- 50us at 48MHz clock is 2400 ticks
- Only 16 bits are available to set the PWM period, so scale the clock by 8:
- 10ms at 6MHz clock is 60,000 ticks
- 50us at 6MHz clock is 300 ticks
```

The waveform is left aligned, starts low, and no dead time is used.

```
#define PWM_CMR0_INIT (u32)0x00000003
#define PWM_CMR1_INIT (u32)0x00000003
```

**PWM Channel Duty Cycle / Update Registers (PWM\_CDTY0...PWM\_CDTY3 /**

**PWM\_CDTYUPD0...PWM\_CDTYUPD3):** The 16 bits that define the high time within the total period (see the PWM\_CPRD registers next). The actual time is this value multiplied by the clock period. For the EiE driver, this will always be half the total period to achieve a 50% duty cycle.

**PWM Channel Period / Update Registers (PWM\_CPRD0...PWM\_CPRD3 /**

**PWM\_CPRDUPD0...PWM\_CPRDUPD3):** The 16 bits that define the total period. The actual time is this value multiplied by the clock period. The resulting frequency is 1 / period. Per the calculations documented above, this value should be between 300 (for 20 kHz high frequency) and 60,000 (for 100Hz low frequency). Important: if the PWM peripheral is already running, then any change in frequency should be applied to the update registers. If the signal is currently off, then the values are written directly to the

PWM\_CPRDx registers.

The registers are initialized to the following:

```
#define PWM_CPRD0_INIT (u32)6000
#define PWM_CPRD1_INIT (u32)1500
#define PWM_CDTY0_INIT (u32)(PWM_CPRD0_INIT << 1)
#define PWM_CDTY1_INIT (u32)(PWM_CPRD1_INIT << 1)
```

**PWM Channel Counter Registers (PWM\_CCNT0...PWM\_CCNT3):** The current count of each channel. The count is reset to 0 when the channel is enabled or when it is in left-aligned mode and reaches the period set in the PWM\_CPRDx registers.

**PWM Channel Dead Time / Update Registers (PWM\_DT0...PWM\_DT3 / PWM\_DTUPD0...PWM\_DTUPD3):** The amount of dead-time (if used). Dead-time is a delay between the end of the duty cycle and the start of the next. This can be set separately for the high and low sides of each channel. A maximum of 12-bits is available.

The PWM peripheral is an excellent example of how comprehensive a microcontroller peripheral can be. Any microcontroller could be programmed to provide these features in firmware, but that would mean the core would be continually executing PWM-related code and consuming power. If the core is involved, there is always the chance of time glitches due to interrupts or other code that requires processor time and/or may have bugs or non-deterministic behavior. This could be disastrous for a motor control application.

Having all of this in a peripheral offloads that responsibility from the core and lets it be handled seamlessly in hardware. The nature of these operations is such that hardware can implement it very efficiently. A whole motor control system can be run safely just by the peripheral without ever bothering the core unless an extraordinary event occurs. Looking at it for the first time may be confusing, but if you regularly worked in applications where the principal task was motor control, this would make a lot more sense.

## 8.7 • Development Board Audio Driver

The goal is to have an easy way to use the PWM peripheral to send a tone (square wave of a certain frequency) to the piezoelectric buzzers on the development board.



To do this, we need to set up the PWM peripheral and write a few API functions to allow any application in the system to use audio easily. INIT values for the PWM peripheral are described above or in `eief1-pcb-01.h` in the “PWM” section near the end of the file. Search for the bookmark, \$\$\$\$.

The audio application we want to do is very simple and very tied to the hardware, so the code will be written in `eief1-pcb-01.c`. No state machine is required. The interface should allow the tone frequency to be specified and turned on and off. We do not need to write this as an application since the PWM peripheral pretty much takes care of all the work. An application might be necessary if you wanted to expand the audio capability to play music or allow priority access to the buzzer.

### 8.7.1 • Audio function initialization



The required configuration for the PWM peripheral should be written in a protected function called `PWMSetupAudio()` that lives in `eief1-pcb-01.c`. Write the `PWMC_CLK_INIT` values and then set each channel to the default values that were given above.



```

/*!-----
@fn void PWMSetupAudio(void)

@brief Configures the PWM peripheral for audio operation on H0 and H1 channels.

Requires:
- Peripheral resources not used for any other function.

Promises:
- PWM is configured for PWM mode and currently off.

*/
void PWMSetupAudio(void)
{
    /* Set all initialization values */
    AT91C_BASE_PPMC->PPMC_CLK = PWM_CLK_INIT;

    AT91C_BASE_PPMC_CH0->PPMC_CMR      = PWM_CMR0_INIT;
    AT91C_BASE_PPMC_CH0->PPMC_CPRDR    = PWM_CPRD0_INIT; /* Set current frequency */
    AT91C_BASE_PPMC_CH0->PPMC_CPRDUPDR = PWM_CPRD0_INIT; /* Latch CPRD values */
    AT91C_BASE_PPMC_CH0->PPMC_CDTYR    = PWM_CDTY0_INIT; /* Set 50% duty */
    AT91C_BASE_PPMC_CH0->PPMC_CDTYUPDR = PWM_CDTY0_INIT; /* Latch CDTY values */

    AT91C_BASE_PPMC_CH1->PPMC_CMR      = PWM_CMR1_INIT;
    AT91C_BASE_PPMC_CH1->PPMC_CPRDR    = PWM_CPRD1_INIT; /* Set current frequency */
    AT91C_BASE_PPMC_CH1->PPMC_CPRDUPDR = PWM_CPRD1_INIT; /* Latch CPRD values */
    AT91C_BASE_PPMC_CH1->PPMC_CDTYR    = PWM_CDTY1_INIT; /* Set 50% duty */
    AT91C_BASE_PPMC_CH1->PPMC_CDTYUPDR = PWM_CDTY1_INIT; /* Latch CDTY values */

} /* end PWMSetupAudio() */

```

The current / updated registers are necessary to assure glitch-less updates in the actual PWM output. In a piezoelectric buzzer audio application, glitches will sound like “clicks” on the buzzer and even tiny glitches are audible. In applications where the PWM peripheral is responsible for driving a large motor, serious damage could be done to the motor or the equipment it is attached to if the driving waveforms are not perfect.



Add the call to `PWMSetupAudio()` just after the `GpioSetup()` call in `main.c` since these are very low-level configurations. We may also want to add some audio feedback in some of the other initializations functions, so the PWM config needs to be done early in the startup sequence.

```

37 void main(void)
38 {
39     /* Low level initialization */
40     WatchDogSetup();
41     ClockSetup();
42     GpioSetup();
43     PWMSetupAudio();
44     InterruptSetup();
45     SysTickSetup();

```

Figure 8-7 Call to `PWMSetupAudio()` function



### 8.7.2 • Audio API Functions

The API should provide channel-specific functions to set the frequency and turn the buzzers on and off. To do this, the firmware functionality will be captured in 3 public functions in `eief1-pcb-01.c`.

- **`void PWMAudioSetFrequency(BuzzerChannelType eChannel_, u16 u16Frequency_)`**: configure the PWM peripheral for the desired audio frequency on the specified channel.
- **`void PWMAudioOn(BuzzerChannelType eChannel_)`**: activate the PWM peripheral for the indicated channel.
- **`void PWMAudioOff(BuzzerChannelType eChannel_)`**: deactivate the PWM peripheral for the indicated channel.

The “Channel” is managed by a typedef that lines up as an ID bit within certain PWM peripheral registers. These bits are named “AT91C\_PWMC\_CHID0” and “AT91C\_PWMC\_CHID1” in `AT91SAM3UH.h` which is rather cryptic. Therefore, add the `BuzzerChannelType` to the board header file:

```
/*-----
%BUZZER% Buzzer Configuration
-----*/
/*!
@enum BuzzerChannelType
@brief Logical names for buzzers in the system.

These definitions correspond to the Channel ID in the PWM peripheral
*/
typedef enum {BUZZER1 = AT91C_PWMC_CHID0, BUZZER2=AT91C_PWMC_CHID1} BuzzerChannelType;
```

#### PWMAudioSetFrequency()

The ability to set the tone frequency with a single function is powerful and pretty much a must-have for any useful API to an audio driver like this. You do not want the user to have to calculate a period value based on how the peripheral works. The function declaration and documentation are as follows:

```
/*!-----
@fn void PWMAudioSetFrequency(BuzzerChannelType eChannel_, u16 u16Frequency_)

@brief Configures the PWM peripheral with the desired frequency on the specified channel.

If the buzzer is already on, it will change frequency (essentially) immediately.
If it is not on, the new frequency will be audible next time PWMAudioOn() is called.

Example:
PWMAudioSetFrequency(BUZZER1, 1000);

Requires:
- The PWM peripheral is correctly configured for the current processor clock speed.
- CPRE_CLKK is the clock frequency for the PWM peripheral

@param eChannel_ is the channel of interest and corresponds to the channel bit
position of the buzzer in the PWM peripheral
@param u16Frequency_ is in Hertz and should be in the range 100 - 20,000 since
that is the audible range. Higher and lower frequencies are allowed, though.
```

**Promises:**

- The frequency and duty cycle values for the requested channel are calculated and then latched to their respective update registers (CPRDUPDR, CDTYUPDR)
  - If the channel is not valid, nothing happens
- \*/

The desired frequency value, `u16Frequency_`, passed to the function is in Hz. Therefore, the period register value is calculated by:

$$\text{Period} = \frac{\text{PWM peripheral clock speed}}{u16Frequency}$$

From the setup values in `eief1-pcb-01.h`, we have a clock frequency constant `CPRE_CLK` which is the clock value used by the PWM peripheral. By referencing the `CPRE_CLK` symbol instead of hard-coding it, the code will automatically update the calculation if `CPRE_CLK` is changed. This line of code calculates the period value that needs to be loaded to the peripheral:

```
u32ChannelPeriod = CPRE_CLK / u16Frequency_;
```

Duty cycle for audio is 50%, so the duty cycle register is just half of the full period (division by 2 is accomplished most efficiently with a simple right bit-shift).

```
psChannelAddress->PWMC_CDTYUPDR = u32ChannelPeriod >> 1;
```

The function must also check to see if the channel is active or not, as that will determine if the current or latch registers are adjusted. If a channel is active, its channel status bit will be set in `PWMC_SR`, so the check is:

```
if (AT91C_BASE_PWMC->PWMC_SR & eChannel_)
```

Lastly, the correct channel base address must be determined depending on `eChannel_`. For efficiency, this value could be determined and captured in a variable, then referenced in the code as a pointer. This is very similar to how functions were generically written for the LED and button drivers.

In the PWM case, the pointer must point to the whole `AT91PS_PWMC_CH` struct so different members in the struct can be referenced. A variable of the struct type is created, and the channel address is assigned with a switch statement. If a channel that does not exist is specified, the function will return without doing anything.



Add the following code.

```
AT91PS_PWMC_CH psChannelAddress;

/* Get the base address of the channel */
switch (eChannel_)
{
    case BUZZER1:
    {
        psChannelAddress = AT91C_BASE_PWMC_CH0;
        break;
    }


    case BUZZER2:
    {
        psChannelAddress = AT91C_BASE_PWMC_CH1;
```

```

    break;
}

default:
{
    /* Invalid channel */
    return;
}
}

```


 With the correct base address in `psChannelAddress`, the rest of the function is straightforward. First, calculate the `u32ChannelPeriod` value. Then check if the channel is active and set the appropriate channel period and channel duty cycle registers based on `u32ChannelPeriod`. If the beeper is already running, use the update registers. If the beeper is not currently running, write the period and duty registers directly.

```

/* Calculate the period based on the requested frequency.
The duty cycle is this value divided by 2 (right shift 1) */
u32ChannelPeriod = CPRE_CLK / u16Frequency_;

/* Set different registers depending on if PWM is already running */
if (AT91C_BASE_PWMC->PWC_SR & eChannel_)
{
    /* Beeper is already running, so use update registers */
    psChannelAddress->PWC_CPRDUPDR = u32ChannelPeriod;
    psChannelAddress->PWC_CDTYUPDR = u32ChannelPeriod >> 1;
}
else
{
    /* Beeper is off, so use direct registers */
    psChannelAddress->PWC_CPRDR = u32ChannelPeriod;
    psChannelAddress->PWC_CDTYR = u32ChannelPeriod >> 1;
}

```


 Build the code to make sure everything is correct. When coding pointers like this, it is a good idea to run the code in the debugger to make sure the correct address is being derived and applied in the code. Call the function from `UserApp1Initialize` and follow it through.

```

91 void UserApp1Initialize(void)
92 {
93     PWMAudioSetFrequency(BUZZER1, 1000);


115     case BUZZER1:
116     {
117         pChannelAddress = AT91C_BASE_PWMC_CH0;
118         break;

```



Expression	Value	Location
pChannelAddress	0x4008C200	R3
PWMC_CMR	3	0x4008C200
PWMC_CDTYR	12000	0x4008C204
PWMC_CDTYUPDR	0	0x4008C208
PWMC_CPRDR	6000	0x4008C20C
PWMC_CPRDUPDR	0	0x4008C210
PWMC_CCNT	0	0x4008C214

**Figure 8-8 Verify BUZZER1 update function**

 Step through and verify that the “Beeper is off” case is selected and then the `CPRDR` and `CDTYR` registers are updated. `CDTYR` is half the value of `CPRDR` as expected. Also, note that the frequency is correct since the PWM clock is  $48\text{MHz} / 8 = 6\text{MHz}$ . 6 million divided by 6000 is 1000 which is the specified frequency.

```

145 else
146 {
147     /* Beeper is off, so use direct registers */
148     pChannelAddress->PWMC_CPRDR = u32ChannelPeriod;
149     pChannelAddress->PWMC_CDTYR = u32ChannelPeriod >> 1;
150 }

```



Watch 4	
Expression	Value
pChannelAddress	0x4008C200
PWMC_CMR	3
PWMC_CDTYR	3000
PWMC_CDTYUPDR	0
PWMC_CPRDR	6000

Figure 8-9 'Beeper is off' function

### 8.7.3 • PWMAudioOn() and PWMAudioOff()

Turning the channel on and off is simple. Both functions take the BuzzerChannelType channel of interest as the only parameter. Write this value to AT91C\_BASE\_PWMC->PWMC\_ENA to turn the buzzer on. Write it to AT91C\_BASE\_PWMC->PWMC\_DIS to turn the buzzer off. Make sure to cast the buzzer channel to u32 when loading it.



Add both the “on” and “off” cases to the code after PWMAudioSetFrequency. The “on” case is shown.

```

void PWMAudioOn(BuzzerChannelType eBuzzerChannel_)
{
    /* Enable the channel */
    AT91C_BASE_PWMC->PWMC_ENA = (u32)eBuzzerChannel_;
} /* end PWMAudioOn() */

```



Test the code by adding initialization to the BUZZER2 to 200Hz below the line of code that initializes BUZZER1 to 1000Hz. In UserApp1SM\_Idle, turn on BUZZER1 if BUTTON1 was pressed and turn on BUZZER2 if BUTTON2 was pressed. Turn off both buzzers if BUTTON3 was pressed.

```

static void UserApp1SM_Idle(void)
{
    /* BUZZER1 is on if BUTTON1 was pressed */
    if(WasButtonPressed(BUTTON1))
    {
        ButtonAcknowledge(BUTTON1);
        PWMAudioOn(BUZZER1);
    }

    /* BUZZER2 is on if BUTTON2 was pressed */
    if(WasButtonPressed(BUTTON2))
    {
        ButtonAcknowledge(BUTTON2);
        PWMAudioOn(BUZZER2);
    }

    /* Both buzzers off if BUTTON3 was pressed */
    if(WasButtonPressed(BUTTON3))
    {
        ButtonAcknowledge(BUTTON3);
        PWMAudioOff(BUZZER1);
        PWMAudioOff(BUZZER2);
    }
}

} /* end UserApp1SM_Idle() */

```

Build and test to make sure you hear noise out of both buzzers. There should be no audible glitches if you try to turn on a buzzer that is already on.



Piezo buzzers will theoretically be louder with a higher duty cycle. If you want another challenge, add a feature to the API to set the volume level. To ensure backward compatibility, write a separate API function called `PWMSetVolume()` that takes parameters for the channel and an enum `ePWMVolume = {VOLUME_LOW, VOLUME_MID, VOLUME_HIGH}` that corresponds to 25, 50, and 75% duty cycles. The function should then modify the existing values and possibly flag a “current volume” local global that would maintain the volume if the frequency changed. We have not tried this, so the actual difference in volume between LOW and HIGH may not be noticeable.

## 8.8 • PWM the Hard Way: Bit Bashing

A great way to really understand a concept is to work through an exercise that forces you to explicitly implement every part of functionality related to it. Generating a square wave on a random output pin is not difficult if a small amount of jitter is tolerable. Jitter is inconsistency in the period of a signal and typically used in the context of clock signals but can be applied to the problem of bit-bashing PWM signals.

Bit-bashing may also be necessary when a peripheral is not available or there are not enough of a peripheral type to meet all the requirements of a circuit. It is often impractical to specify a bigger or better processor just to get an extra peripheral. In some cases, a processor with everything on your wish list simply does not exist. Adding features to existing hardware may also negate the option of choosing a processor that has what you need. This is another example of how carefully specifying a processor in a new design requires a lot of thinking about future requirements.

The biggest problem of bit-bashing is the overhead required to program each part of the function. This becomes increasingly difficult as the speed or complexity of whatever signal you are trying to create increases. For example, bit bashing a communication protocol that transmits and receives at MHz speeds is very challenging if the processor is running at only 12MHz. In this case, you might be stuck writing assembler routines to get a very efficient algorithm, and even then, you might not be able to do it! Creating a PWM signal requires almost no overhead, and it is identical each period so should be no problem.

The manual PWM will be used to upgrade the LED driver that we have already written for the development board to allow dimming of the lights via PWM. We have to do this the hard way because the processor does not have enough PWM outputs to drive each LED (it would be very unlikely that you would find a processor that would have eleven PWMs). Dimming of the discrete LEDs will allow for some nice effects, but the main purpose is to have dimming control for the LCD backlight LEDs. Since the backlight has red, green and blue lights (denoted “RGB”), adding them together with different levels of intensity will allow you to form any color you want!

This is exactly how LED displays work – if you look closely, you will see groups of RGB LEDs that are simply dimmed to a level to create the desired color just like mixing paint in art class. If each color has 256 levels of intensity (in other words, 8 bits to control the level), then you can achieve  $256 \times 256 \times 256 = 16.7$  million colors! Not that your eye can distinguish that many different colors, but as you have or will no doubt experience when shopping for monitors or displays, “24bit” or higher color is a major selling feature that you will pay a bunch of money for (note that no one really knows for sure how many colors you can resolve, and the arguments tend to be more philosophical than logical – opinions have it ranging anywhere from a few thousand to several million). By the way, if you have equal parts red, green and blue, the result is white since mixing light is an “additive” process. This is also why the primary light colors are red, blue and green instead of red, blue, and yellow for “subtractive” processes like painting.

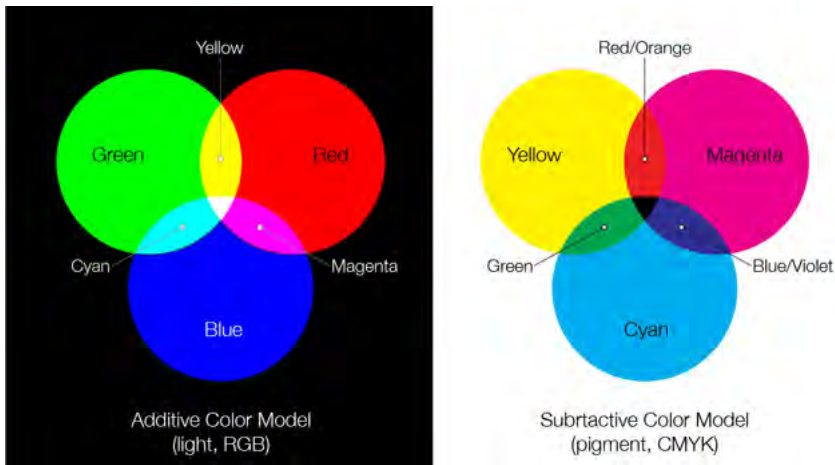


Figure 8-10 Additive (left) and subtractive (right) colour processes

### 8.9 • LED PWM Design

The most difficult part about bit-bashing a periodic signal is solving the jitter problem and accurately toggling the output regardless of what else is happening in the system. The 1ms loop period in the EiE system already solves this by providing a “good enough” timing base on which to manually generate the PWM signal. Because there are other tasks in the system, there is a chance that the exact timing of the function call to toggle the output will vary slightly within the loop. This can be mitigated by calling the LED task first in the loop, though for this application it doesn’t really matter as jitter on the order of less than 1ms will be undetectable to the viewer.

What will matter is if any tasks violate the 1ms rule and therefore stretch out the current period of an LED PWM cycle. If the LED is currently off, this will appear as a dark glitch. If the LED is currently on, it will momentarily flash brightly. Just like small glitches in audio that will manifest themselves as noticeable audible clicks, even very short period disruptions in the PWM output to the LEDs will be noticeable. If they only happen once every few minutes or when a certain event is occurring, it doesn’t really matter.

If a task in the system is consistently disrupting the 1ms loop time, it will be obvious for PWMing LEDs. In designing the LED PWM driver, the system rules and the resources that are available must be considered. It happens that this will work quite nicely. The defining factor is the PWM duty cycle resolution of the LEDs. As discussed earlier in the chapter, the frequency of the PWM driving signal for an LED must be fast enough to trick your eye into seeing a steady, dimmed LED instead of a flickering light. The minimum frequency for this is around 30Hz which gives a period of 33.3ms. Keep that in mind.

The second consideration is how much PWM resolution we need/want in the system. For example, if you wanted 1% duty cycle control, you would need to have a total PWM period of 100 “units” so that you could adjust the duty cycle in 1 unit increments thereby achieving 1% duty cycle steps. Therefore, each LED could have 100 levels of brightness and you could have  $100^3 = 1$  million different colors. The problem is that we need 100 cycles available to manage that amount of resolution but of course, we only have 1ms periods/cycles to work with. That would mean we could only achieve 10Hz frequency for the LEDs and guaranteed you would see them flicker.

To get around this, we either move away from the 1ms “clock,” or we admit that we really do not need a million backlight colors! If higher resolution was necessary, you would have to either speed up the 1ms system tick or use another timer to provide a faster interrupt to dedicate to the PWM. However, higher resolution is NOT necessary, so we proceed with our 1ms system intact.

The maximum output frequency that can be generated comes from toggling the output every cycle. Toggling every 1ms results in a period of 500Hz but that would only allow 50% duty cycle. To make numbers work out nicely, let us choose to generate the PWM signal at 50Hz which equates to 20ms periods for the PWM. If we have 20ms to work with at 1ms resolutions, this gives 5% duty cycle resolution and results in  $20^3 = 8000$  colors. That is plenty to create a very nice display as you will see once you have completed the chapter exercise. Note that we could probably get away with 25Hz and thus 40 cycles to work with, but it is likely that some people would notice a flicker in the slower signal.

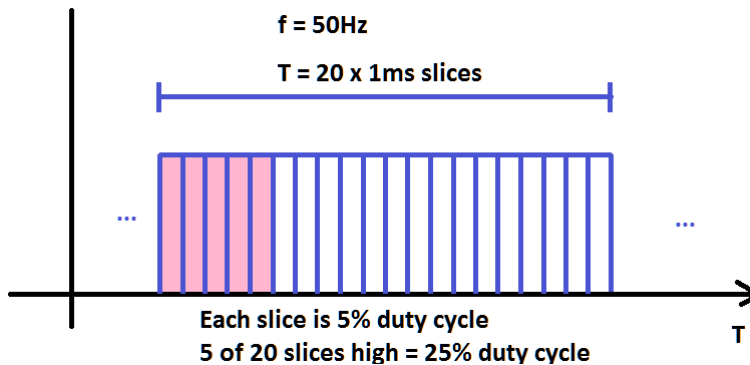




Figure 8-11 PWM duty cycle plan

With those decisions made, writing the driver function is almost easy. The LED task already blinks the LEDs and PWMing them is very similar to the blinking case except it happens much faster and the duty cycle is variable.

 Make sure the `pwm_audio` branch is loaded and open `leds.h`. The `LedModeType` enum currently has “LED\_NORMAL\_MODE” and “LED\_BLINK\_MODE.” Add “LED\_PWM\_MODE” to the typedef.

```
/*!
@enum LedModeType
@brief The mode determines how the task manages the LED */
typedef enum {LED_NORMAL_MODE, LED_BLINK_MODE, LED_PWM_MODE} LedModeType;
```

 Open `leds.c` and add a second case inside the for-loop in `LedSM_Idle()` to check for `LED_PWM_MODE`.

```
/* Check if LED is in LED_PWM_MODE */
if(Led_asControl[(LedNameType)i].eMode == LED_PWM_MODE)
{
} /* end LED_PWM_MODE */
```

The LED task needs to know what duty cycle the LED is set to. During the design, it was decided that the frequency would be 50Hz and thus there are 20 available duty cycles where each is 5% (1ms out of 20ms). These values work almost the same as the existing

LedRateType definitions for LED\_BLINK\_MODE so we can add them to that enum.

```

/*!
@enum LedRateType
@brief Standard blinky values for blinking.

Other blinking rate values may be added as required. The labels are frequencies,
but the values are the toggling period in ms.

*** The PWM rates are set up to allow incrementing and decrementing rates within the allowed
values. Be careful at the edge cases. ***
*/
typedef enum {
    LED_0HZ = 0, LED_0_5HZ = 1000, LED_1HZ = 500, LED_2HZ = 250, LED_4HZ = 125,
    LED_8HZ = 63,
    LED_PWM_0 = 0,    LED_PWM_5 = 1,    LED_PWM_10 = 2,    LED_PWM_15 = 3,    LED_PWM_20 = 4,
    LED_PWM_25 = 5,    LED_PWM_30 = 6,    LED_PWM_35 = 7,    LED_PWM_40 = 8,    LED_PWM_45 = 9,
    LED_PWM_50 = 10,   LED_PWM_55 = 11,   LED_PWM_60 = 12,   LED_PWM_65 = 13,   LED_PWM_70 = 14,
    LED_PWM_75 = 15,   LED_PWM_80 = 16,   LED_PWM_85 = 17,   LED_PWM_90 = 18,   LED_PWM_95 = 19,
    LED_PWM_100 = 20
} LedRateType;

```

The numerical values for the PWM duty cycles are sequential from 0 to 20 which means that we can abuse this enum and increment and decrement the PWM duty cycle to go to the adjacent levels. This was not deliberate but ended up as a useful consequence of the design. Notice the comment “\*\*\* The PWM rates are set up to allow incrementing and decrementing rates within the allowed values. Be careful at the edge cases.” Documenting this “feature” is important so any user of the API knows how to use/abuse it correctly. It is still risky, but this is mitigated and acceptable for this system.

When an LED is in PWM mode, the frequency is always 50Hz which means the period is always 20 cycles of the main loop. The LED\_PWM\_X values represent the “high” portion of the duty cycle, so the “low” portion is always LED\_PWM\_100 – the current LED\_PWM\_X value. The LED driver will need to calculate and track the low time and know if the LED of interest is currently in the high or low part of its cycle.



Define an enum for the current duty cycle high or low time. Code the definition just after LedModeType definition in leds.h.

```

/*!
@enum LedPWMDutyType
@brief Duty cycle state when tracking PWM mode*/
typedef enum {LED_PWM_DUTY_LOW = 0, LED_PWM_DUTY_HIGH = 1} LedPWMDutyType;

```



Add a member of this new type to the LedControlType struct so every LED will have this information.

```

typedef struct
{
    LedModeType eMode;           /*!< @brief Current mode */
    LedRateType eRate;           /*!< @brief Current rate */
    u16 u16Count;                /*!< @brief Value of current duty cycle counter */
    LedPWMDutyType eCurrentDuty; /*!< @brief Phase of the current duty cycle */
}LedControlType;

```





Since `LedControlType` has been modified, its initialization should also be updated in `LedInitialize()`. Add initialization of `eCurrentDuty` in `Led_asControl[]` to `LED_PWM_DUTY_LOW`.

```
/* Initialize the LED control array */
for(u8 i = 0; i < U8_TOTAL_LEDS; i++)
{
    Led_asControl[i].eMode = LED_NORMAL_MODE;
    Led_asControl[i].eRate = LED_0HZ;
    Led_asControl[i].u16Count = 0;
    Led_asControl[i].eCurrentDuty = LED_PWM_DUTY_LOW;
}
```



The API function to set an LED to PWM mode is almost identical to BLINK mode. The LED can always start in the high PWM state, so the `u16Count` member can be initialized to the PWM rate requested and the `eCurrentDuty` should be initialized to `LED_PWM_DUTY_HIGH`.

```
void LedPWM(LedNameType eLED_, LedRateType ePwmRate_)
{
    Led_asControl[(u8)eLED_].eMode = LED_PWM_MODE;
    Led_asControl[(u8)eLED_].eRate = ePwmRate_;
    Led_asControl[(u8)eLED_].u16Count = (u16)ePwmRate_;
    Led_asControl[(u8)eLED_].eCurrentDuty = LED_PWM_DUTY_HIGH;
} /* end LedPWM() */
```



Adding the PWM control in the LED task is a classic example of dealing with a situation that has a typical mode of operation and two edge cases (0% and 100% duty cycles). The edge cases are easy: no cycle tracking is required as the corresponding LED is just totally on or totally off.

```
/* Handle special case of 0% duty cycle */
if( Led_asControl[i].eRate == LED_PWM_0 )
{
    LedOff( (LedNameType)i );
}

/* Handle special case of 100% duty cycle */
else if( Led_asControl[i].eRate == LED_PWM_100 )
{
    LedOn( (LedNameType)i );
}
```

The typical case where the LED task is managing the high and low time follows a basic flow chart.

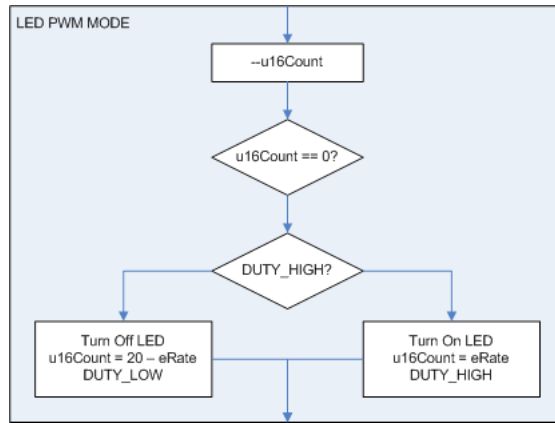


Figure 8-12 LED task flowchart



Write this code as the 3<sup>rd</sup> case after the two edge cases. Our solution is shown for reference.

```

/* Regular PWM: decrement counter; toggle and reload if counter reaches 0 */
else
{
    if(--Led_asControl[(LedNameType)i].u16Count == 0)
    {
        if(Led_asControl[(LedNameType)i].eCurrentDuty == LED_PWM_DUTY_HIGH)
        {
            /* Turn the LED off and update the counters for the next cycle */
            LedOff( (LedNameType)i );
            Led_asControl[(LedNameType)i].u16Count =
                LED_PWM_100 - Led_asControl[(LedNameType)i].eRate;
            Led_asControl[(LedNameType)i].eCurrentDuty = LED_PWM_DUTY_LOW;
        }
        else
        {
            /* Turn the LED on and update the counters for the next cycle */
            LedOn( (LedNameType)i );
            Led_asControl[i].u16Count = Led_asControl[i].eRate;
            Led_asControl[i].eCurrentDuty = LED_PWM_DUTY_HIGH;
        }
    }
}
}

```



The last thing to do is something you would likely never remember until you tried the PWM code that was written and it didn't work. Calling LedOn() and LedOff() always sets the LED mode back to LED\_NORMAL\_MODE, so at the end of the PWM section, the LED should always be reset back to LED\_PWM\_MODE.

```

/* Set the LED back to PWM mode since LedOff and LedOn set it to normal mode */
Led_asControl[(LedNameType)i].eMode = LED_PWM_MODE;

} /* end LED_PWM_MODE */

```



Build the code to make sure the compiler is happy. The LED task update is now complete and ready for test. Write a short program in UserApp1 to make the blue LCD backlight LED continually cycle through PWM rates from 0% to 100%. Remember you can increment the eRate value safely through all the duty cycle levels. Set up a constant value


for the time between the changes and start this at 500ms. Don't forget to initialize the Red and Green LED backlights to "Off" or your LCD will instead cycle from yellow to white (why?). Run the code and make sure you can see the discrete changes in brightness. The program is shown below.


```
static void UserApp1SM_Idle(void)
{
    static u16 u16TimeCount = 0;
    static LedRateType eCurrentRate = LED_PWM_0;

    /* Fade the blue LCD backlight on */
    u16TimeCount++;
    if(u16TimeCount == U16_TIME_DELAY_MS)
    {
        u16TimeCount = 0;

        /* Check the current rate and either increment or go back to 0 */
        if(eCurrentRate == LED_PWM_100)
        {
            eCurrentRate = LED_PWM_0;
        }
        else
        {
            eCurrentRate++;
        }

        /* Update the LED PWM signal */
        LedPWM(LCD_BLUE, eCurrentRate);
    }
}
```

 Now change the 500ms value to 100ms and re-run the program. You can still see the discrete level changes, but just barely. Change it again to 40ms and now the duty cycle transitions are fast enough that your eye cannot distinguish them. Your eye is being tricked twice, now! What happens if you use a value that is not a multiple of 20? Try using 50 and explain what you see. Relate this to the PWM peripheral that has "update" registers in addition to the period and duty cycle registers.

 To customize your board, use what you have learned to make LedInitialize a little more interesting. You will see in the next chapter that the EiE firmware has been updated to turn on all LEDs and then fade them out very quickly when the board starts up. Buzzer sounds were also added, though with a #define switch so the sound could be easily turned off as it gets very irritating (it is off by default).

### 8.10 • Audio Bits

The features that have been added to the EiE firmware in this chapter give us a remarkable improvement in what the development board can accomplish. It's like we have taken a purely digital system and added a rich set of analog features even though those "analog" features are just emulated using PWM. Before we show you just what you can do now, we need to give a quick music lesson that might come in handy. There are a lot more details to music beyond this, but this is plenty to get you going.

1. You should fully understand that different musical notes are a function of frequency. The lower the frequency, the lower the sound. Deep bass is sub-100Hz. The highest sound a human can hear is around 16kHz depending on how loud your music was when you were a teenager and thus how sensitive your ears are still. Dogs can hear much higher frequencies.

2. Different speakers are better at reproducing different frequencies (i.e. subwoofers for bass, tweeters for very high sound). All piezoelectric buzzers sound terrible, but they are particularly poor at producing low frequencies.
3. Music is written with notes from A to G which span one “octave”. A note in octave  $n$  is exactly double the frequency of the same note in octave  $n-1$ .
4. A full piano has 88 keys and spans 8 octaves (frequencies from C1 = 32Hz all the way to C8 = 4186Hz). In musical terms, it does not make much sense to talk about sounds beyond those frequencies, though there is theoretically no upper limit. Once you reach G, you restart at A (one octave higher than the last A note). The difference in frequency between notes is not linear.
5. In every octave, you have the main notes and sub-notes. The sub-notes are called “flats” or “sharps.” The “sharp” of a note is the same as the “flat” of the next note (e.g. A-sharp is the same as B-flat). The symbol for “sharp” is # like C# (not to be confused with the programming language, but no doubt someone thought they were being clever when they named it) and the symbol for “flat” is  $\flat$  like A $\flat$ .
6. On a piano, the main notes are the white keys and the sub-notes are the black keys, so there are actually 11 keys per octave: A, A-sharp (B-flat), B, C, C-sharp (D-flat), D, D-sharp (E-flat), E, F, F-sharp (G-flat), G, G-sharp (A-flat of the next octave).
7. The frequencies of notes are relative and mathematically defined based on a reference value. There are tons of online resources if you want to learn more.
8. Music is written by defining single notes or combinations of notes that are played for a certain duration. The notes are shown on a five-line “staff” and are written on either a line or in a space. Staves are denoted “treble” (the ampersand-like symbol is called a “treble clef) and “bass” (the comma-like symbol is called the “bass clef”). The bottom line in the “treble” staff starts at E and the top line is F. A memory tool is “Every Good Boy Does Fine.” The spaces are F-A-C-E. The bass staff lines are from G to A, or, “Good Boys Do Fine Always.” The bass staff spaces are A to G, or, “All Cows Eat Grass”

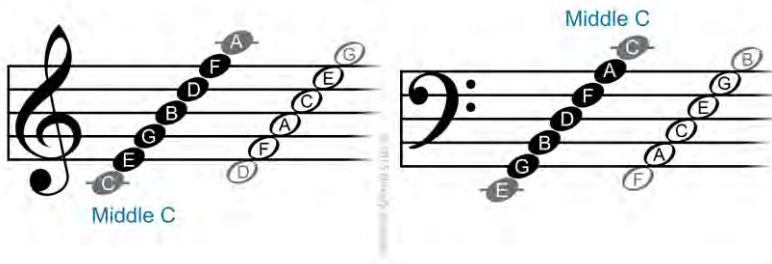


Figure 8-13 Musical notation

9. If any note is to be played as a flat or sharp, there is an indicator next to the treble or bass clef. An example is the sheet music for “Mary had a little lamb” which has one sharp (F).

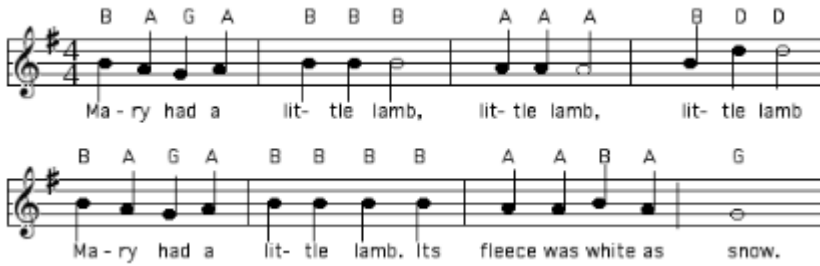



Figure 8-14 Music for “Mary had a little lamb”

There are some great ways to program algorithms that will play simple single-note songs over a piezo buzzer. Essentially you write two arrays: one with the sequence of notes, and one with a corresponding duration to play each of those notes. There are even some standard algorithms for computer generated music that you could program and then download source files with various songs. There are several examples in the EiE library of some songs that sound pretty decent.

 Add music.h from `\\..\\firmware_common\\application` into the project and look at the file. There are 4 octaves of notes defined that include all values that sound ok on the piezo buzzers. It is unlikely that any song you want the board to play will need values outside of this range.

### 8.11 • Multiple User Tasks

In preparation for the Chapter exercise, it is important to remember and make use of the fact that the EiE firmware system is a multi-tasking system. So far, the example programs have been written in UserApp1 and have focused on demonstrating one particular function or API. Don't forget that there are several other tasks running in the system, and several more will still be added. Every task may have multiple states – do not confuse the two. The terms “task,” “object,” and “application” are synonymous.

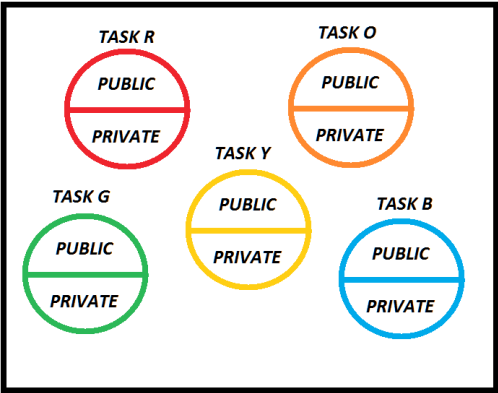


Figure 8-15 Objects/Tasks/Application in system

Each task can be treated like an independent application just like programs running in an operating system. This drastically simplifies making the EiE development board do different things that are inherently unrelated. The only thing the programmer needs to worry about is sharing resources. If two tasks both want to use the blue LED to

represent something in their respective programs, then there will be a problem. This can be avoided in various ways including simply stating that all tasks will have knowledge of the resources that are in use, or using semaphores (locks) that allow tasks to claim a resource and block any other task from using that resources until it is unlocked.

The next few chapters deal with resources that are more complicated than LEDs and buttons, so you will see some techniques to either lock off a resource to prevent conflicts, or offer an API that facilitates a multi-tasking system.

The other issue with running multiple tasks is passing information between those tasks if they do have some dependencies. The first thing to consider is if this is necessary. It is possible that two tasks that need to share information could be designed as a single task and therefore freely share local (private) data. If that's not an option, a mechanism needs to be in place to move data between the two tasks. This ends up being the similar problem of sharing private member data between two objects in OO programming.

In resource-limited embedded systems, data is often shared through global variables. High-level programmers who haven't worked with small embedded systems tend to lose sleep over this idea. Global variables are frowned upon for many reasons and are a bit of a hack when it comes to using them to solve data sharing problems. However, it may simply be impractical to share data in small embedded systems in any other way. There's a balance here and so it is very important to carefully think about and document the solution you choose.

The EiE platform is not exactly resource-limited unless hundreds or thousands of bytes need to be shared. Using callback functions as a method of connecting independent pieces of code was already introduced in the last chapter, though this is quite specific to interrupt service routines. However, the concept of sharing data between tasks is very similar. If any private data needs to be exposed to other tasks, a public member function can be offered to access the data. Specifically, the function provides a copy of the data so that the original data stays intact. The downside of this is the extra coding it takes to write the function, and the extra memory and processor resources it takes to make a copy of the data.

If private data needs to be modified by an external task, an API function can also be made available to do this safely. The function allows the task to verify the data, and update the private data when and how the task wants.

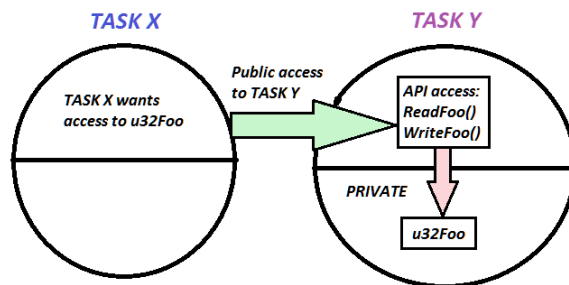


Figure 8-16 API function to access private data



You will find UserApp2 and UserApp3 in the `\\.\firmware_common\application` folder. Add these into the project now. With three User apps available, you have lots of space set up for multiple tasks. If you want more tasks, follow the instructions at the top of the UserApp source files to make additional tasks. You can make as many as you want, but keep in mind the system as a whole must still follow the rules.

**8.12 • Chapter Exercise**

The chapter exercise is in two parts that test the new audio and LED functionality separately. You should complete each part in a separate task. Both tasks must run at the same time when you are complete.

1. Create a 4-note synthesizer where the notes G4, A4, B4, and D5 will play when BUTTON0 thru BUTTON3, respectively, are pressed. Prove that your program works by playing “Mary had a little lamb” (the sheet music is shown above). It does not have to be perfect, for example, do not worry about what happens when two buttons are pressed – assume that only one button will be pressed at the same time. Hint: every time you call PWMAudioOn, the current PWM cycle is reset so if you call it repeatedly, you will not get the correct tone.
2. Color cycle the RGB backlight. Write a tight piece of code to color cycle the three RGB LEDs so they mix together and show off most of the 8000 colors available. Though you could write this brute force, take some time to design a solution that is code efficient. Make sure the PWM rates for each LED to cycle through a rainbow of colors in the correct ROYGBIV order as follows:
  - a. Red LED ramps up from 0 to 100%
  - b. Green LED ramps up from 0 to 100%
  - c. Red LED ramps down from 100 to 0%
  - d. Blue LED ramps up from 0 to 100%
  - e. Green LED ramps down from 100 to 0%
  - f. Red LED ramps up from 0 to 100%
  - g. Blue LED ramps down from 100 to 0%
  - h. Repeat from b.

Remember that there is probably a ton of different ways to program this solution. The code space required for our implementation is  $276 + 19 = 295$  bytes and uses 26 bytes of RAM. Can you do better?

342	Module	ro code	ro data	rw data
343	-----	-----	-----	-----
360	user_app.o	276	19	26

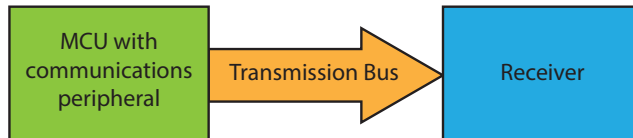
**i** *Closing thought: You could port this code to a purpose-built PCB and create a very neat night light or other great lighting effect in your home or car. Color-cycling RGB LED lighting is popping up all over the place from small electronics to lighting up entire buildings. This is a great example of how the learning you do with a development board can be applied to a real-life product!*

## Chapter 9 • DMA and Messaging

Now we switch gears and start designing an extremely important part of an embedded system: communications. This chapter raises the bar significantly in terms of design and programming complexity. Nothing we do in this chapter will be visible with any added functionality on the development board yet, but it gets us ready for the following chapters where a LOT of new capabilities will be added. Consider this an important introduction to the pieces that will be used to solve a larger puzzle. You should also notice that a lot of the discussion in this chapter challenges you to start thinking about the consequences of trade-offs of your design choices. These are the true challenges in embedded design engineering.

### 9.1 • Data Transmission

It is safe to say that most embedded systems will require the microcontroller to send and receive data. The data may go to an IC on the same circuit board, or to a totally different digital system that is connected. Regardless of the application, sending data means taking the data from somewhere in memory and transmitting over a physical medium following a defined protocol to a receiver.



**Figure 9-1 Data transmission**

In a typical embedded system, data is always moving. Even in the EiE system, data is continually moving inside the processor: to the LCD, to and from the ANT radio, debug port, SD card, and USB, and possibly in and out of the daughter board connection. Most of the data is serial, and it can move very fast. The connection between the SAM3U2 and the nRF51422 processor uses a 2MHz clock. The SD card is even faster.

When the data is ready to come out of the processor, transmitting between two different devices requires an agreed upon standard for how it will happen. These are called protocols and define the physical and logical interface between the two devices. There is a myriad of communication protocols that have been developed for digital communications. The list below is by no means complete but does highlight some of the most common protocols in the embedded world.

- |                                      |          |
|--------------------------------------|----------|
| • RS-232                             | • CANBUS |
| • RS-422                             | • LINBUS |
| • RS-485                             | • MODBUS |
| • SPI                                | • USB    |
| • I <sup>2</sup> C / IIC / I2C / TWI | • IRDA   |

Each of the above has a defined signaling that must take place to ensure that both sides of the connection communicate properly. Some of these protocols are “synchronous” where a clock signal is sent. Others are “asynchronous” where the clock is either previously known somehow or derived from the data. The relationship between the



devices may follow a Master-Slave model or use hardware handshaking to coordinate. Some are strictly point-to-point, and some are bus-based where many devices are connected via shared wires. Communications can also be full duplex with simultaneous or completely independent send and receive channels, or half-duplex where one device is sending while the other is receiving. In most cases, data is sent serially.

For any communication to work, the data must exist somewhere in the transmitter, be organized or formatted into the protocol that will be used to send it, sent, received, and put back together and stored. In many cases, and certainly in the context of this and the next two chapters, the protocol portion of the data transmitting problem will be solved by using the correct communications peripheral.

Communications peripherals on microcontrollers typically handle one byte at a time. That means that the processor must provide the byte and wait for it to be sent before providing the next byte. It's a little bit like a group of skydivers waiting in a plane to jump out. Everyone lines up and waits their turn. It takes a little bit of time for each person to climb out on the wing, get ready, and let go. The jump Master is like the peripheral that is monitoring the "data transmission." When the coast is clear, he signals the next person to go.

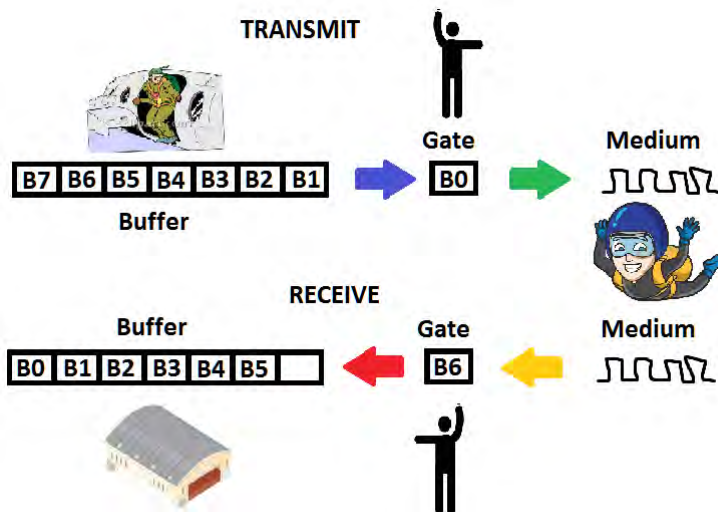


Figure 9-2 Data transmission - skydiver analogy

It works the same on the receive side. One byte comes in at a time, the peripheral tells the processor that a new byte has arrived, and the processor puts the byte somewhere before the next byte arrives. The skydiver analogy still applies as the skydivers land, a person on the ground will tell them to go to a particular location so they are out of the way of other skydivers landing. In both cases, you have a buffer and a gate, with a transmission medium in between. Transmitting and receiving share many of the same concepts but receiving is slightly more difficult. Developing a good foundation for transmitting is our focus here.

In a very basic embedded system that is not multitasking and could focus exclusively on sending data when it needed to be sent, it is quite easy to fill up a RAM buffer with a known number of bytes to send, then wait in a while loop like this:

```
while(u8BytesToSend != 0)
{
    Queue byte to peripheral send register;
    Wait for peripheral to indicate byte is sent;
    u8BytesToSend--;
}
```

Even if two different functions needed to use the same peripheral to send data, the first function runs to completion before the second function runs so there is no resource conflict. Of course, while this is happening, the processor is blocked from doing anything else. In some systems that's fine, in the EiE firmware or any microcontroller running an operating system, it would be a problem.

## 9.2 • Resource Conflicts

This is a good segue into solving a common problem in a multi-tasking system: resource sharing. A similar approach, as shown above, could be used to make the code work in the existing EiE system with only a slight adjustment so the processor is not blocked. Let's assume that the task always sends 100-byte data packets and that sending 1 byte occurs in less than 1ms.

```
void DataSenderSM_Transmit(void)
{
    static u8BytesToSend = 100;
    if(u8BytesToSend != 0)
    {
        Queue byte to peripheral send register;
        Wait for peripheral to indicate byte is sent;
        u8BytesToSend--;
    }
    else
    {
        u8BytesToSend = 100;
        pfDataSenderSM = DataSenderSM_WaitForNextPacket;
    }
}
```

There are still two issues with such a simple model. First, the maximum data speed is now only 1kHz since the Transmit state is called only once every millisecond. If the protocol being used was synchronous and did not depend on any handshaking or ready-status of the receiver, the code could be modified to send several bytes per loop (maybe 500us worth or whatever was deemed appropriate). Even asynchronous systems like RS-232 could survive this since the clock at the receiver is usually triggered by a start-bit so it doesn't matter if the time between bytes is not consistent (not true for all asynchronous systems). If the data packets were always small, then this might work successfully most of the time in a single cycle, but that's not a very robust design.

The second more significant problem is that other tasks in the system that have no knowledge of the DataSender task could try to use the same peripheral resource when they get processor time. This would cause a conflict with the peripheral since it could be busy sending a byte while the other task was trying to give it another byte to send. Or the bytes might go to the wrong location or break the protocol as one task tries to start a new communication sequence.

This problem can be solved by using what is called a "semaphore." A semaphore is just a mechanism to somehow "check-out" or lock a resource. In the simplest terms, a binary semaphore can be a Boolean variable that tracks if the resource is busy or not. If a

system must share resources like a communication peripheral, tasks wishing to use the peripheral would have to check the “busy” variable. If it’s busy, the task waits. If it’s not busy, the variable is changed to busy and the task uses the resource. This could be done with an API function that offers some sort of request service for the resource. The pseudo code could look like this:

```
bool DataSenderRequest(void)
{
    /* Make sure DataSender (static local-global variable) is not busy */
    if(DataSender_bBusy == FALSE)
    {
        /* Not busy, so change to busy and tell the calling task that it now has
        the resource */
        DataSender_bBusy = TRUE;
        return TRUE;
    }
    else
    {
        /* Busy, so tell the task that it may not have the resource */
        return FALSE;
    }
}
```

There is always the risk that due to timing some tasks will never (or rarely) get access to the resource they need. In that case, instead of a binary semaphore, a counting semaphore can be used which essentially lets tasks “take a number” any time they want to send to ensure they have a place in line to eventually access the resource. “Your data is important to us, please stay on the line.” There are always ways to solve problems that come up.

In any case, it is important that the resource is released when the task is finished by changing the semaphore back to the “not busy” state.

### 9.3 • Direct Memory Access – DMA

We know that we have data that needs to be sent and received, transmission will follow a protocol, and there are processor peripherals available to do this but need to be managed to work with only one message at a time. These are much more specific and manageable pieces to work with.

Any time data moves through a microcontroller, it is typically retrieved by the core from flash, RAM or a peripheral memory location, brought into a local core register, and then moved out to its new location in RAM or a peripheral. A good example is the simple text input and output functionality to read and write bytes between an embedded system and a computer terminal program. Data sent from the embedded system could be a string in RAM that is addressed by the core byte-by-byte where each byte is brought to a core register and then sent to a serial peripheral called a UART. The byte is placed in the UART’s transmit register which triggers the peripheral to send it bit-by-bit to the computer. Receiving data is identical but in reverse.

The process is straightforward, but it consumes a lot of resources when you look at it on a per-byte basis. The core cannot do anything else while it is moving data around. If only a few bytes need to be moved, then it is not a big deal. However, embedded systems are often moving millions of bytes all the time, so efficiency becomes important especially with battery-powered devices.

Many microcontrollers will build larger buffers into their peripherals, for example, 16 bytes. That way, a decent amount of data can come into the peripheral before the core

is interrupted to unload all 16 bytes at once rather than waiting for each byte to send and then loading the next. The process of transferring multiple bytes of sequential data is usually more efficient vs. transferring single bytes because you only have the overhead of setting up the data once as a group instead of with each byte. The ARM instruction set even has specific instructions defined to load and store multiple bytes to minimize the number of processor cycles required.

Efficiency can further be improved with interrupts that alert the core when a peripheral needs more data or has data that needs to be read. Unfortunately, the core still must dedicate itself to moving those bytes through its registers.

With that in mind, the concept of direct memory access (DMA) practically jumps out. If pointers can be set up for source and destination addresses for a sequential array of bytes (e.g. any normal buffer or data array), and RAM and peripheral memory are all connected on the same bus structure inside the processor, then surely a bit of logic could be set up to move a known number of bytes between those pointers?

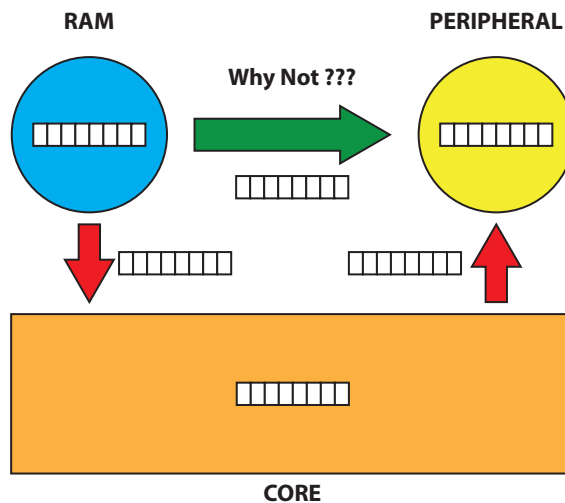


Figure 9-3 Direct Memory Access (DMA)

Some cores advertise memory-to-memory transfer capability which is very close to the same thing. DMA is perhaps a step further by providing multi-byte capability that is completely independent of the core once the transfer is set up and initiated. Generally, all you need is to set the source and destination pointers and let the DMA controller know how many bytes to move. Interrupts are used to flag the core once the transfer is done, so you can put the processor to sleep and still be sending hundreds of bytes around the peripherals and memory.

The SAM3Ux processor has a very powerful DMA system that has two different parts. The first – which we will not cover here beyond a brief mention – is the main DMA controller or DMAC. The purpose of the DMAC is to allow direct data transfer between combinations of peripherals and memory. The DMAC is described in the “DMA Controller (DMAC)” section of the User Guide.

The DMA system we want to look at is called the Peripheral DMA Controller, or PDC. When the EiE development board was initially designed, the PDC feature of Atmel Cortex implementations really stood out as an advantage over other vendors. The PDC facilitates peripheral-to-memory or memory-to-peripheral data transfers. It is highly integrated

into all the communications peripherals and even some others like the ADC. This is likely why the peripherals themselves do not have multi-byte FIFOs as you might find on other processors. There is no need for dedicated FIFOs because the PDC essentially provides an unlimited buffer to the peripheral. The PDC is covered in the “Peripheral DMA Controller (PDC)” section of the SAM3U Reference Guide.

Using the PDC is straightforward although the documentation is slightly awkward since each peripheral that uses the PDC has specific (although very similar) requirements. Each applicable peripheral has at least one channel that can be assigned to it for transfers. Below is a summary of what is available for each of the peripherals on the SAM3U. Be sure to read the “Using the Peripheral DMA Controller” section in each of the peripheral descriptions to program the PDC interaction correctly.

- **Serial Peripheral Interface (SPI):** Two PDC channels, one each for the receiver and transmitter. SPI is a synchronous full duplex protocol.
- **I<sup>2</sup>C / Two-wire Interface (TWI):** works in Master mode, only. One PDC channel for receive, one for transmit. TWI is a synchronous half-duplex protocol.
- **UART:** Two PDC channels, one each for the receiver and transmitter. UART stands for “Universal Asynchronous Receiver Transmitter” which includes RS-232.
- **USART:** Two PDC channels, one each for the receiver and transmitter. USART stands for “Universal Synchronous / Asynchronous Receiver Transmitter” which means it can be used for several different protocols.
- **ADC:** One PDC channel for conversion results.
- **PWM:** One PDC channel for updating duty-cycle registers

There are four 32-bit address pointers and two 16-bit counter registers for the interface between a peripheral and the PDC.

The four pointer registers are:

- **RPR:** receive pointer register
- **RNPT:** receive next pointer register
- **TPR:** transmit pointer register
- **TNPR:** transmit next pointer register

Each channel in each peripheral has a set of these registers if the peripheral can use the PDC.

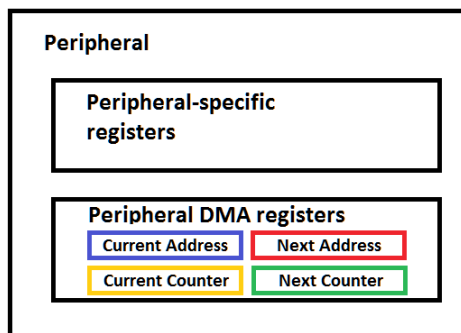
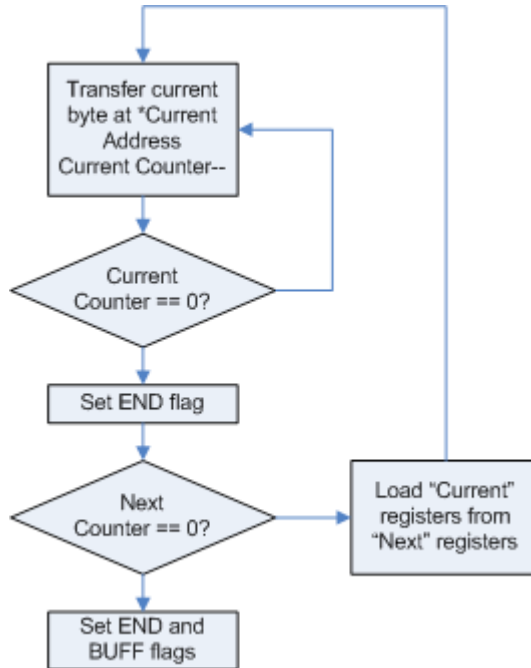


Figure 9-4 Peripheral registers

The counter registers track how much information is sent or received. This value typically refers to 8-bits of data but could be 16-bits (half-words) or 32-bits (full-words). The size of the data is configured in setup, and the address pointers are moved according to this size. Most communication peripherals work byte-wise.

There is one counter register for the transfer that is currently in progress, and another counter register for the next transfer to be done if any. Once the current transfer finishes and the counter reaches zero, the PDC checks the “next” transfer register to see if there is more data to be sent. If so, the “next” values in the address and counter registers are automatically moved to the “current” registers.



**Figure 9-5 PDC counter register flowchart**

In this way, the core can be doing something else while the PDC is transferring data and interrupts or polling can be used once the transfer is complete. The next transfer might already be queued up using the next registers, or there may be no more transfers necessary for the time being. In very busy systems, the core can always be preparing or processing data while another transfer is in progress which enables very high throughput. A good example could be streaming sensor data through USB to a PC. The processor could continually choose groups of readings to queue up and hand-off to the PDC to send. That could get underway while the core gets the next group of data ready.

For a DMA transfer to work, the amount of data that will be sent or received must be known. On the transfer side this is fairly easy, but on the receive side, it may be more difficult. Receiving devices can pre-arrange the size of data packets that they receive. Alternatively, single-byte PDC transfers can be used that run perpetually.

### 9.3.1 • PDC Registers

The user interface registers for the PDC do not have absolute addresses. Each peripheral that can use the PDC has its own set of PDC registers. The PDC peripheral looks over

all these registers to manage the data transfers that are queued up. It's not clear how exactly it does that, especially when multiple peripherals are trying to use the PDC. There are some clues in the documentation about how the PDC requests access to the bus needed for the transfer, so with some deeper reading into how bus arbitration works on Cortex processors we could figure it out. That's not necessary, though, and what we need to know for firmware development is that the PDC will correctly manage it. Look at the example communications peripheral user interface below and note the PDC register block.

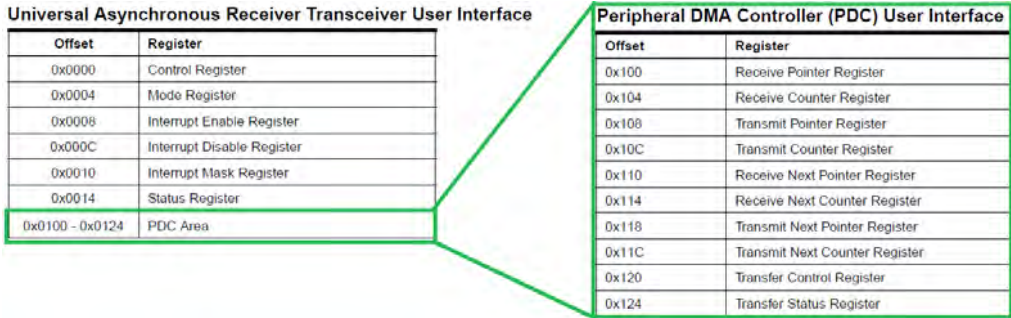


Figure 9-6 Communications peripheral's PDC user interface

This peripheral is referred to as the "Debug UART" in AT91SAM3U4.h, and you can find these registers in the register struct.

```

1356 typedef struct _AT91S_DBGU {
1357     AT91_REG   DBGU_CR;    // Control Register
1358     AT91_REG   DBGU_MR;    // Mode Register
1359     AT91_REG   DBGU_IER;   // Interrupt Enable Register
1360     AT91_REG   DBGU_IDR;   // Interrupt Disable Register
1361     AT91_REG   DBGU_IMR;   // Interrupt Mask Register
1362     AT91_REG   DBGU_CSR;   // Channel Status Register
1363     ...
1374     AT91_REG   DBGU_RPR;   // Receive Pointer Register
1375     AT91_REG   DBGU_RCR;   // Receive Counter Register
1376     AT91_REG   DBGU_TPR;   // Transmit Pointer Register
1377     AT91_REG   DBGU_TCR;   // Transmit Counter Register
1378     AT91_REG   DBGU_RNPR;  // Receive Next Pointer Register
1379     AT91_REG   DBGU_RNCR;  // Receive Next Counter Register
1380     AT91_REG   DBGU_TNPR;  // Transmit Next Pointer Register
1381     AT91_REG   DBGU_TNCR;  // Transmit Next Counter Register
1382     AT91_REG   DBGU_PTCR;  // PDC Transfer Control Register
1383     AT91_REG   DBGU_PTSR;  // PDC Transfer Status Register
    
```

Figure 9-7 Debug UART example

Since the PDC does not have its own registers, there is no specific base address. Each peripheral that can use the PDC has its own PDC register set, and these registers are always at the same address offset within the peripheral. AT91SAM3U4.h lists a peripheral-specific PDC base address. Keeping with the debug UART example, the base address of the PDC register set in the peripheral is always provided alongside the base address of the peripheral:

```

6705 #define AT91C_BASE_PDC_DBGU (AT91_CAST(AT91PS_PDC) 0x400E0700) // (PDC_DBGU) Base Address
6706 #define AT91C_BASE_DBGU (AT91_CAST(AT91PS_DBGU) 0x400E0600) // (DBGU) Base Address

```

The PDC registers are always offset by 0x100. Depending on how the code is written, PDC access may be done relative to the PDC base address of the peripherals, or directly through the peripheral's main register struct. For example, you can set the transmit address pointer register in either of the following ways:

```

/* Set the transmit pointer register using the PDC register struct relative to the
DBGU peripheral */
AT91C_BASE_PDC_DBGU->PDC_TPR = &u32TransmitBuffer[0];

/* Set the transmit pointer register using the DBGU register struct */
AT91C_BASE_DBGU->DBGU_TPR = &u32TransmitBuffer[0];

```



A good check of your understanding of how all this register addressing works is to explain why those two lines of code do the same thing.

The user interface of each peripheral does not describe the PDC registers but usually just shows them as the “PDC Area” in the user interface. In fact, sometimes there is very little discussion in a peripheral about how to use the PDC. It works essentially the same for all the peripherals once you understand it. To find out the function of each register, look at the PDC section. To be complete, the registers are summarized here. There are no setup values required. The PDC registers are used on a per-peripheral basis at the instant that data transfer needs to take place. In all cases where there is a “Next” register, those values are loaded to the current registers when a transfer is complete (i.e. when the associated counter reaches 0).

**Receive Pointer Register / Receive Next Pointer Register ( PERIPH\_RPR / PERIPH\_RNPR):** The receive buffer for incoming data on the channel.

**Receive Counter Register / Receive Next Counter Register (PERIPH\_RCR / PERIPH\_RNCR):** Number of data to be transferred. If this register is 0, the associated peripheral will stop transferring data.

**Transmit Pointer Register / Transmit Next Pointer Register (PERIPH\_TPR / PERIPH\_TNPR):** The transmit buffer for outgoing data on the channel.

**Transmit Counter Register / Transmit Next Counter Register (PERIPH\_TCR / PERIPH\_TNCR):** Number of data to be transferred. If this register is 0, the associated peripheral will stop transferring data.

**Transfer Control Register (PERIPH\_PTCR):** This control register has 4 bits that tell the associated peripheral to use the PDC for transmit and/or receive. For half-duplex communications peripherals like TWI, transmit and receive cannot be active at the same time. The next three chapters detail the different communications peripherals, so you will see how this comes in to play.

**Transfer Status Register (PERIPH\_PTSR):** Two bits that indicate if the transmit and receive PDC functions are enabled.

### 9.3.2 • PDC Interrupts

Interrupts associated with the PDC are also peripheral-specific, so each peripheral that can use the PDC will have interrupt control and status bits specific to PDC interrupts. There are four PDC interrupts:

- **ENDRX:** The receive counter register (RCR) has reached 0 since it was last written.



- **ENDTX:** The transmit counter register (TCR) has reached 0 since it was last written.
- **TXBUFE:** Transmission buffer empty: both the current (TCR) and next (TNCR) transmit counter registers are 0.
- **RBUFF:** Reception buffer full: both the current (RCR) and next (RNCR) receive counter registers are 0.

The PDC itself is not connected to the NVIC. Each PDC interrupt for each peripheral triggers the NVIC through the peripheral's NVIC connection. The four bits are shown for the debug UART

**UART Interrupt Enable Register**

31	30	29	28	27	26	25	24
—	—	—	—	—	—	—	—
23	22	21	20	19	18	17	16
—	—	—	—	—	—	—	—
15	14	13	12	11	10	9	8
—	—	—	RXBUFF	TXBUFE	—	TXEMPTY	—
7	6	5	4	3	2	1	0
PARE	FRAME	OVRE	ENDTX	ENDRX	—	TXRDY	RXRDY

**Figure 9-8 UART Interrupt Enable Register's PDC interrupt bits**

Just from the description of the PDC registers and interrupts available, you should be getting an idea of how the PDC works and how it will be used. It takes some getting used to, but effective use of DMA can have a huge performance improvement in how an embedded system runs from practically every aspect: memory usage, speed, power consumption, and overall efficiency.

### 9.3.3 • Transmitting with DMA

To transmit using the PDC, data is prepared in a sequential RAM location. This is typical and usually takes the form of a transmit buffer on the stack or heap. Even if DMA was not used, a buffer of some sort is required, anyway. Data can also come from a flash location.

In the transmit function of the task that wants to send data, the PDC address registers are set for the start of the data along with the size of the data transfer in the counter registers. Writing to the counter registers clears the associated interrupt flags. Interrupts can then be enabled for the completion of the transfer if desired.

The last step is to set the enable bit for the transmit PDC. This will trigger the peripheral to do what it needs to make a transfer which usually means the first byte to transmit is loaded into the peripheral's transmit register. This is the same transmit register that the processor core would write to queue a byte to send if the firmware was directly using the peripheral instead of the PDC. The PDC knows what register to use.

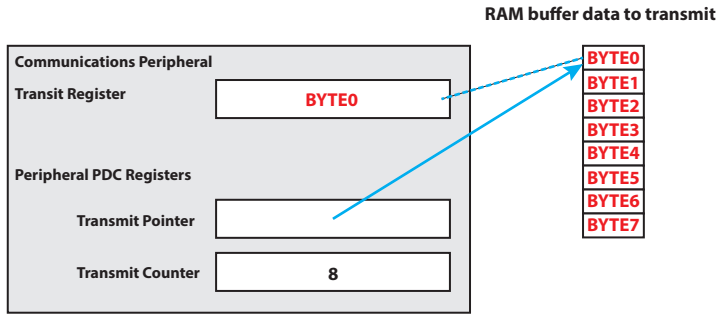


Figure 9-9 PDC Register pointer

Once the transfer has started, the PDC monitors the peripheral status registers for a bit that indicates the transmit register is ready for the next byte. When the peripheral completes the byte transfer, the PDC automatically loads the next byte and adjusts the remaining bytes counter.

When the whole transmission is complete (the counter value reaches 0), the ENDTX interrupt will fire. If there are any values in the “next” registers, the PDC will load them into the current registers and begin the new transfer. If there are no “next” values, then the PDC will also raise the buffer empty flag, TXBUFE. It is up to firmware to decide what to do at the end of a transfer. The PDC could be left enabled but the interrupts would have to be disabled because those flags are cleared only by writing a non-zero value to the counter register which would then trigger the PDC to start sending data again. So more likely the interrupt would be disabled in the ISR. If you really wanted to clear the interrupt flags, the PDC transmit could be disabled and either the next transfer counter could be loaded (if known) or a dummy value. The register values can always be overwritten. Just remember that if the counter register is not zero and the PDC transmit flag is set, data is going to be sent.

### 9.3.4 • Receiving with DMA

Receiving data via DMA is a bit trickier especially on an asynchronous communication peripheral or on a synchronous Slave peripheral since incoming data can arrive at any time and you might not know how many bytes to expect. For synchronous serial transfer protocols like I<sup>2</sup>C and SPI, it is easier if the system is running in Master mode since the number of bytes transferred from the remote Slave to the Master is known and controlled by the Master. In this case, the DMA controller can be configured for the expected bytes and the system can simply wait for an interrupt from the DMA when all the bytes have been received. This is discussed in more detail in the next chapters.

For asynchronous or Slave-mode synchronous reception, the system must be designed carefully. This is not even available for the TWI (I<sup>2</sup>C) peripheral. The first question is whether flow control lines are available. If so, reception becomes much easier from the perspective that each byte is controlled into the system and can be buffered and/or processed before the next byte starts coming in. However, the firmware will need to manage the flow control lines, although the PDC does have very basic RTS and CTS control for standard UART. This implies that single-byte reception is likely necessary to manage flow control. This is exactly how the ANT protocol works so you will see how that is implemented.

Another issue is whether the system will know how many bytes are to be received. If this information is not available, then single-byte transfers are probably necessary. It is still slightly more efficient to use DMA because you can get the byte into your application

without any processor cycles (just the DMA). However, after each byte, the system must immediately get ready for the next byte because it could be arriving very quickly. Depending on the processor speed and communication speed, you might be able to update the DMA registers before the next byte arrives. If not, then you can make use of the “Next” pointer registers to alternate between two buffer pointers. That means you’d have to patch the data back together or very carefully manage the pointers so they fill a single buffer sequentially. If receiving single bytes this way is still too slow, then you need to increase the buffers in size until you have enough processor cycles to process one buffer while the other is being used. This could work if there is a known steady stream of data. The scenario likely will not come up, however, because it is very rare to find a device external to the microcontroller that is a Master. Most sensors, LCDs, memories, etc. are Slave devices.

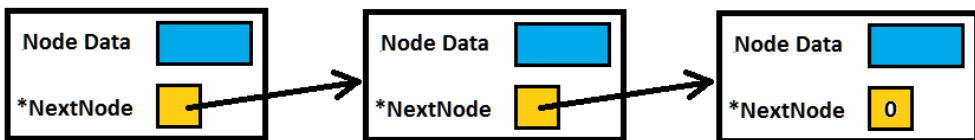
This was just an introduction to the PDC and the considerations for using it. Each chapter that uses a communication peripheral with the PDC will discuss the considerations and decisions that need to be made in designing the system. Keep this PDC overview in mind as the application of the PDC with specific communications peripherals in conjunction with the messaging task is detailed.

#### 9.4 • Linked Lists

There is one more concept that is very relevant to the messaging task that needs to be discussed. If you have ever done a programming course, it is almost guaranteed that you have learned about a data structure called a linked list. A linked list is not a specific type in C, it is a structure built using fundamental data types that are organized in a way that ends up looking like a collection of data nodes that are all linked together. The purpose of a linked list is to keep similar data together in memory in a coherent way that is flexible and still efficient regardless of how much data is currently being stored.

In the basic form, a linked list is two or more nodes. A node is typically a struct with some information and a pointer to the next node that has the same structure. This is called a singly-linked list. Doubly-linked lists have a pointer to the next node and a pointer to the previous node. This can speed up navigating and managing the list which becomes more important as the length of the list grows.

##### Singly-linked list



##### Doubly-linked list

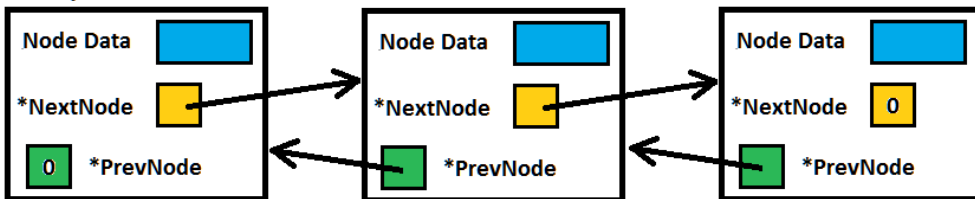


Figure 9-10 Linked Lists example

Special cases for linked lists are an empty list and a full list. The “next node” pointer in the last node in the list is usually NULL, though circular lists may point back to the start.

Some linked lists start with a unique head node that might contain different information than the rest of the nodes in the list, such as a total count of list items. The head node never disappears even when the list is empty.

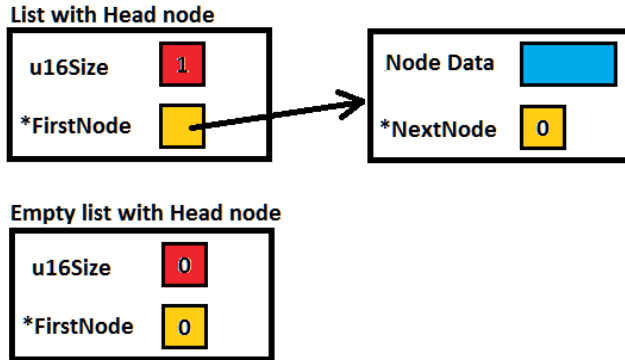


Figure 9-11 List head nodes

A common bug in linked lists is “memory leaking” due to improperly inserting or deleting nodes in the lists. This is especially prevalent when it comes to adding or removing a node in the middle of the list. It does not matter if the list is allocated statically or dynamically, managing the list entries properly is essential. If the list is in a known location, it could be totally reset in firmware if entries ended up getting lost, but band-aiding a solution like this instead of doing it correctly in the first place is poor practice.

Several pointers are required to make sure that existing nodes are not lost when the size of the list changes. Consider the following:

1. A linked list has 3 nodes where the middle node needs deletion
2. A pointer is placed on the middle node by parsing through the list to reach the correct location (which will depend on the criteria being used to select the node to delete). We like to call this pointer the “DoomedNode” pointer.

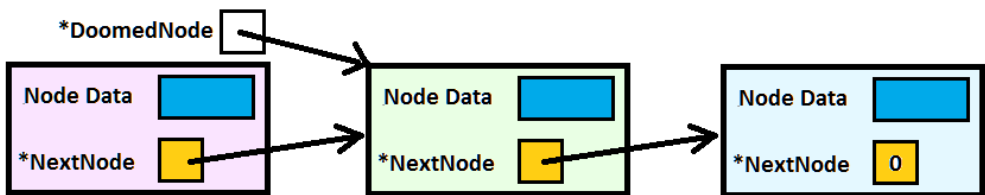


Figure 9-12 “DoomedNode” pointer

3. The `NextNode` pointer of the node before the doomed node is reassigned to `NextNode` of the middle node so it now points to the end node. If the `DoomedNode` pointer was not present, then this action would cause the middle node to be lost in memory since it would otherwise not have any pointers to it.

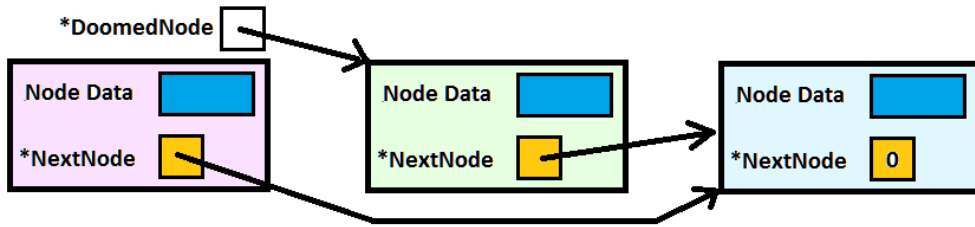


Figure 9-13 "NextNode" pointer

4. Any data can still be accessed at the DoomedNode or list cleanup can be done. If dynamic allocation is being used, the node can be safely freed.

The EiE message task will use a linked list but the list is maintained in an array of statically allocated RAM. It is still important to properly manage the list, or eventually, there will be no more spaces within the list available due to lost memory. This is essentially the same problem as losing memory with dynamic allocation, but by using a static array the scope of the problem is limited to the messaging task.

When writing code to create a linked list, it can be very helpful to draw a picture and show the pointers especially when adding or removing nodes, sorting, or any other operation involving the list pointers. Debug the code very carefully to make sure the pointers do what you think they are doing and nothing works by accident. If the code loses a node, the memory contents will still be there in the debugger window, and you might think everything is working. You might even get lucky and get a pointer to point to the same location that was lost for a few instruction cycles. At the bench, it might work every time you step through it, but when the code is released to a product another function or an interrupt could end up using that memory and you have a serious and probably difficult-to-recreate bug. Bugs like this can produce wild behavior that may not ever be repeatable because of the nature of the error.

## 9.5 • Hard Faults

As you start working with larger pieces of data, lists, arrays, and all the associated pointers, you will inevitably stumble upon one of the most difficult and frustrating problems in working with processors: Hard faults.

Hard faults are exceptions that occur because something is fundamentally wrong when the processor core is executing instructions. The most common sources of hard faults are invalid operations like divide by zero, indexing invalid memory, following null or invalid pointers, or executing invalid opcodes because you made the program counter go crazy. They are serious defects that the core simply cannot recover from. With no operating system watching out for these types of errors, the result is typically fatal.



If you want to do some serious digging into the ARM Cortex processor architecture, read about the different processor exceptions in the ARM Cortex Technical Reference manual including hard faults. There is a good chance that one day you will end up needing to read about core exceptions to solve a major issue in your product, so you might as well look at the resource now.

If a hard fault occurs, it is virtually impossible to take action that is not potentially corrupted by whatever caused the hard fault. There is a default hard fault handler that the processor will vector to and trap the code in a while(1) loop. If the system appears to "freeze" that is probably what happened. If you are running the debugger, halt the code and confirm that indeed you are stuck in the HardFaultHandler() exception handler.

Arguably the only way to truly “recover” from a hard fault is to do a software reset on the system. Obviously, this is undesirable and again simply band-aiding a serious defect in the code. Adding a bit of code in the handler is permissible, but any code that runs in the hard fault handler should be as simple as possible. No peripheral should be used, memory accesses should be limited and assumed to contain corrupted data, and function calls should be avoided since the stack could be destroyed.

If a system needs to recover, it is recommended that a dedicated RAM location be written with a special value and then a software reset performed on the system. When the system restarts, check the memory location for the special value to determine if the restart is due to a hard fault. If so, it would be safe to execute additional code to re-initialize anything that could be corrupt because of a hard fault. If the system has logging capability, you would record that a hard fault occurred. If the system can alert a user, a message should be sent.

It is also possible to dump the core registers and some memory locations to a known RAM location in the hard fault handler before the system is reset. Again, there is a risk that any of those values are corrupt, and there is still a chance that the system could be really broken so anything attempted doesn’t work properly. Use inline code and hard-coded addressing but take the data with a grain of salt due to the possibility of memory corruption.

The EiE firmware is a development system and does not make any attempt to recover from a hard fault. The default handler is updated to turn off all the discrete LEDs except for the red LED which is turned on. At least then a system that is running without a debugger is obviously in a hard fault state. If the EiE firmware was used in a commercial application, this is one of the things that would be updated suitably for the application. Releasing firmware with while(1) loops would be ridiculous.



Make sure you have the dma\_mesg branch open. Add a new HardFault\_Handler(void) function to interrupts.c. This will automatically be called instead of the WEAK default handler in exceptions.c. Use LedOn() and LedOff() to put only the red LED on in the handler (turn off all other LEDs). The LED functions have very basic code and do not require the rest of the system to operate, so they are safe to use. The handler function should still end in a while(1) loop to hold the code there.

```
void HardFault_Handler(void)
{
    LedOff(WHITE);
    LedOff(CYAN);
    LedOff(PURPLE);
    LedOff(ORANGE);
    LedOff(BLUE);
    LedOff(GREEN);
    LedOff(YELLOW);
    LedOn(RED);

    while(1); /* !!!!! update to log and/or report error and/or restart */
} /* end HardFault_Handler() */
```

Leave a breakpoint set on the while(1) so that if this occurs when running with the debugger, the code will halt right away in the handler.

Finding hard faults is particularly difficult because when they occur, the processor is really broken and usually unable to execute any code as a result. It can’t return to where the hard fault occurred because the error is terminal. Hundreds, thousands, or even millions of invalid instructions might have occurred before the hardware faults, so there is no assurance that the state of the processor in the hard fault handler is representative of its

state just before the exception was triggered. There are some registers that flag different conditions that may have led to the hard fault, but if you looked at the ARM TRM, you will quickly see how difficult the information is to use.



Hopefully, any condition that results in hard faults is repeatable. In this case, the debugger can be very helpful. If you are lucky enough to be debugging during a hard fault, there are ways to get information from the system that can be extremely useful in tracking down the source of the fault. Perhaps the most useful is the Call Stack that might have an accurate representation of the history of function calls and interrupts that occurred leading up to the fault. This is infinitely helpful when it comes to debugging hard faults since it can narrow down the source of the issue to a particular function.

We can demonstrate this with a simple exercise. An easy way to cause a hard fault is to set the UserApp1 function pointer to NULL during initialize and run the code.

```
void UserApp1Initialize(void)
{
    /* If good initialization, set state to Idle */
    if( 1 )
    {
        //UserApp1_pfStateMachine = UserApp1SM_Idle;
        UserApp1_pfStateMachine = 0;
    }
}
```

As soon as the UserApp1 function pointer is dereferenced the processor will hard fault since setting the program counter to address 0x00 is illegal. Halt the code and open the Call Stack window.

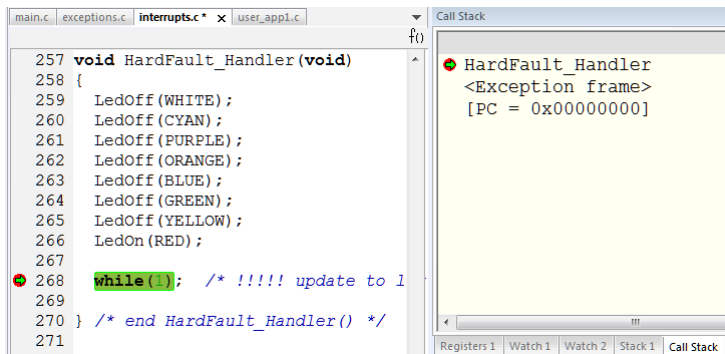


Figure 9-14 Call Stack window

Unfortunately, in this case, the Call Stack is not helpful and actually misleading, although it can be deduced that a NULL function pointer was called since the program counter is indicated as 0.



Set a breakpoint inside UserApp1RunActiveState() to halt the code just before the NULL pointer is dereferenced. The Call Stack correctly shows that the program was in main() and is now in UserApp1RunActiveState(). That information is lost when the hard fault occurs.

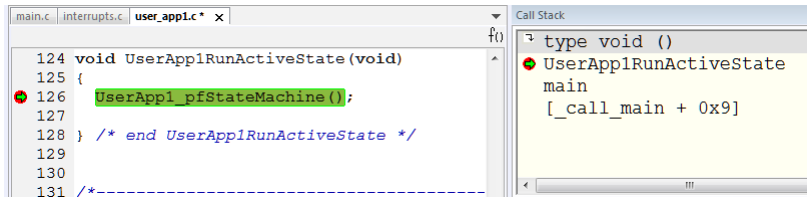


Figure 9-15 Call Stack showing the main() function

It is difficult to discuss strategies for solving hard faults beyond the general problems that lead to them because the details are very specific to the application. Building and testing code frequently helps to mitigate the challenge of finding when a hard fault takes place by minimizing the amount of new code to review. In some cases, you will have to revert back to a known working build and start re-adding the new code to determine when the fault conditions occur.

Commenting out blocks of code is sometimes useful but can make the hard fault go away because of subtle address changes in the code that is built and flashed to the system. Changing code can also free RAM addresses that might be the source of the hard fault purely by accident. Adding breakpoints or single-stepping code may also change or fix the hard fault because of how the debugger runs and interacts with code.

Hard faults that occur from interrupts happening at *\*just\** the right moment are about the worst problems to solve, especially when they occur in the hands of your customers. It is very unlikely that the customer will remember exactly what the system was doing when it crashed, or they will describe what they saw and it will not have anything to do with the real problem. More likely, no one will have noticed the crash until long after it happened, and it could be so rare that it is virtually impossible to re-create at the bench.

These are the type of errors that can be avoided by great coding standards and practices, strong foundational knowledge and experience, and careful design. Thoroughly testing firmware systems, checking edge cases, logging events and running self-diagnostics in firmware can help find problems before they get out in the field.

## 9.6 • EiE Messaging Task

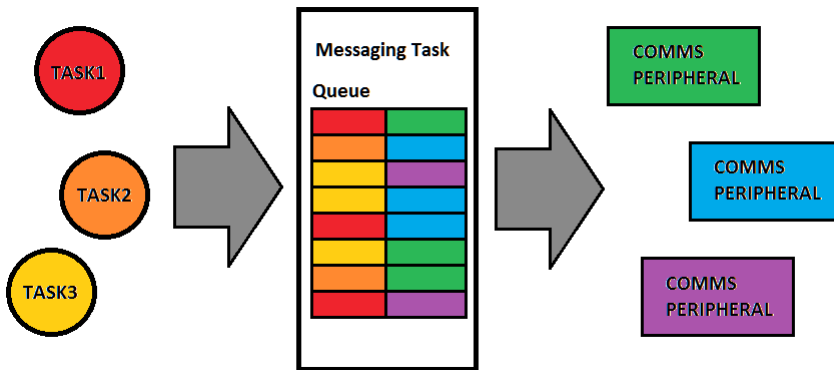
All this discussion relates to the challenges in creating a message system that integrates the PDC and DMA transfers into the multitasking and resource-sharing firmware system for EiE. A general rule of thumb is that the easier it is to use a system from the user's perspective, the more complicated the underlying design is likely to be. That's a very significant tradeoff to consider.

Setting a pointer and a counter and then saying "go" seems like a fantastic way to get a lot done in a system. Taking advantage of the DMA capabilities of the SAM3U2 was essential in designing the EiE system. Granted, it created many problems that had to be solved, and a big tradeoff was made in added complexity and code to achieve the performance and multitasking ability that was desired. It is important to understand how the process of transmitting and receiving using DMA works so that the problems that need to be solved can be anticipated in the design. Full disclosure: this is a lot easier to describe in hindsight and we didn't get it right the first time.

To start the design, we must really understand what is to be accomplished. If you have never written code like this, there is a lot to consider. Using an example can help. In the next chapter, we will write code for the debug UART that will add a "printf" style of functionality to the system. Many different tasks may want to use printf so there needs to be some sort of queuing system available. Some tasks may need to know if the message was sent or not, so a mechanism to monitor the message status needs to be included.



We want to use the messaging system for any type of message, so it must somehow accommodate any peripheral. Of course, it must support the multitasking environment while still sending messages as fast as possible. A graphical representation would look something like this where the messaging task is the interface between tasks that provide messages targeted to peripherals.



**Figure 9-16 Messaging in multi-peripheral, multi-tasking environment**

The messaging task solution that was designed for the EiE system provides a queueing system that any task can use. It is designed to handle any type of message to any type of communications peripheral. Most importantly, the messaging task will track the status of the messages to know when they have finished sending and provide a mechanism to communicate that back to the tasks.

The EiE messaging task is not used directly by high-level tasks in the system, which relaxes some requirements in complexity to avoid problems that direct high-level access to peripherals might create. Mid-level driver tasks will call protected API functions for the functions they need, and those API functions will know how to properly use the messaging task. The messaging task allows any number of tasks to queue messages and will send them in order and manage all the handshaking with the peripheral.

Each message that the messaging task is given is assigned a unique number or “token”. The only public function offered allows tasks to check the status of a message based on this token. If a task needs to know if their message has been sent, they simply ask through an API function.

That describes the essence of the messaging task design, so now we need to write the code to make it happen. Make sure `messaging.c` and `messaging.h` are added to the Drivers folders in the `dma_mesg` code branch. Add the `#include` for `messaging.h` in the “Common driver header files” sections in `configuration.h`.

### 9.6.1 • Message Task Data Structures

Creating a queueing system brings about some fundamental memory problems in a resource-limited embedded system. One way to mitigate this is by using dynamic allocation, but that has some severe potential consequences that have been alluded to and will be discussed more next chapter. Many organizations do not even allow the use of dynamic memory, nor does the MISRA C standard and we tend to follow that rule almost all the time.

The messaging queue will be built with static memory, so we must decide how big each message will be, how many message slots there will be, and how to manage messages

that are larger than the size allocated.

The size of each message can be estimated based on the use-cases of the device. The EiE firmware will use the processor to send messages to the debug UART, the LCD, ANT, and potentially other devices through the daughter board port. It will also manage command messages and data to and from the SD card. Since an SD is a mass storage medium with potential gigabytes of data moving around, we won't try to build data transfer into the messaging system.

Of the systems we will design for, most communications will be command strings that are not usually very long – under 50 bytes. ANT messages are at most 20 bytes, LCD messages could be about 50 bytes to write a whole line of text, and external devices will be similar. The debug UART that will support the EiE version of `printf` is less predictable because other tasks will oversee queuing messages there. However, if you think about the kind of messages that would be sent, it is unlikely that they would exceed 100 characters (think about how much you can say in a Twitter message when it was limited to 140 characters). Debug or status messages in an embedded system are typically short and to the point.

The size 128 is chosen to avoid most situations where messages would have to be split up. Splitting messages is not the end of the world unless a device that required a long message had to receive all that data as one contiguous transmission. We don't have any of those devices. Since the EiE firmware system has been used for several years before this book was written, hindsight tells us that 128 was a good choice! The message size is called "U16\_MAX\_TX\_MESSAGE\_LENGTH" and is set up as 16-bits to make sure the code is written to accommodate larger queues if needed.

```
#define U16_MAX_TX_MESSAGE_LENGTH (u16)128 /* Max bytes in message */
```

The bigger problem is the number of messages that should be supported as that is virtually unbounded since the number of tasks in the system is unbounded. This is a big decision because it can affect memory usage considerably. If we chose 10, then we need 1280 bytes of RAM. Remember that in total the SAM3U2 has 32kbytes of RAM. How much RAM is used so far? Check the `.map` file from the most recently built system.

Module	ro code	ro data	rw data
-----	-----	-----	-----
board_cstartup_iar.o	20	188	
buttons.o	756		52
eiefl-pcb-01.o	980	120	
exceptions.o	70		
interrupts.o	308	32	
leds.o	876		92
main.o	104		12
timer.o	218		12
user_app1.o	26		4
utilities.o	60		
-----	-----	-----	-----
Total:	3 418	340	172

Figure 9-17 Checking the `.map` file

Only 172 bytes of RAM has been used, but we have only coded very simple functions so far that have not had large data storage requirements. The upcoming chapters will require a lot more memory, but that's mitigated by centralizing most of the memory usage to the messaging task – another advantage of designing the system this way. Still, since the message task is only for outgoing messages on the most common communications peripherals, the various receive buffers and big memory consumers like USB and SD will still need space.

In the original design, the EiE system used 16 as the number of messages available simultaneously for the system. This meant that 2k of RAM was allocated for messages. It was found that this was not enough when it came to tasks that tended to output a lot of text to the debug terminal especially when people new to the EiE system were trying to output formatted text in a not-so-efficient way.



For this release of code, 32 message slots will be allocated. By no means will this guarantee that the system won't run out of message slots, but doubling from a size that was safe for most applications will take care of a substantial number of the few cases that did run out of space. An error flag can be tracked to tell firmware if there is no space left. Another error flag will indicate if the queue is approaching being full. Add these to messaging.h.

```
#define U8_TX_QUEUE_SIZE                (u8)32
/*! Number of messages allowed in the queue */
#define U8_TX_QUEUE_WATERMARK          (u8)(U8_TX_QUEUE_SIZE - 3)
/* Number of messages in the queue that will trigger a warning flag */
```

Organizing all the messages requires a data structure suitable to the task. The term "message slots" was used above which starts to paint a visual of the structure to be used. The messaging task will use a pool of messages that will be assigned to different peripherals and matched back to the originating task. Since the messages will line up to wait for their respective peripheral to become available, and since some peripherals send messages faster than others, there is no guarantee that they will be requested and returned in order. This necessitates a linked-list style of organization.



The message list will be built from structs of MessageSlotType. It has two members, a Boolean to indicate if the slot is free, and a MessageType that contains all the message information. Take a moment to visualize how the list will function as the system queues and sends messages. The messages are their own type as they will be shared around the system.

```
/*!
@enum MessageSlotType
@brief Message node in the message list
*/
typedef struct
{
    bool bFree;                /* TRUE if message slot is available */
    MessageType Message;      /* The slot's message */
} MessageSlotType;
```

The messages themselves, MessageType, holds critical details of each message. What does the system need to know for each message? The message content must be there which is a byte array of size U16\_MAX\_TX\_MESSAGE\_LENGTH that was defined already. Since MessageType is part of a linked list, a pointer to the next node is required. A member variable holding the size of the message is required since not every message will use the full space available. This only needs to be 16 bits since the maximum length is 16 bits, but we will use u32 due to byte alignment in the struct. Lastly, the token that will be associated with the message is kept in the struct so it is paired properly with its message. The token is 32-bits to ensure no duplication or at least avoid duplication that would ever risk causing an issue.



```
/*!
@enum MessageType
@brief Message struct for data messages
*/
```

```
typedef struct
{
    u32 u32Token;           /* Unique token for this message */
    u32 u32Size;            /* Size of the data payload in bytes */
    u8 pu8Message[U16_MAX_TX_MESSAGE_LENGTH]; /* Data payload array */
    void* psNextMessage;    /* Pointer to next message */
} MessageType;
```



The message queue will be global to the messaging task along with a counter to track how many messages are currently queued. The next message token will also be managed globally. Add the following declarations in `messaging.c`:

```
/******
Global variable definitions with scope limited to this local application.
Variable names shall start with "Msg_<type>" and be declared as static.
******/
static fnCode_type Messaging_pfnStateMachine; /* State function pointer */

static u32 Msg_u32Token;                      /* Incrementing message token */

static MessageSlotType Msg_asPool[U8_TX_QUEUE_SIZE]; /* Transmit queue */
static u8 Msg_u8QueuedMessageCount;             /* Number of slots occupied */
```

Applications that have queued a message need to know the status of that message. Each message has a large memory footprint and applications that are waiting for their messages to be sent do not need to access the message data. Once the message is sent, the slot it used should be returned to the pool as soon as possible to make room for other messages. However, there is no guarantee that the task to which a message belongs will check the status of the message before the slot is reused. All these factors suggest that tracking the status of the message should be done in a separate data structure.



First, define an enumerated type `MessageStateType` to hold the current state of a message. The messaging task should be able to track and communicate any message status that could occur. Before looking at the code below, consider what states might be needed? Again, visualizing and anticipating everything that could happen in the system is an essential skill to develop and apply during the design of a system. Using an enum helps since it is easy to add, remove, or modify the states that need to be tracked and reported to the client tasks while ensuring a consistent and self-documenting system in how the firmware interacts with messages.

```
/*!
@enum MessageStateType
@brief Possible statuses of a message in the queue.
*/
typedef enum {EMPTY = 0, WAITING, SENDING, COMPLETE, TIMEOUT, ABANDONED, NOT_FOUND = 0xff}
MessageStateType;
```

The message states apply to unique situations that the message could be in. Read the descriptions of the status states and visualize how the rest of the system is going to work.

- **EMPTY:** There is no message status information about the message. A valid message token in an EMPTY slot would be an unlikely scenario that should only result from a bug in the system.
- **WAITING:** The message has not started processing by the peripheral it is waiting for.
- **SENDING:** The message has been picked up by the peripheral task and

transmission is in progress.

- **COMPLETE:** The message has been sent. If a task queries a message that is in this state, the message status will automatically be marked back to the EMPTY state.
- **TIMEOUT:** The messaging task could be coded to monitor message time stamps and clean out messages that don't seem to be getting sent by their assigned peripherals. This would be necessary for a busier system or if it was found that buggy tasks were being written that started clogging up all the queues. If a task queries a message that is in this state, the message status will automatically be marked back to the EMPTY state.
- **ABANDONED:** If a message was queued to a communications peripheral but the peripheral was deallocated in the system before the message was sent. More on this when each peripheral driver is examined. If a task queries a message that is in this state, the message status will automatically be marked back to the EMPTY state.

When a message is queued, a slot will be allocated in the message array and the token will be sent with another function call to log the creation of a message in a `MessageStatus` array. This array is of type `MessageStatusType`. There is no relationship between the index of the message in the message array and the index of the message's status in the status array. The message and its status are linked exclusively by the message token. A `MessageStatusType` has the message token, the current state of the message, and a timestamp for when the message status was created.

```
/*!
@enum MessageStatusType
@brief Message tracking information
*/
typedef struct
{
    u32 u32Token;           /* Unique token for this message */
    MessageStateType eState; /* State of the message */
    u32 u32Timestamp;       /* Time the message status was posted */
} MessageStatusType;
```



The message status array will be private to the messaging task but global within it and accessed with API functions where needed. Define the array and a pointer to the current message.

```
static MessageStatusType Msg_asStatusQueue[U8_STATUS_QUEUE_SIZE];
/* Array of MessageStatusType used to monitor message status */

static MessageStatusType* Msg_psNextStatus; /* Next message status */
```

Unlike the message queue, the status array will just be a circular buffer that will overwrite the oldest information. This solves the problem of trying to ensure that message client applications check their message statuses and clear the entries in the status array. By removing this requirement, simple tasks can “fire and forget” messages, such as printf-style messages, but tasks that need to know message statuses can do so easily and not have to worry about cleaning anything up.

The status queue will be much smaller in total memory size compared to the message queue, though not negligible. Set this size as double the message queue size to give a reasonable chance that the status will be checked in time by the client that queued the message. On a production system, this should be calculated more carefully.

```
#define U8_STATUS_QUEUE_SIZE (u8)(2 * U8_TX_QUEUE_SIZE) /* # of statuses */
```

If a task is very slow and the status of a message it queued gets overwritten before it is checked, the system will report “NOT\_FOUND” as the status when the message token is searched and not located. Given the size of the status array and the processing capability of the system, this should not happen. The messaging task documentation includes this information. It is recommended that tasks look for a “NOT\_FOUND” response and adjust their querying time accordingly. The message task itself could be updated to provide some sort of system message or log if such an event occurred.

The last thing to add is a global flags variable to help with debugging or communicating issues to tasks that use the messaging task, so those clients can take more action if the messaging task is unable to carry out the functions it offers.



Make the variable declaration at the top of messaging.c

```
/* *****
Global variable definitions with scope across entire project.
All Global variable names shall start with "G_<type>Messaging"
***** */
/* New variables */
u32 G_u32MessagingFlags;          /*!< @brief Global state flags */
```



Add the following four flag bits which will be explained as they are encountered in the code description.

```
/* *****
Constants / Definitions
***** */
/* G_u32MessagingFlags */
#define _MESSAGING_TX_QUEUE_FULL      (u32)0x00000001
#define _MESSAGING_TX_QUEUE_ALMOST_FULL (u32)0x00000002
#define _DEQUEUE_GOT_NULL             (u32)0x00000004
#define _DEQUEUE_MSG_NOT_FOUND        (u32)0x00000008
```



Build the code to ensure that no errors have been made to this point. There will be a few warnings about unreferenced variables which can be ignored for now.

### 9.6.2 • Message Task Protected Functions

The messaging task will work closely with the peripheral communications tasks. When a high-level task wishes to send a message, it will make the request from the associated communication task API. The communication task will in turn access the messaging task via protected functions to properly obtain a slot from the available pool. The functions are considered protected because it is not typical that a high-level application would access them. The message token associated with the queued message is returned to the communication task which then returns it to the task that wants to send the message.

#### MessagingInitialize()



Like any task in the system, the messaging task needs to be initialized. The messaging task's global arrays, counters, and pointers should be properly set up here. Start the message count at 0, the token at 1, and the next status at the start of the status queue. Make sure all message slots are set up as free and all the associated values are zeroed out. Do the same for the values in the message status queue.

```

void MessagingInitialize(void)
{
    /* Initialize variables */
    Msg_u8QueuedMessageCount = 0;
    Msg_u32Token = 1;

    /* Ensure all message slots are deallocated and status queue is empty */
    for(u8 i = 0; i < U8_TX_QUEUE_SIZE; i++)
    {
        /* Clear the Slot value */
        Msg_asPool[i].bFree = TRUE;

        /* Clear the slot's message values */
        Msg_asPool[i].Message.u32Token = 0;
        Msg_asPool[i].Message.u32Size = 0;
        Msg_asPool[i].Message.psNextMessage = NULL;

        /* Clear the slot's message's contents */
        for(u16 j = 0; j < U16_MAX_TX_MESSAGE_LENGTH; j++)
        {
            *(Msg_asPool[i].Message.pu8Message + j) = 0;
        }
    }

    /* Clear the message status queue */
    for(u8 i = 0; i < U8_STATUS_QUEUE_SIZE; i++)
    {
        Msg_asStatusQueue[i].u32Token = 0;
        Msg_asStatusQueue[i].eState = EMPTY;
        Msg_asStatusQueue[i].u32Timestamp = 0;
    }

    Msg_psNextStatus = &Msg_asStatusQueue[0];

    G_u32MessagingFlags = 0;
    Messaging_pfnStateMachine = MessagingSM_Idle;
} /* end MessagingInitialize() */

```

Add the function call to `MessagingInitialize()` in `main.c` as the first call in the Driver initialization section. The call to run the active state is the same as any other task. The `MessagingRunActiveState()` function is already included in the source code for this chapter. Just add the call in `main.c` after the call to `TimerRunActiveState()` although the order in the super loop does not matter.

### QueueMessage()

The main protected API function is `QueueMessage()` which is what a communications task calls to obtain a message slot and connect it to the communication peripheral's transmit buffer. The connection is made with a pointer-to-pointer-to-`MessageType`. Pointers to pointers are always confusing, especially dereferencing them correctly, so be careful. Fortunately, the errors that emerge are typically syntax-based, which can be thoroughly tested in development.



Examine the function header and try to design the function.

```

/*!-----
@fn u32 QueueMessage(MessageType** ppsTargetTxBuffer_, u32 u32MessageSize_,
                    u8* pu8MessageData_)

@brief Allocates one of the positions in the message queue to the calling func-

```

tion's send queue.

Requires:

- Msg\_asPool should not be full

@param ppsTargetTxBuffer\_ peripheral transmit buffer  
 @param u32MessageSize\_ size of the message data array in bytes  
 @param pu8MessageData\_ points to the message data array

Promises:

- The message is inserted into the target list and assigned a token
- If the message is created successfully, the message token is returned; otherwise, NULL is returned

\*/

In our solution, the function starts by ensuring that the message is not empty and there is space available in the queue. Be careful when checking the required space since there are edge cases when the message size equals the queue size. Take note of the variables that are declared. The parameter `u32MessageSize_` is copied to `u32BytesRemaining` when it is declared for use in the function.

```
u32 QueueMessage(MessageType** ppsTargetTxBuffer_, u32 u32MessageSize_,
u8* pu8MessageData_)
{
    MessageSlotType *psSlotParser;
    MessageType *psNewMessage;
    MessageType *psListParser;
    u8 u8SlotsRequired;
    u32 u32BytesRemaining = u32MessageSize_;
    u32 u32CurrentMessageSize = 0;
    u32 u32MaxTxMessageLength = (u32)(U16_MAX_TX_MESSAGE_LENGTH) & 0x0000FFFF;

    /* Check for empty message */
    if(u32MessageSize_ == 0)
    {
        return(0);
    }

    /* Carefully check for available space in the message pool */
    u8SlotsRequired = (u8)(u32MessageSize_ / u32MaxTxMessageLength);
    if( (u32MessageSize_ % u32MaxTxMessageLength) != 0 )
    {
        u8SlotsRequired++;
    }
}
```


Since EiE is a development system, only a flag is set if there is not enough space. Firmware on a production device might take more action. The function which calls `QueueMessage` will get a "0" token in return if not enough slots are available, so this puts the onus on the client to act if the queue is full. It is imperative that `Msg_u8QueueMessageCount` is properly maintained.

```
if( (Msg_u8QueuedMessageCount + u8SlotsRequired) > U8_TX_QUEUE_SIZE)
{
    G_u32MessagingFlags |= _MESSAGING_TX_QUEUE_FULL;
    return(0);
}
```

If space is available, the process of copying the message into a queue begins. The most difficult challenge in allocating the message slot is dealing with messages that are too



large to fit in a single slot. The variable `u32BytesRemaining` will be used to condition a while loop that runs until all the message data has been allocated to message slots. The checks at the beginning of the function already ensure that there are enough slots for the full message. The communication peripheral tasks will need to manage what happens with messages that are split up with respect to sending the messages out of the processor since the DMA will interrupt at the end of each slot.


 Start by framing out the while loop conditioned on `u32BytesRemaining`. The basic requirements of the loop are that it increments the message counter and the message token.

The message counter is global to the messaging task, so it will be decremented by other function calls including ones that can occur during interrupts. Therefore, interrupts must be disabled around the increment since an interrupt could occur just after the variable is read to the core registers which would then undo the decrement that would occur in the ISR.

There are over 4 billion message tokens available, so we definitively state that errors due to duplicated message tokens will be effectively 0. The only thing to worry about is with the eventual rollover. The token is not allowed to be 0 since 0 is a special case for an invalid token request.

```
while(u32BytesRemaining)
{
    /* Increment the message count. Interrupts are disabled since this
    global can be accessed by an interrupt. */
    __disable_irq();
    Msg_u8QueuedMessageCount++;
    __enable_irq();

    /* Increment message token; Token 0 is not allowed on rollover. */
    Msg_u32Token++;
    if(Msg_u32Token == 0)
    {
        Msg_u32Token = 1;
    }
} /* end while */
```

 If you do not understand why interrupts must be disabled when `Msg_u8QueuedMessageCount` is incremented, take time to figure it out. Reviewing the assembly language chapter will help. Remember this concept, because it applies to so many situations and results in some of the most difficult bugs to find in released code.

A system warning flag is added that can be used for debugging or high-level system control if the message queue is getting too full. This is known as “watermarking” and the concept often appears in robust systems. The flag is not currently used, but it gives some options to help a system that could get too busy. As an exercise, think about how this flag could be used to throttle back certain tasks or change the behavior of the main system loop or even the processor clock. How would this be designed and what impacts would it have on the performance of the development board? Mitigating a busy system is more reasonable than mitigating problems that might arise due to bugs. A busy system is not a bug, though if the system is busy all the time then the design itself may need to be revisited.

```
/* Flag if we're above the high watermark */
if(Msg_u8QueuedMessageCount >= U8_TX_QUEUE_WATERMARK)
{
    G_u32MessagingFlags |= _MESSAGING_TX_QUEUE_ALMOST_FULL;
```

```

    }
    else
    {
        G_u32MessagingFlags &= ~_MESSAGING_TX_QUEUE_ALMOST_FULL;
    }

```



Allocate a message slot to the message of interest. Think about the design considerations involved here:

- There is no guarantee where the empty slot is going to be.
- There is no requirement of using sequential slots from the pool in the case of a multi-part message.
- A basic linear search is fine on such a small queue and in most cases, the empty slot will be near the start of the list, anyway.
- The checks that have already been completed ensure that there are enough slots available.
- It does not matter if new slots become available as another slot is being allocated due to being released in an interrupt.

Parse through the list to find the first slot where bFree is TRUE. Once the slot is found, mark it not-free and set a pointer to its Message member.

```

/* Find an empty slot: this is non-circular and there must be at least
one free slot if we're here */
psSlotParser = &Msg_asPool[0];
while(!psSlotParser->bFree)
{
    psSlotParser++;
}

/* Allocate the slot and set the message pointer */
psSlotParser->bFree = FALSE;
psNewMessage = &(psSlotParser->Message);

```



Decide if the full message can fit in the slot or if the message needs to be split up.

```

/* Check the message size and split the message up if necessary */
if(u32BytesRemaining > U16_MAX_TX_MESSAGE_LENGTH)
{
    u32CurrentMessageSize = U16_MAX_TX_MESSAGE_LENGTH;
    u32BytesRemaining -= U16_MAX_TX_MESSAGE_LENGTH;
}
else
{
    u32CurrentMessageSize = u32BytesRemaining;
    u32BytesRemaining = 0;
}

```



Copy the message data from the function parameter into the message slot. Copying data is an unfortunate consequence of many buffer or queuing system designs. There are ways to code the system to avoid copying data, but that can drastically complicate the code and memory usage. On the bright side, copy functions are reasonably efficient and the data size is quite small. The C-library function `memcpy()` can be used or even a DMA transfer could be done to increase efficiency. In balancing the trade-offs, the messaging task was designed to use a basic loop to copy the data.

```

/* Copy all the data to the allocated message structure */
psNewMessage->u32Token      = Msg_u32Token;
psNewMessage->u32Size       = u32CurrentMessageSize;
psNewMessage->psNextMessage = NULL;

/* Add the data into the payload */
for(u32 i = 0; i < psNewMessage->u32Size; i++)
{
    *(psNewMessage->pu8Message + i) = *pu8MessageData_++;
}

```



Once the message information has been copied into a message slot, the slot must be assigned to the client's transmit buffer. We do not need to know anything about that buffer, just that it is a buffer that will be used with the messaging task. We assume it is typed correctly for messages based on the requirements for the `QueueMessage()` function. Interrupts must be disabled again here since a message transmission in progress could end as this function is taking place which would result in movement of the peripheral function's transmit buffer pointers.

As with any list, the special case of an empty list must be managed. No new memory is being assigned at this point, so there is no "list is full" case to consider. It makes sense that the transmit buffers are FIFO, so the target transmit list keeps building up by adding message objects to the end of the list. When the current message finishes sending, the memory is given back to the message pool and the transmit buffer pointer advances to the next message object to send (or becomes NULL if the list is empty).

The most important consideration is keeping interrupts off when adding a new list entry since there is nothing preventing the system from finishing a transfer in progress as the new message object is being added. This will become clearer as the peripheral communications tasks are written and the `DeQueueMessage()` function is coded. For now, disable interrupts, add the new message, then re-enable interrupts.

```

/* Link the new message into the client's transmit buffer. This must
Happen with interrupts off since other functions can operate on the
transmit buffer. */
__disable_irq();

/* Handle an empty list */
if(*ppsTargetTxBuffer_ == NULL)
{
    *ppsTargetTxBuffer_ = psNewMessage;
}

/* Add the message to the end of the list */
else
{
    /* Find the last node */
    psListParser = *ppsTargetTxBuffer_;
    while(psListParser->psNextMessage != NULL)
    {
        psListParser = psListParser->psNextMessage;
    }

    /* Found the end: add the new node */
    psListParser->psNextMessage = psNewMessage;
}

/* Safe to re-enable interrupts */
__enable_irq();

```

```
/* Update the Public status of the message in the status queue */
AddNewMessageStatus(Msg_u32Token);
```

The last step in `QueueMessage()` is to return the message token corresponding to the new message that was added to the system. For messages that were split into multiple messages, only the highest (last) token needs to be returned as the completion of that message should reflect the status of all the messages in the group. The system will still tag and track the other messages and properly dequeue them. Their statuses will never be read so they will eventually get over-written in the status queue.

```
/* Return only the current (and highest) message token, as it will be the
   last portion to be sent if the message was split up */
return(psNewMessage->u32Token);

} /* end QueueMessage() */
```

### DeQueueMessage()

Removing a message from the message pool is the riskiest function in the messaging task because it relies on external tasks to run correctly and clear out their messages once they are complete. While this could be mitigated by adding a garbage collection feature to the messaging task, for now, it will remain the responsibility of the functions to manage correctly. These functions are part of the core EiE system, so it is pretty safe to assume we will code our own system properly.

`DeQueueMessage()` is designed to occur when the final byte has been transferred by a peripheral. Without knowing the details of those other functions, it is difficult to imagine the whole design and how it will work together. Most likely this will take place when the peripheral DMA transfer function interrupts to indicate when the transfer is complete. `DeQueueMessage()` is interrupt-safe because a slot being dequeued would never be in the process of being allocated by a regular running task.

The key assumption is that the transmit buffer of the calling task is always pointing to the message that should be dequeued. This must be true for a FIFO buffer by design. Since it is the communication task that requests the message be dequeued, the task's transmit buffer pointer is known which is why it is the parameter used to find the message. The peripheral task does not really have knowledge of the message slot and may not know the message token.



Read the function header and try to write the function. As an additional exercise, draw some memory diagrams of messages being queued and dequeued and imagine a generic communication task using the functions.

```
/*!-----
@fn void DeQueueMessage(MessageType** pTargetQueue_)

@brief Removes a message from a message queue and adds it back to the pool.

Requires:
- The message to be removed has been completely sent and is no longer in use
- New messages cannot be added into the list during this function (via interrupts)

@param pTargetQueue_ is a FIFO linked-list where the message that needs to be de-
leted is at the front of the list

Promises:
- The first message in the list is deleted; the list is hooked back up
```

```
- The message space is added back to the available message queue
*/
```

In our solution, two debugging flags are added to `G_u32MessagingFlags` to signal unexpected conditions. They are not currently checked in the firmware but are good debugging tools to monitor during development. The code that sets the flags make good breakpoint locations, also.

```
#define _DEQUEUE_GOT_NULL (u32)0x00000004
#define _DEQUEUE_MSG_NOT_FOUND (u32)0x00000008
/* end G_u32MessagingFlags */
```

The function declares a message parsing pointer and checks to ensure the target queue pointer parameter is not NULL as this would be invalid for the function. In this case, the error is flagged and the function returns.

```
void DeQueueMessage(MessageType** pTargetQueue_)
{
    MessageSlotType *psSlotParser;

    /* Make sure there is a message to kill */
    if(*pTargetQueue_ == NULL)
    {
        G_u32MessagingFlags |= _DEQUEUE_GOT_NULL;
        return;
    }
}
```

The message pool is searched until a slot whose message pointer points to the start of the target transmit queue is found. This is the message that should be removed as it just finished transmitting which would have triggered the peripheral communications function to call `DeQueueMessage()`. A check is done to ensure the message was found. If not, a flag is set.

```
/* Find the message's slot: this message pool is non-circular and the message must
be one of the slots */
psSlotParser = &Msg_asPool[0];
while( (&psSlotParser->Message != *pTargetQueue_) &&
      (psSlotParser != &Msg_asPool[U8_TX_QUEUE_SIZE]) )
{
    psSlotParser++;
}

/* Make sure the message has been found */
if(psSlotParser == &Msg_asPool[U8_TX_QUEUE_SIZE])
{
    G_u32MessagingFlags |= _DEQUEUE_MSG_NOT_FOUND;
    return;
}
```

It is debatable as to whether searching for a matching pointer instead of searching for the message token is better. The matching pointer may be less intuitive, but more to the point. The consequences of removing the wrong message by token at an unknown location in the chain of queued messages waiting for transmit instead of removing the first message in the transmit buffer are potentially worse since a message in the middle of the list has no knowledge of messages preceding it. The worst that happens if the first message is erroneously removed is that message is not sent, but the linked list would not be broken. Depending on the message content, this could be as harmless as a missed

debug output, or as critical as missing important data or configuration.

That being said, designing to “minimize consequences” is a terrible approach to writing firmware. There is no justification for simply mitigating the effects of defective code in a system that you have 100% control over. Code it properly so there are no defects!

With the message to be removed confirmed, all that remains is to update the target transmit queue to the next message and clear out the contents of the deleted message slot that is no longer needed. `Msg_u8QueuedMessageCount` can then be decremented and the function is complete.

```
/* Unhook the message from the current owner's queue and put it back in the pool */
*pTargetQueue_ = (*pTargetQueue_)->psNextMessage;
psSlotParser->bFree = TRUE;
Msg_u8QueuedMessageCount--;

} /* end DeQueueMessage() */
```

The other members of the message content could be cleared, too, so the message slot appears untouched. Keeping the content intact offers some valuable debugging insight and in fact was very helpful upon initial testing of the code as the debugger could be used to look at the message pool to see how busy it got and what tasks were queuing a lot of messages. It also saves processor cycles by not running code to clear memory. On the downside, writing new message content over old messages can make it difficult to see what message is in the slot. This should not matter to the system since the contents always have an associated length byte.

#### Private Function: AddNewMessageStatus ()

The last part of the `QueueMessage()` loop called the only private function in `messaging.c` called `AddNewMessageStatus()`. This is used to place the status of the new message in `Msg_asStatusQueue`. This is done as a function since it is a very specific piece of functionality even though it is private to `messaging.c` and not called anywhere else. Using a function helps readability in `QueueMessage()` and gives flexibility to how the message status is maintained. At the same time, it provides rigidity to the messaging system design by ensuring the token-based tracking of messages. A totally different implementation of storing message statuses could be designed that could simply drop in as long as it still returned a token.



Based on the function header, write the `AddNewMessageStatus` function. Be sure to update the three member variables at `Msg_psNextStatus`. A new message should always start in the `WAITING` state and the timestamp is the current 1ms count. Finish by safely moving `Msg_psNextStatus` to the next location in the circular `Msg_asStatusQueue[]`.

```
/*!-----
@fn static void AddNewMessageStatus(u32 u32Token_)

@brief Adds a new message into the message status queue.

Due to the tendency of applications to forget that they wrote a message here,
this buffer is circular and will overwrite the oldest message if it needs space for
a new message.

Requires:
- Msg_psNextStatus points to the next status location

@param u32Token_ is the token of the message of interest

Promises:
```

```

- A new status is created at Msg_psNextStatus indexed by u32Token_
*/
static void AddNewMessageStatus(u32 u32Token_)
{
    /* Install the new message */
    Msg_psNextStatus->u32Token = u32Token_;
    Msg_psNextStatus->eState = WAITING;
    Msg_psNextStatus->u32Timestamp = G_u32SystemTime1ms;

    /* Safely advance the pointer */
    Msg_psNextStatus++;
    if(Msg_psNextStatus == &Msg_asStatusQueue[U8_STATUS_QUEUE_SIZE])
    {
        Msg_psNextStatus = &Msg_asStatusQueue[0];
    }
}

} /* end AddNewMessageStatus() */

```

### UpdateMessageStatus()

The message status is independently updated by the communications peripheral task using the protected function, UpdateMessageStatus(). Again, this API function is designated protected since it should not typically be used by high-level tasks in the system.

```

/*!-----
@fn void UpdateMessageStatus(u32 u32Token_, MessageStateType eNewState_)

@brief Changes the status of a message in the statue queue.

Requires:
@param u32Token_ is a message that should be in the status queue
@param eNewState_ is the desired status setting for the message

Promises:
- if the token is found, the eState of the message is set to eNewState_
*/

```



Implementing the function requires parsing the Msg\_asStatusQueue array to find the token. Since Msg\_asStatusQueue is global, this can be done by indexing the array directly or by using a parsing pointer. Our solution uses a pointer. Make the update if the token is found, otherwise, do nothing. It might be useful to return a Boolean or the message timestamp if the message was found and updated so the client knows it was found. There has yet to be a need for this, so it was excluded.

```

void UpdateMessageStatus(u32 u32Token_, MessageStateType eNewState_)
{
    MessageStatusType* pListParser = &Msg_asStatusQueue[0];

    /* Search for the token */
    while( (pListParser->u32Token != u32Token_) &&
           (pListParser != &Msg_asStatusQueue[U8_STATUS_QUEUE_SIZE]) )
    {
        pListParser++;
    }

    /* If the token was found, change the status */
    if(pListParser != &Msg_asStatusQueue[U8_STATUS_QUEUE_SIZE])
    {

```

```

    pListParser->eState = eNewState_;
}

} /* end UpdateMessageStatus() */

```

## 9.7 • Messaging Public Functions

Only one public function is made available to high-level applications that allows any task to check the status of a message in the queue. Obfuscating the details of how the system messaging works helps keep complicated data processing out of sight of the applications. This also adds to the portability of the firmware as any of the messaging system design could be replaced by something different as long as the basic concept of sending a message and getting a token back on which status could be checked is still implemented. Simple systems without queues that use peripherals directly could be used, or the application could be incorporated in an operating system with very little change.

### 9.7.1 • QueryMessageStatus()

Any application that needs to know if a message is properly sent should be coded to receive the message token that will be provided through the messaging application via whatever intermediate function signs out a message slot. The high-level task must understand the MessageStateType status and respond appropriately. The function simply parses Msg\_asStatusQueue looking for the token provided and returns the message status that it finds. NOT\_FOUND is returned if the message is not there.

When the messaging system was first discussed, it was mentioned that querying the message status was the mechanism by which the status queue would know to clear out a message status. COMPLETE, TIMEOUT, and ABANDONED are considered final states, so if a task queries a message token in this state, that triggers the automatic clean-up of the status slot. The worst-case lag time is one cycle of the 1ms main loop where a message that just finished has not triggered an update in the status queue.



Read the function documentation below and write the code for the function.

```

/*!-----
@fn MessageStateType QueryMessageStatus(u32 u32Token_)

@brief Checks the state of a message and returns the current MessageStateType

If the state is COMPLETE, TIMEOUT or ABANDONED, calling this function
forces the associated status to be cleared from the message queue.
Since the queue is quite short, most of the time it will hold very little entries.
New entries are always filled at the front, using a simple linear search starting
at index 0.

Requires:
@param u32Token_ is the token (ID) of the message of interest

Promises:
- Returns MessageStateType indicating the status of the message
- if the message is found in COMPLETE, TIMEOUT, or ABANDONED state, the status is
  removed from the queue and time-stamped for debugging purposes.
*/

```

In our solution, the code initializes the return value eStatus to an assumed state of NOT\_FOUND and then proceeds to sequentially parse through Msg\_asStatusQueue[]. If the token is found, the current state is updated in eStatus. If the state is a final state, the slot is cleared.



```

MessageStateType QueryMessageStatus(u32 u32Token_)
{
    MessageStateType eStatus = NOT_FOUND;
    MessageStatusType* pListParser = &Msg_asStatusQueue[0];

    /* Brute force search for the token - the queue will never be large enough
    on this system to require a more intelligent search algorithm */
    while( (pListParser->u32Token != u32Token_) &&
           (pListParser != &Msg_asStatusQueue[U8_STATUS_QUEUE_SIZE]) )
    {
        pListParser++;
    }

    /* If the token was found pListParser is pointing at it */
    if(pListParser != &Msg_asStatusQueue[U8_STATUS_QUEUE_SIZE])
    {
        /* Save the status */
        eStatus = pListParser->eState;

        /* Release the slot if the message state is final */
        if( (eStatus == COMPLETE) || (eStatus == TIMEOUT) ||
            (eStatus == ABANDONED))
        {
            pListParser->u32Token = 0;
            pListParser->eState = EMPTY;
            pListParser->u32Timestamp = G_u32SystemTime1ms;
        }
    }

    /* Return whatever status is in eStatus */
    return(eStatus);
} /* end QueryMessageStatus() */

```

That completes the description of the functions needed for the messaging task.

## 9.8 • Messaging State Machine

The messaging task is coded as a state machine, but it only has an Idle state and the Idle state does not do anything. Some hooks are in place to suggest some garbage collection actions that might be taken.

```

/*!-----
@fn static void MessagingSM_Idle(void)

@brief Right now this doesn't do anything
*/
static void MessagingSM_Idle(void)
{
    static u32 u32CleaningTime = U32_MSG_STATUS_CLEANING_TIME;

    /* Periodically check for stale messages */
    if(--u32CleaningTime == 0)
    {
        u32CleaningTime = U32_MSG_STATUS_CLEANING_TIME;

        /* ??? Probably should add clean of the main message queue to detect any
        messages that have become stuck */
    }
}

```

```
} /* end MessagingSM_Idle() */
```

There are some constants in `messaging.h` that are set up as suggested time-out periods for different actions to take place, though these have not been vetted and are merely seed values for a possible starting point.

```
/* Future: possible time-to-live constants for messages in the queue */
#define U32_MSG_STATUS_COMPLETE_TIME    (u32)1000
/* Max time in ms that a message can be in the COMPLETE state */

#define U32_MSG_STATUS_WAITING_TIME      (u32)3000
/* Max time in ms that a message can be in the WAITING state */

#define U32_MSG_STATUS_TIMEOUT_TIME      (u32)5000
/* Max time in ms that a message can be in a TIMEOUT state */

#define U32_MSG_STATUS_CLEANING_TIME      (u32)10000
/* Time in ms between cleaning the message queue */
```



Add the code and do a final build to make sure that everything is syntactically correct. Real testing will start in the next chapter as a communications task is coded to start using the messaging task.

Take a moment to view the `output.map` file and see the memory that is now allocated to the messaging task. Try changing the size of the messages and the number of slots available to see the impact of the total RAM usage.

200	Module	ro code	ro data	rw data
201	-----	-----	-----	-----
202	board_cstartup_iar.o	20	188	
203	buttons.o	756		52
204	eiefl-pcb-01.o	980	120	
205	exceptions.o	70		
206	interrupts.o	308	32	
207	leds.o	876		92
208	main.o	112		12
209	messaging.o	272	4	5 397
210	timer.o	218		12
211	user_appl.o	26		4
212	utilities.o	60		
213	-----	-----	-----	-----
214	Total:	3 698	344	5 569

**Figure 9-18 RAM usage**

At this point, there is still plenty of RAM available in the system, but the messaging task has taken a significant chunk of the available resource.

## 9.9 • Chapter Exercise

The messaging task is the start of the most complicated part of the EiE embedded system but is critical in solving the problem of running a multitasking system where resources are shared, and tasks must coexist without knowledge of what other tasks are doing.

There is no direct application of the messaging task. Take the time to ensure the data structures and general operation of the messaging task are well understood. Review the code where interrupts were disabled and understand this. A great self-directed activity would be to code a function that uses the message queues with a simulated communication peripheral. Messages could be coded as LED colors that could save and then display sequences of lights as they are “sent” by your application.



## Chapter 10 • Serial and Bugs for Breakfast

Serial communication seems to be the standard choice of sending data in digital systems these days, because of the small hardware overhead and the wickedly fast speeds that are possible despite having only a single data line. You are probably familiar with USB and Ethernet – both are serial communication protocols. Firewire and SATA are other examples. All of these are capable of extremely fast data rates but there is significant software overhead in implementing them successfully.

The complexity of the protocol of these examples makes them too complicated for much of the low-level communications needed for basic data transfer between circuits in an embedded system. This is not to say that an embedded system cannot or will not use a fast serial protocol, but it is typically only done for applications that talk to the outside world like a web server or USB data connection. On the embedded system itself, the microcontroller and supporting integrated circuits optimize data transfer for power, low latency, low overhead, and hardware simplicity. Understanding simpler protocols helps to digest the more complicated ones.

In general, serial protocols have common layers, though each protocol is implemented differently. These layers correspond to the bottom two layers of the OSI networking model, namely the Physical layer (PHY) and Data Link layer (MAC). The output is single bytes of data. The signaling details are often managed by a dedicated microcontroller peripheral so the firmware designer just needs to understand how to properly configure the peripheral and then feed it and take data from it. Additional layers to parse and process the data are added as required, but generally, that is where firmware takes over to handle the transferred information in a manner appropriate to a specific system or application.

This chapter introduces a serial communication standard that is probably the most commonly used method to exchange data in embedded systems: RS-232. The signaling protocol is typically handled by a microcontroller peripheral called a “UART” (Universal Asynchronous Receiver Transmitter) and embedded engineers tend to use the terms RS-232 and UART interchangeably to refer to this type of communications even though that’s not quite correct.

RS-232 is plenty fast enough for onboard communications or debugging services and almost every microcontroller will have a hardware peripheral for it. Even if it does not, you can always bit-bash a driver to implement it. This chapter examines RS-232 signaling in depth since you are pretty much guaranteed to come across it as an embedded designer. The chapter connects the messaging task from the last chapter to the UART peripheral on the SAM3U2 to finally give us a “printf” style of output and a lot of other useful debugging features. In the next two chapters, we will do the same for two other very common serial interfaces, the Serial Peripheral Interface (SPI) and the Inter-Integrated Circuit (IIC or I<sup>2</sup>C) interface.

### 10.1 • RS-232 Overview

RS-232 is, or at least was, the defacto standard for serial communication in computer and embedded systems. Throughout the 1980s and 1990s, pretty much every system that needed to get data in and out could do so through a serial port and a standard “DB9” connector. Even modems were built to logically look like serial ports to a PC. The only real competition for RS-232 was the LPT parallel port, but the cables were much bulkier and more expensive even though the data throughput was essentially 8 times faster since it was byte-wise rather than bit-wise.

Now, of course, those technologies are pretty much obsolete on mainstream PCs, but embedded designers still make use of them extensively, especially RS-232. A serial

connection for debugging or backup data access is essential in an embedded system and it would be extremely rare not to find a serial connection on a device.

On the physical side, RS-232 uses a 9-pin “D-sub” connector, or DB9 as shown. These connectors come in both male and female versions. The pin-out is standard.

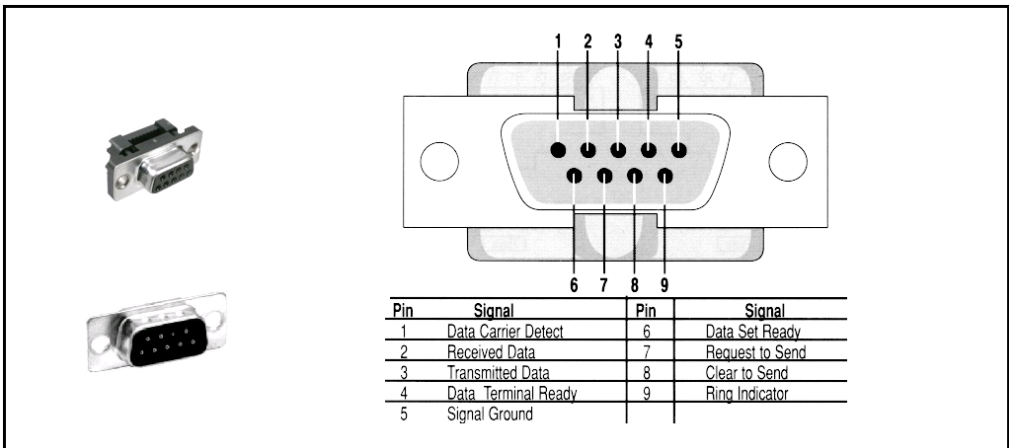


Figure 10-1 RS232 9-pin D-sub connector

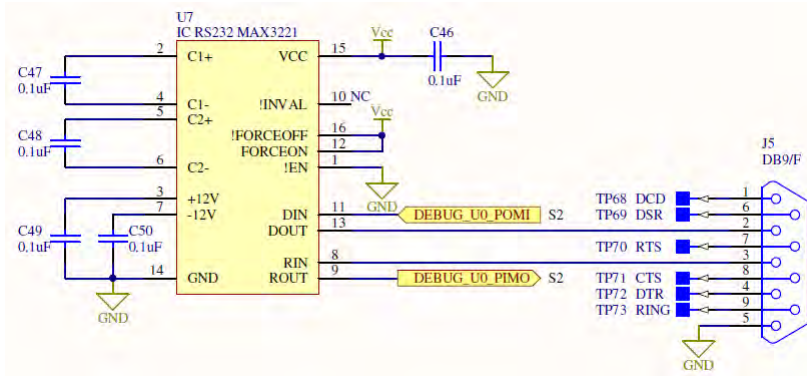
Complete RS-232 involves transmit (Tx) and receive (Rx) data lines along with several lines used for optional hardware flow control and other signaling that go back to the modem world where they were necessary. The jargon is “Terminal” (usually a PC) and “Device” (usually a modem in the original context), which are sort of like Master and Slave, though the protocol is full duplex and both sides can initiate data transfer. One device usually plays a Master-like role.

With hardware flow control, both the Terminal and the Device can indicate their readiness for data transfer using RTS (Ready to Send) and CTS (Clear to Send) lines. This is especially helpful when either the Terminal or Device does not have a large data buffer for receiving data and/or takes a long time to process the incoming data. The Device can also signal that it needs to send an incoming byte with the Ring line. A lot of processors are so fast now, that serial communications with RS-232 can assume that both Terminal and Device are always ready to send and receive data, so the transmit and receive lines (and common/ground) are the only connections 100% necessary. This is how a lot of modern devices would connect including the EIE development board, so the rest of the discussion will assume this simple case.

If you have full control over both sides of the system, the other pins can be used for anything such as additional signaling, connection detect, or power. There are no defined power connections in the RS-232 standard. If connecting to a computer, it is sometimes possible to get some power from the signaling lines which are supposed to be +/- 12V, but the standard voltage levels were never really enforced so it can vary substantially. The amount of current available on the lines will also vary across different systems, so any circuit that tries to draw power from the 12V lines may load it down too much. You might be able to design something that works for a particular computer, but the chance of it working on other computers is slim.

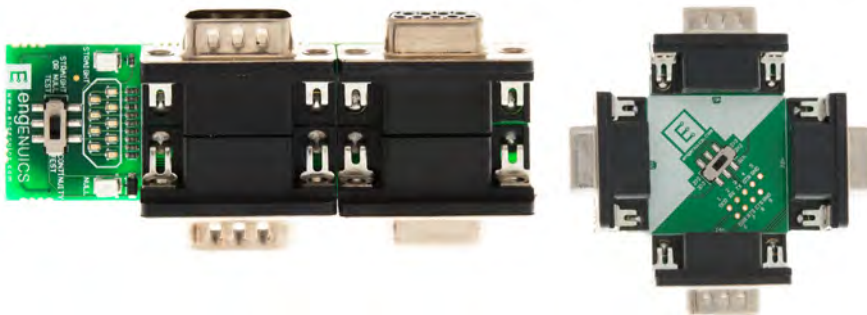
If you try to use power from the serial port, you must make sure your embedded system that is running on 0/5V or 0/3.3V is properly protected from over or under voltage. This also applies to the signaling lines. There are lots of charge pump/inverter circuits built for RS-232 level translation like the MAX3221 that is/was used on different versions of the

EiE development board. The "POMI" line is "Processor Out Machine In" and the PIMO line is "Processor In Machine Out."



**Figure 10-2 RS-232 level-translator schematic**

POMI and PIMO are common terms to see because getting the signal direction correct is essential for the system to work especially through a driver chip. Sometimes systems connect directly through a serial cable. Typically, two devices connect “straight through” so the transmitted data sent on pin 3 from the terminal is received on pin 3 of the Device. There is an alternate configuration called “null modem” that swaps the transmit and receive pins. In a way, this makes sense because intuitively the target device should receive on the receive pin. This can be a total pain because the cables do not ever seem to be marked to tell you if it is a straight-through cable or null modem with Tx and Rx swapped. A good quality cable will have “null modem” or “X” (for cross-over) or some other indicator to tell you what it is. If it does not, a piece of masking tape works nicely. To save yourself a lot of time, always label the serial cables you have lying around! However, you still must know how your system is configured and whether it will be expecting to transmit or receive on pin 2 or pin 3. We have built hardware devices for the specific purpose of easily identifying and even reversing this common mistake!



**Figure 10-3 Engenuics Serial cable tester (left) and Universal Gender Changer (right)**

Setting up RS-232 is something that you are probably familiar with even though you might not have realized it. At some point in your computing experience, you may have configured a dialog box that was asking for settings like baud rate, flow control, stop bits and data bits. These same settings will need to be configured on a microcontroller using firmware.

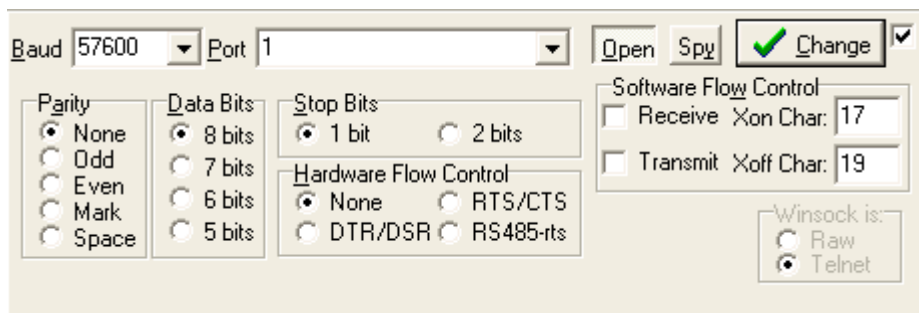


Figure 10-4 Typical RS-232 configuration options

### 10.1.1 • Clocking

RS-232 is an asynchronous protocol which means that the clock signal needed to time and parse the message bits is not sent as part of the communication. The two ends of the system use a pre-determined clock speed, and the receiver starts counting when it sees the first transition on the transmit line. Some RS-232 communication systems offer an “auto bauding” feature that is supposed to allow the system to determine the clock rate automatically. However, this requires that both systems know they are trying to autobaud and must send a preliminary data stream of 010101010101... for a long enough period that the clock can be determined. It does not always work.

Timing errors can arise quite easily with RS-232 because it is asynchronous. It requires a clock signal to generate and read the waveform like any other digital system, but the clock signal is not transmitted, so each side of the system must generate its own clock.

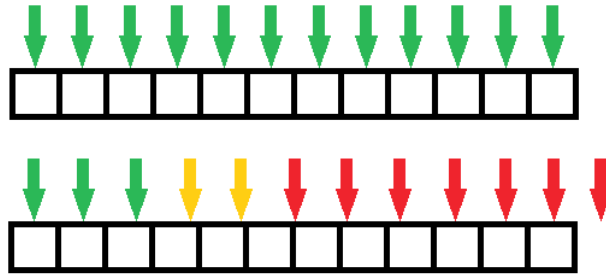
The serial clock is usually derived from the system’s main clock source through some sequence of dividers, prescalers or shift registers. As a result, specific standard baud rates may not be possible. Computers with GHz core clock speeds can usually generate very good clocks for serial data for even the fastest standard RS-232 rates. However, some embedded systems will not have a clock that is fast enough or of the right frequency that can be divided down to support the exact baud rate required.

A lot of very low power embedded systems use a 32kHz watch crystal, which is actually 32.768kHz. 32768 is a power of two and most power-of-two clock speeds can be divided down in hardware to support a standard baud rate with 0% clock error.

In addition to the problem of dividing a clock signal to get the RS-232 clock, the hardware itself will vary from system to system. As you can imagine, two independent systems generating “the same” clock are never going to be identical even if their hardware is the same. Differences in the generated clocks can eventually lead to data errors, so data transfers must be properly managed.

In an ideal world, both sender and receiver clocks are exactly synchronized and do not drift. In such a scenario an endlessly long data stream could be sent without any errors due to timing. However, if one clock is slightly faster than the other, then the sampling point of each bit will drift as more data is sent and eventually fall into the next bit or previous bit resulting in an error.

The drawing below shows an exaggerated case. The sampling clock is too slow so only the first 3 bits are sampled correctly, the fourth and fifth bits are pretty close, and the remaining bits are totally off.



**Figure 10-5 Perfect sampling (top) vs. sampling clock too slow (bottom)**

Another issue to consider is the speed at which incoming bytes can be processed by the system. Those low-power embedded systems with 32.768kHz crystals could theoretically clock data at almost 33kbps. Standard RS-232 bit rates are 14,400, 9600, 4800, 2400 and even slower. Since it takes at least a few instruction cycles to read and deal with a data byte that has arrived, the system likely cannot operate at 14,400 bps or even 9600. At 2400bps, the system would have just over 13 instruction cycles per bit which should be plenty even for a bit-bashed receiver.

Another system might have a much faster clock, like 4MHz. This system will have no problem running enough instruction cycles between data bits to manage the data at 2400bps or even 115,200bps which is a popular “high speed” RS-232 rate. However, 4,000,000 is not a power of 2, so most standard microcontrollers cannot scale it down exactly to provide a standard serial bit rate. There will be a limitation of how many consecutive bytes can be sent before the clock drift will be too much. This limitation can be mitigated, or the system clock could be adjusted to, say 4.192MHz which is divisible by a power of two and thus can be divided down to produce an exact baud rate. Even then, differences in the system will likely result in some drift.

If the 32.768kHz and 4MHz systems were talking at 2400bps, the 4MHz system might have a harder time staying in sync than the slower system since it has some error in the generated baud. If the first transmitted bit triggers the receiver’s clock and all subsequent bits of however many bytes are clocked relative to the first bit, then clock drift between the two systems will eventually lead to errors.

### 10.1.2 • Signaling

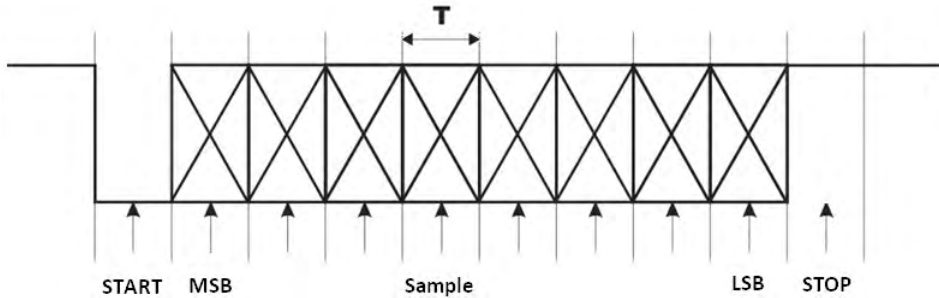
The framing of a byte sent via RS-232 helps to solve errors due to clock drift and provides the standard for how a transmitter and receiver know when each byte starts and stops. Most systems resynchronize at every start bit, so there can be some pretty high clock errors (or drift over time, temperature, voltage, etc.) and still have very reliable communication.

Once communication parameters are set, data is transferred one bit at a time until a byte has been sent. The number of data bits can be adjusted, but it’s safe to assume it will be 8. The stop and start bits add an additional 2 bits of overhead, so each byte costs 10 bits to send. This means that RS-232 is only 80% efficient at a maximum. This is further reduced if extra stop bits or parity bits are added.

The data lines are at a logic high state when no traffic is present. A start bit is a high-to-low transition and the line remains low for the duration of the start bit. The receiver looks for the falling edge of the start bit to know when a frame is starting. As soon as the start bit edge is detected, timing starts that controls when each bit is sampled to determine the bit value. The receiver uses its baud clock to sample the start bit, data bits, and the



stop bit. In signaling diagrams, the data bits are shown as “X” symbols since we do not know and don’t care if they are high or low. Data can be MSB or LSB first – there does not tend to be any standard about that, so just ensure that both systems are the same.

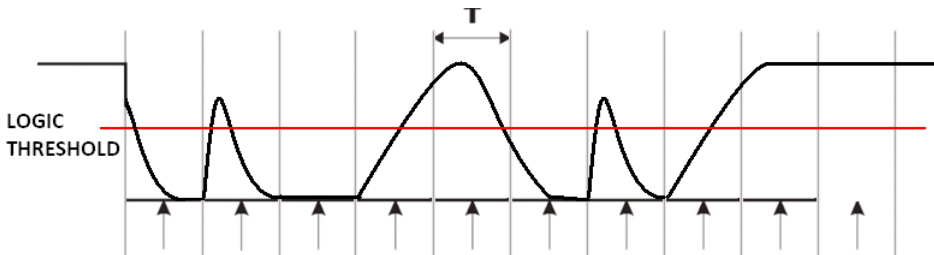


**Figure 10-6** A serial frame with one start bit, one stop bit, no parity and 8 data bits

If the stop bit is not logic high when it is sampled, then a “framing” error has occurred. On a microcontroller, the error would be reported as a flag in one of the serial peripheral registers. Of course, there is nothing physically different between a start bit, a stop bit and a data bit. The start and stop bit are the only two bits in the frame that have a known (or at least expected) state. The start bit of the next frame can occur immediately after the stop bit of the previous frame. Some systems may require a short delay between bytes. This would be specified somewhere in the datasheet for the device.

The bit period is “T” and the frequency of the signal (the baud rate) is  $1/T$ . In many cases of hardware serial ports, a single bit sample is taken at the half-way point of each bit based purely on timing as calculated from the start bit falling edge. A more robust system might take 3 or more samples per bit and then decide on the bit state based on majority rules. Extra sampling is another way to improve data integrity, though it still does not guarantee perfection.

Problems can also occur if there is too much resistance, capacitance, inductance or noise on the transmission line that causes the signal to distort. The longer the serial cable connecting the device, the more capacitance, inductance, and resistance is present. If there is a noise source present, then the signal could be further distorted, and ones could start looking like zeros or vice-versa. Transmit speed will also cause distortion due to parasitic capacitance and inductance in the data path and further increase the chance of errors when clocks are not synchronized exactly. If you are having communications problems, sometimes slowing down the baud rate will make the problems go away. If so, then you have a great hint to prompt yourself to verify the signals.



**Figure 10-7** Poor transmit line (capacitive / resistive)



*A nice bonus about bit-bashing a communication protocol is that you can toggle a separate IO line every time you sample and see exactly where your sample is reading the incoming data. Just be careful that the extra instruction cycles to toggle the IO do not disrupt your timing.*

## 10.2 • Data Errors

Considering everything that might happen in a system, it is the designer's job to create the hardware as best as possible (limit line lengths, shield wires, etc.) and then implement some sort of scheme to ensure an acceptable level of data integrity. Like practically every other problem and solution set in engineering, there are tradeoffs in the choices made. There is a balance between data throughput, probability of error and error checking, and the right balance will depend on the system requirements.

If the data is repetitive, non-critical, and heavily averaged, a few bit errors now and then may not matter. If you need bullet-proof communications, you may add complex error checking, error correction, data acknowledgments, checksums, redundancy, etc. Any technique will cost you in data overhead and firmware size and complexity. Making the right decision can be very tricky, but the best you can do is evaluate your requirements thoroughly. The most standard RS-232 communication settings do not include any error checking.

The protocol defines a parity bit that can be added to help detect errors. The parity bit is usually the binary sum or XOR of all the data bits. The problem with a parity bit is that it can only detect an odd number of errors since an even number of errors will always cancel each other out. The parity bit itself could be wrong as well.

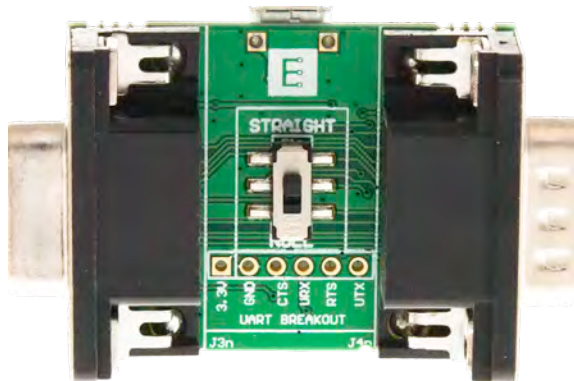
A slightly better approach is to add a checksum byte and perhaps some known signaling bytes within the data stream. This is done by the application and has nothing to do with the RS-232 protocol. This also increases the complexity and overhead in the system. If you are willing to "risk" sending a complete message with a single checksum, then size overhead is minimized unless the checksum is determined to be wrong. The receiver may be able to ignore messages that fail the checksum, for example, if it was regularly reading temperature values from a sensor where missing an occasional reading doesn't matter. If the data was critical, then the receiver must somehow inform the transmitter of the problem and the entire message must be sent again.

A happy medium is to define a message protocol that uses some known pre-amble, a strict message structure, and some error checking such as a checksum after a complete message. If a message is typically 10-20 bytes or more, then a few bytes of overhead is reasonable. If the data is non-critical such as debug messages in text where errors would be obvious like a missing character, or where system integrity is known to be quite reliable and an occasional error does not matter, it may be fine to simply spit out data as fast as possible and not care about errors.

A good example is the ANT radio protocol. Although communication to the ANT processor on the EiE development board is done over SPI, there are implementations where a UART interface is used. In both cases, the message structure is the same. ANT messages always start with a known signaling character as the first byte in its message, and the last byte of a message is a checksum which is an XOR of the signaling byte and all the data bytes. This provides reasonably strong error checking for the system, though it is not completely fail-safe. That being said, we have NEVER noticed a bit error in years of using ANT.

Overall, RS-232 is great because it is so widely used and available on any PC even though you now need a USB-to-RS-232 adapter since it is very rare to find a computer with a serial port. This is another tool that we have built and sell because it is still so handy for

embedded development and the commercial ones are getting more expensive and difficult to find each year and a few extra features are nice to have.



**Figure 10-8 Engenuics full-featured USB to RS-232 converter**

The major downfall of RS-232 is the need for precise clocks, and the problem is compounded when devices operate in the real world where temperature, battery life, and other factors will come in to play. Reliable RS-232 communication necessitates that a crystal oscillator be your main clock source, or at least your system must have a way to calibrate itself prior to communications. Calibration is inconvenient and likely involves a crystal or external signal of predictable and stable frequency, anyway.

### 10.3 • ASCII

You need to be up to speed on ASCII characters. ASCII is an acronym for “American Standard Code for Information Interchange” and is probably one of the most ingrained standards in the computing world. Discussion of ASCII goes hand-in-hand with discussion of RS-232 because very often character strings are being sent back and forth between an embedded application and a PC terminal for display. A major part of this is converting binary data to printable characters, so you will spend lots of time turning numbers into character strings and vice-versa.

If you are not familiar with the ASCII code set for the first 256 character codes, check out [www.asciitable.com](http://www.asciitable.com). The tables define the binary number that is sent for each character and symbol that can be typed on a keyboard. For example, the capital letter ‘A’ is 65 (0x41 in hex, or b’01000001’ in binary). You can send many of the standard ASCII codes from your keyboard just by pressing the key with the right label. The little embedded system in the keyboard automatically sends the correct ASCII code for the button you pressed. You can also enter ASCII codes directly by holding the “Alt” key while typing the character code you want. For example, if you press ALT-6-5 you will send the ASCII code for the letter A. Try it! Codes 248 (degrees symbol °) and 253 (squared symbol ²) are useful, as are the codes for accented characters if you need them but do not have a keyboard that supports them directly.

For the record, there are different code pages for the upper half of the 8-bit ASCII codes as they were never really standardized. ASCII codes have long-since been superseded by 16-bit (and greater) standardized Unicode to support every symbol, character in every language, and even emojis. We will avoid going into detail beyond stating that the first 128 characters in Unicode are the same as ASCII. We will assume that we can work safely with the first 128 character codes and most of the next 128 character codes based on North American computers with apologies to our international readers.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EXT (end of transmission)	36	24	044	&#36;	&	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	}	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: [www.LookupTables.com](http://www.LookupTables.com)

Figure 10-9 First 128 characters of ASCII or the same as Unicode

128	Ç	144	É	160	á	176	ð	192	Ł	208	ł	224	α	240	≡
129	ü	145	æ	161	í	177	é	193	ł	209	ŧ	225	β	241	±
130	é	146	Æ	162	ó	178	ë	194	Ł	210	Ŧ	226	Γ	242	≥
131	â	147	ø	163	ú	179	ı	195	ł	211	Ł	227	π	243	≤
132	ä	148	ö	164	ñ	180	ı	196	—	212	Ł	228	Σ	244	ƒ
133	à	149	ò	165	Ñ	181	ı	197	+	213	Ł	229	σ	245	Ƶ
134	â	150	û	166	•	182	ı	198	Ł	214	Ł	230	μ	246	÷
135	ç	151	ù	167	°	183	ı	199	Ł	215	Ł	231	τ	247	≈
136	ê	152	ÿ	168	ı	184	ı	200	Ł	216	Ł	232	Φ	248	°
137	ë	153	Ö	169	ı	185	ı	201	Ł	217	Ł	233	Θ	249	•
138	è	154	Ü	170	ı	186	ı	202	Ł	218	Ł	234	Ω	250	•
139	ı	155	•	171	½	187	ı	203	Ł	219	Ł	235	δ	251	√
140	ı	156	£	172	¾	188	ı	204	Ł	220	Ł	236	∞	252	∞
141	ı	157	¥	173	ı	189	ı	205	=	221	ı	237	φ	253	²
142	Ä	158	É	174	«	190	ı	206	Ł	222	ı	238	ε	254	■
143	Å	159	ı	175	»	191	ı	207	Ł	223	ı	239	∩	255	

Source: [www.LookupTables.com](http://www.LookupTables.com)

Figure 10-10 ACSII extended characters

#### 10.4 • Storing and Displaying Characters

In programming, you specify an ASCII character value by using single quotes. The computer does not care how you represent the number, it always stores the binary value – it is just a matter of convenience to use a character explicitly. All three of the following

are equivalent and result in the SAME value stored in the variables:

```
u8 u8Char1 = 'A';  
u8 u8Char2 = 65;  
u8 u8Char3 = 0x41;
```

No quotes, single quotes, and double quotes are totally different. The following are NOT equivalent and all result in DIFFERENT values stored in the variables:

```
u8 u8Char1 = 8;  
u8 u8Char2 = '8';  
u8 u8Char3 = "8"; /* Compiler error */
```

The `u8Char1` variable has the binary value `b'00001000'` stored in it (literally the value eight). If you check the ASCII table, the decimal value 8 is the backspace character so if you sent this to a terminal, the cursor would move back one space. The ASCII character code for the number 8 is 56 (0x38), so the value in memory of `u8Char2` above is 56. The number 8 would be printed in a terminal window if this was sent. The value `"8"` is a C-string, which is a null-terminated character array of length 2 {0x56, 0}, so it would be illegal to try and make the assignment as shown.

This extends to the problem of sending numbers to character displays like a computer's RS-232 terminal program or an LCD screen. Consider the following:

```
u32 u32Number = 1313820741;
```

How do you print that to an ASCII-based display terminal? The variable `u32Number` is a number stored in binary in 4 bytes on your embedded system. In hex, the value is `0x4E4F5045`. If your first thought is to just send the bytes, you would break it up byte-wise and send `0x4E`, `0x4F`, `0x50` and `0x45`. Using the ASCII table, you just sent N, O, P, E. Not exactly what you intended, no doubt.

When an embedded system is attached to a terminal program for debugging, the default display mapping is ASCII. That means that whatever byte comes through the serial port, that 8-bit binary value is converted to a single character based on the ASCII code table being used. So, if you really want 1313820741 to appear on the screen, you must parse out the number into individual ASCII digits. For the example number, you would send 10 bytes in total for the ASCII characters '1', '3', '1', '3', '8', '2', '0', '7', '4', and '1'. Likewise, if you were typing that same number in a terminal program and wanted the binary number to be stored in your embedded system, the 10 ASCII characters would be received and then built up into the binary equivalent.

Because of all this, working with ASCII can be a bit of a pain. Sending and storing data in ASCII is also inefficient by at least a factor of two. Fortunately, there are some common library C functions that can take care of translating ASCII to integers and integers to ASCII. The functions are `atoi()` and `itoa()`. The first of the two, `atoi()`, is part of the standard C library so including `stdlib.h` should make it available. It takes a pointer to a string and returns an integer. There is also an `atol()` function so you can work with 32-bit values. In IAR, `atoi()` will properly return a 32-bit number.

The `atoi()` function is relatively easy to code. The key is that the string you provide is a null-terminated array of numerical ASCII codes. The variable `u32Number`, as shown above, could, in fact, be initialized like this:

```
u32 u32Number = atoi("1313820741");
```



A great exercise is to try to write your own version of `atoi()`. Do that now, at least in pseudo code. Note that specifying a string constant as shown in the example qualifies as providing a pointer-to-string.



The second function, `itoa()`, is not part of the standard C library though it might be present in your favorite compiler's library. It is not standard to C because there is a bit of a problem that should be obvious to you. Think about how you would implement `itoa()` and what is the issue that results?

The issue is strings in C, namely the fact that there is no string type in C. By definition, a C-string is a null-terminated array of char. A function like `itoa()` will result in a character array anywhere from 1 digit to (assuming 16-bit integers for now) 5 digits since 65535 is the maximum 16-bit integer. Dealing with variable string lengths implies dynamic memory allocation but having a standard C function that uses the heap is a disaster waiting to happen.

You cannot create the array as a parameter of the function because the memory on the stack in the activation record of that function goes away when the function returns. The alternative could be to always return a 5-digit character array, but that is not a valid return type of a function. So, you are stuck with creating an array of the maximum possible size needed prior to calling the function and passing a pointer to that array. The function still must manage the varying lengths of the numbers. One way is to add leading zeros for numbers with less than 5 digits and then you would need to manage those leading zeros in your user interface since most of the time people do not like to see them.

There seems to be a lot of trouble surrounding ASCII, strings, numbers and user interfaces, and unfortunately, there are no easy solutions. Things are slightly better in C++ since there is a string type and `itoa()` is part of `stdlib`, but rest assured there is a ton of code behind that, which can eat up precious space on your embedded system. You have to decide what you need and choose the best way to get that. There are plenty of examples of ASCII handling in this chapter, so by the time you are finished, you should have a pretty good handle on it all.

At this point, we will add some ASCII-related functions and constants to our `utilities.c` library to help, too.

- **`u8 ASCIIHexCharToNum(u8 u8Char_)`**: Determines the numerical value of a hexadecimal ASCII char of that number. '0' to 'F' or '0' to 'f' is returned as 0 - 15.
- **`u8 HexToASCIICharUpper(u8 u8Char_)`**: Returns the upper-case ASCII char of a single digit number. 0 - 15 -> '0' - 'F'
- **`u8 HexToASCIICharLower(u8 u8Char_)`**: Returns the lower-case ASCII char of a single digit number. For example, 0 - 15 -> '0' - 'f'
- **`u8 NumberToAscii(u32 u32Number_, u8* pu8AsciiString_)`**: A version of `itoa` written specifically for the EIE system. Converts a long into an ASCII string. Maximum of 10 digits + NULL.
- **`bool SearchString(u8* pu8TargetString_, u8* pu8MatchString_)`**: Searches a string for another string. Finds only an exact match of the string (case sensitive).



The details of the implementation are left to the reader as an exercise. It's important to point out that the offset between the ASCII digits and their binary equivalents is fixed. The constant "NUMBER\_ASCII\_TO\_DEC" holds this value. You can verify it and the other constants defined in `utilities.h` using the ASCII code table.

## 10.5 • SAM3U2 UART Peripheral

The SAM3U2 family has several peripherals able to speak RS-232 and other serial protocols. There is a very basic UART (Universal Asynchronous Receiver Transmitter), and a more complex peripheral called a USART (Universal Synchronous / Asynchronous Receiver Transmitter) with three instances. In other words, you could configure up to four RS-232-style peripherals on the SAM3U2.

 Open the User Guide and find both the UART and USART sections. Starting with the simpler UART, once again begin the steps of learning how to use it:

1. Read the whole peripheral description section making sure to note relevant features.
2. Examine the block diagrams to get an idea about the hardware and logical relationships.
3. Study the user interface to determine the registers required for configuration and usage of the peripheral.
4. Code the low-level driver and test thoroughly.

Look at the user interface registers for both the UART and USART and notice that the UART is just a subset of the USART. In designing the driver, we carefully checked that a common code base could be used across both peripherals, so we wouldn't have to code separate tasks to manage RS-232 functionality in both.

UART		USART	
Offset	Register	Offset	Register
0x0000	Control Register	0x0000	Control Register
0x0004	Mode Register	0x0004	Mode Register
0x0008	Interrupt Enable Register	0x0008	Interrupt Enable Register
0x000C	Interrupt Disable Register	0x000C	Interrupt Disable Register
0x0010	Interrupt Mask Register	0x0010	Interrupt Mask Register
0x0014	Status Register	0x0014	Channel Status Register
0x0018	Receive Holding Register	0x0018	Receiver Holding Register
0x001C	Transmit Holding Register	0x001C	Transmitter Holding Register
0x0020	Baud Rate Generator Register	0x0020	Baud Rate Generator Register
0x0024 - 0x003C	Reserved	0x0024	Receiver Time-out Register
0x004C - 0x00FC	Reserved	0x0028	Transmitter Timeguard Register
0x0100 - 0x0124	PDC Area	0x2C - 0x3C	Reserved
		0x0040	FI DI Ratio Register
		0x0044	Number of Errors Register
		0x0048	Reserved
		0x004C	IrDA Filter Register
		0x0050	Manchester Encoder Decoder Register
		0xE4	Write Protect Mode Register
		0xE8	Write Protect Status Register
		0x5C - 0xFC	Reserved
		0x100 - 0x128	Reserved for PDC Registers

Figure 10-11 UART and USART user interface registers



10.5.1 • Peripheral Highlights

The simple UART is indeed very simple. The details described in the user guide essentially repeat everything that we have talked about for the RS-232 protocol with details of how the peripheral runs to implement it. From the block diagram, we can see the PDC connection and clock input MCK. MCK will be divided down in the Baud Rate Generator which will be the only mildly complicated part of setting up the peripheral.

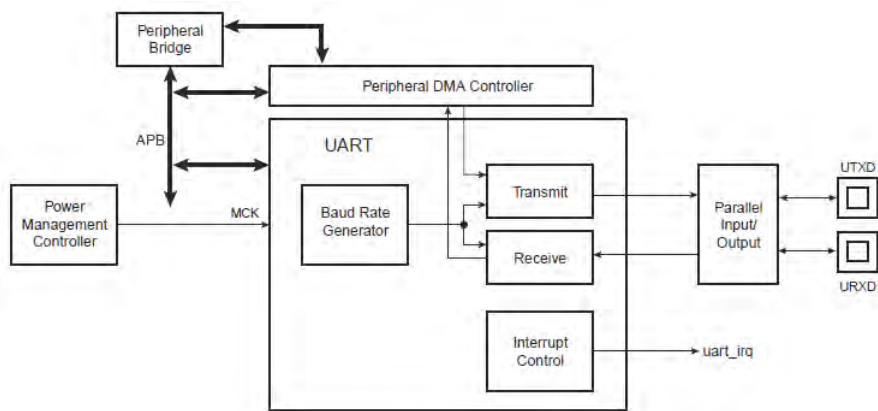


Figure 10-12 UART block diagram

There are only two available pins to the simple UART and they are managed through the PIO controller. This UART is attached to the Blade Daughter Board connector. The setup values that were configured way back in the GPIO chapter have already correctly set the pins to be under peripheral control.



Figure 10-13 UART connect to the Blad Daughter Board

The USART peripheral has many more features, some of which will be discussed in the next chapter as the USART is configured for the SPI/SSP protocol. You can see all the available configurations that are based on the USART peripheral.

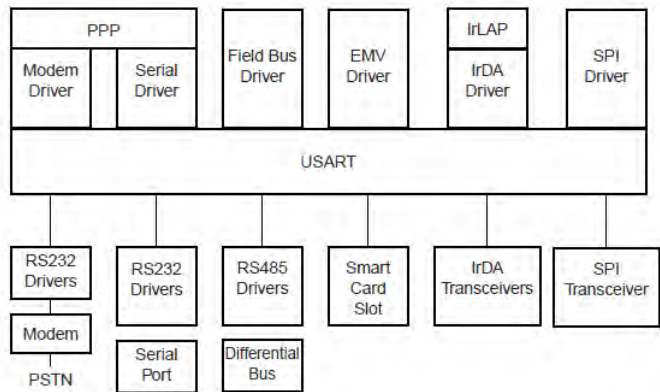


Figure 10-14 USART hardware supported protocols



The USART is configured with the user interface registers to operate per the intended protocol and will then handle a lot of the signaling details automatically. It is still up to the user to write code to feed and retrieve data from the peripheral and use it properly in an application. The user code must monitor the peripheral status bits to make sure that the peripheral has not detected an error.

There are 4 USART peripherals available on the SAM3U2. From the hardware design, we can see that USART0 is used for the “debug” RS-232 connection that connects the EiE development board to a PC.

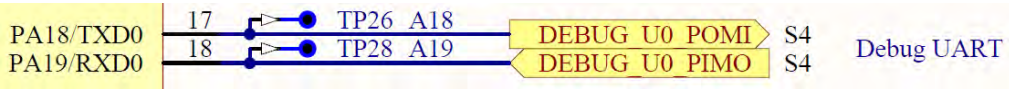


Figure 10-15 USART debug connections

On the latest versions of the dev board, these lines run into the J-Link OB processor that has its own built-in USB-to-RS232 converter and thus provides the serial port functionality via the programming USB connection. Later we will write our own USB-to-serial driver so the main processor can speak directly through USB to a computer.

The simple UART and USART peripherals each have their own line to the NVIC and connections to the PDC. These are both critically important to our implementation of the peripherals and connection into the message task.

### 10.5.2 • Baud Rate Generator

No matter what processor you use, configuring a communications peripheral will require setting up the data clock. Since we don’t know what will be connected to the Blade UART, we will focus on configuring the USART0 clock for the Debug port. The USART peripheral offers what is called a “fractional baud rate” generator which includes special hardware that ultimately results in a more accurate clock because it provides additional frequency division beyond just an integer division of MCK. This is only available when the USART is used in asynchronous mode which makes sense because synchronous systems where the clock is controlled by only one side of the communication would probably never care about the absolute clock speed even if it was drifting all over the place.

The block diagram for the USART fractional baud rate generator looks complicated:

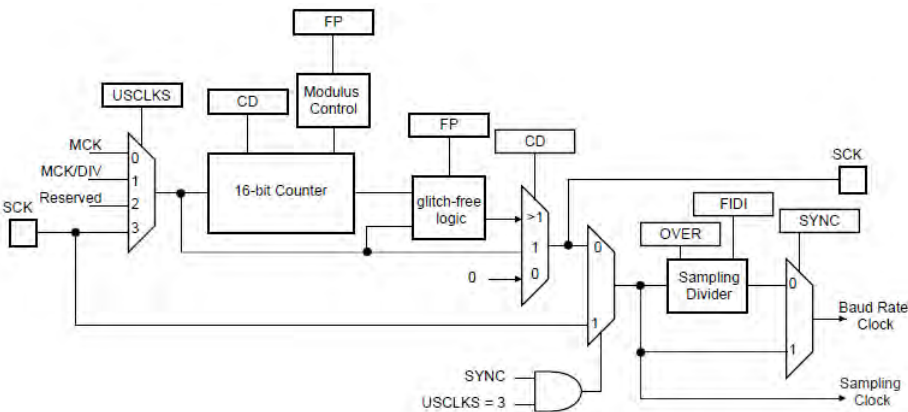


Figure 10-16 USART fractional baud rate generator block diagram

It is easier just to look at the equation and then follow through the block diagram if you want to understand how it works.

$$\text{Baudrate} = \frac{\text{SelectedClock}}{\left(8(2 - \text{Over})\left(\text{CD} + \frac{\text{FP}}{8}\right)\right)}$$

For the RS-232 (asynchronous) application, we will make use of the fractional baud rate generator to get us as close to 115,200bps as possible. There are three parameters, “Over,” “CD,” and “FP” that need to be determined to get the correct rate. The datasheet explanation could be clearer about these various parameters, but with a bit of digging it can be figured out. Between the user guide section on the baud rate generator and the user interface description, we determine the meaning of the symbols in the above equation.

- **“Over”** references a single bit OVER found in the US\_MR register. It controls whether the peripheral will take 8 samples or 16 samples of the input signal. For this application, more samples are better and will not cause any issues with the baud rate we need, so we will keep this bit as 0 which equates the value of 0 to “Over” in the equation. OVER needs to be set for very high-speed baud rates to reduce the number of samples since each sample must be instigated by a clock.
- **“CD”** stands for Clock Divider and is the main integer divider that will first divide MCK down from 48MHz. The CD bits are found in the US\_BRGR register. There are 16 bits that will be taken at their integer value to divide MCK.
- **“FP”** is the “Fractional Part” and is also found in the US\_BRGR register. These are just three bits that define the adjustment that could be made to tune the clock division slightly. Three bits gives 1/8 fractions within a whole clock period (0.125 per bit).



Calculate the values for the three variables using the equation to get 115,200 baud. Since you end up with two unknowns and just one equation, there’s a little bit of guess-and-test. Hint: perhaps the fractional part isn’t helpful in this case.

In our solution, we solve for CD as a function of FP assuming OVER = 0.

```
BAUD = MCK / (8(2-OVER)(CD + FP / 8))
=> CD = (MCK / (8(2-OVER)BAUD)) - (FP / 8)
BAUD desired = 115200 bps
=> CD = (48e6 / (8(2-0)115200)) - (FP / 8)
=> CD = 26.042 - (FP / 8)
```

The smallest FP value is 0.125 for the fractional part which would give 26.125 for the overall divide factor. 26.000 is closer to 26.042 than 26.125, so choosing FP = 0 makes more sense. Keep these values in mind for when the register settings are described.

The user guide provides additional details for other modes of operation for the USART. Keep an eye on the heading levels for each topic to ensure you are reading relevant information for the mode of operation being configured. For example, the “Receiver and Transmitter Control” applies completely, but some sections under “Synchronous and Asynchronous Modes” will not apply. If you are unsure if a section applies, read through it. At the very least, it can give you ideas about the capabilities of the peripheral and features of other protocols.

Pay close attention to the different error flags that are available. Parity checking is an easy way to add basic error control. Framing errors are important to check for in RS-232

as it provides important information about how the system is running. As you read these details, you can start imagining the flow and logic of your peripheral driver. Make notes, draw pictures, ask questions about the design as it begins to form. How will data flow? Will it be fast enough? What are the risks? How will it perform in a multi-tasking environment and with the 1ms system loop?

Hardware handshaking should be understood even though it is not used for the EiE debug UART. As mentioned earlier, fast embedded systems are unlikely to be resource-constrained when it comes to typical RS-232 data rates. The SAM3U2 could theoretically support 6Mbps transmit clock in asynchronous mode and still the dominating factor is the peripheral's minimum required 8x sampling on each bit. It is more likely that a slower device downstream may have hardware flow control, but it seems very rare these days.

Ignore the descriptions of the other protocols and jump ahead to the Test Modes section. There are four test modes that can change the Tx and Rx signal routing using firmware. This is great for testing code or doing device self-testing in production or even in the field. While the EiE board and all practical devices would use Normal mode, having the capability to run test modes just with a firmware-controlled change is invaluable and would make a great exercise. "Loopback" mode is equivalent to wiring the output of the transmitter to the input of the receiver. This is a good way to check full duplex operation of the peripheral driver.

## 10.6 • UART Registers

The common registers to the standard UART and more advanced USART configure and manage the asynchronous RS-232 communication that the development board will provide. There are a few things to note about the base addresses for the UART peripherals. The simple UART is referred to as the "Debug UART" or "DBGU" in the AT91SAM3U4 header file. The EiE firmware system will use USART0 as our "Debug" UART so watch out that the two are not confused. The PDC is integral to communication peripherals, so every communication peripheral has an explicit base address to the PDC register block associated with it. This was introduced in the previous chapter. Therefore, the complete collection of base addresses available are as follows (US3 is not available on the SAM3U2):

#define AT91C_BASE_PDC_DBGU	(AT91_CAST(AT91PS_PDC)	0x400E0700) // PDC_DBGU
#define AT91C_BASE_DBGU	(AT91_CAST(AT91PS_DBGU)	0x400E0600) // DBGU
#define AT91C_BASE_PDC_US0	(AT91_CAST(AT91PS_PDC)	0x40090100) // PDC_US
#define AT91C_BASE_US0	(AT91_CAST(AT91PS_USART)	0x40090000) // US0
#define AT91C_BASE_PDC_US1	(AT91_CAST(AT91PS_PDC)	0x40094100) // PDC_US1
#define AT91C_BASE_US1	(AT91_CAST(AT91PS_USART)	0x40094000) // US1
#define AT91C_BASE_PDC_US2	(AT91_CAST(AT91PS_PDC)	0x40098100) // PDC_US2
#define AT91C_BASE_US2	(AT91_CAST(AT91PS_USART)	0x40098000) // US2
#define AT91C_BASE_PDC_US3	(AT91_CAST(AT91PS_PDC)	0x4009C100) // PDC_US3
#define AT91C_BASE_US3	(AT91_CAST(AT91PS_USART)	0x4009C000) // US3

Configuring the UART peripherals is the most onerous configuration we have done so far. One reason is that the EiE firmware is written to access the peripherals generically, so all the peripherals will be set up here. Since the driver will apply to multiple development boards, the register initialization values will be placed in configuration.h. This file can be considered a level of abstraction above the board-specific file, but not completely generic to the UART driver. It was difficult to decide exactly how this should work, so you can debate the decisions made.



Make sure you have the starting code loaded from the `uart_debug` branch. This branch includes both the `uart` driver source files and the `debug` application source files starting points. The register descriptions and init values will focus on the more complicated USART

peripheral but are essentially directly applicable to the simple UART. Since USART0 is going to be our DEBUG UART resource, the INIT values all start with DEBUG\_.

**USART Control Register (US\_CR):** Bit-wise access to control the peripheral. The receiver and transmitter in the peripheral are controlled separately. The peripheral can be reset, and various status bits and flow control hardware lines can be accessed (CTS, RTS, and CS) since they are under the peripheral's control and not the PIO controller. The initialize value enables the transmitter and receiver.

```
#define DEBUG_UART_US_CR_INIT (u32)0x00000050
```

**USART Mode Register (US\_MR):** The main configuration values for the peripheral. The bottom 4 bits set the protocol operation which for RS-232 is considered "Normal" mode. The clock source and data length are chosen as well as synchronous/asynchronous and stop bits. Some of the configuration bits are protocol-specific and will be used in the next chapter. Bits that do not apply to the protocol of interest can be left at 0. The configuration word we chose selects Normal mode, 8-bit, 1 stop bit, no parity.

```
#define DEBUG_UART_US_MR_INIT (u32)0x000008C0
```

**USART Interrupt Enable / Disable / Mask Register (US\_IER / US\_IDR / US\_IMR):** A USART peripheral has many interrupt sources. Data reception will rely almost entirely on interrupts based on the ENDRX signal from the PDC. Transmitted data will also use the PDC so that interrupt will be necessary. The planned implementation for this configuration will start with just the ENDRX interrupt enabled, so that is the only interrupt set in the INIT value. The mask register shows the currently enabled interrupts.

```
#define DEBUG_UART_US_IER_INIT (u32)0x00000008
```

**USART Channel Status Register (US\_CSR):** This status register contains flags related to the same events that drive the interrupts plus a few more. There are some subtle differences between the flags such as ENDRX and RXBUFF, so read carefully. Your application design can use these in different ways. When checking interrupts that have occurred, check the bit in the CSR and make sure the corresponding bit in the IMR is set. The CSR bits will set and clear based on their associated rules regardless of whether there is a corresponding interrupt available/enabled. This is a read-only status register, so it does not get initialized.

**USART Receiver Holding Register (US\_RHR):** The latest value received by the peripheral. If the UART is being used without the peripheral DMA, this is the register that is read to get the received value. The PDC automatically reads this register and puts the value at the configured destination pointer. The USART has an additional bit that the UART does not have related to one of the available protocols. Read-only, no initialization.

**USART Transmit Holding Register (US\_THR):** The current value that will be written to the transmit hardware in the peripheral. If the UART is being used without the peripheral DMA, this is the register that is written with the next byte to send. The PDC automatically writes this register using the next value at the configured source pointer. The USART has an additional bit that the UART does not have related to one of the available protocols. Do not write to this register during initialization.

**USART Baud Rate Generator Register (US\_BRGR):** The calculated CD and FP values that were determined earlier in the chapter are added here. The simple UART only has the CD value available, but that does not affect the EiE system since FP = 0.

```
#define DEBUG_UART_US_BRGR_INIT (u32)0x0000001A
```

**USART Receiver Time-out Register (US\_RTOR):** A hardware timer that can be used to monitor reception delays/timeouts. EiE will not use this register or the related events.

**USART Transmitter Timeguard Register (US\_TTGR):** A hardware timer that can be used to monitor transmission delays/timeouts. EiE will not use this register or the related events.

**USART Write Protect Mode / Status Register (US\_WPMR / US\_WPSR):** Protection against accidental writes to the USART register group. The protected registers are listed. Protection is totally on, or totally off, but individual register access attempts are flagged in WPSR. Like other write-protected peripheral registers, this ability is part of very robust firmware design. It will not be used in EiE.

The remaining registers are protocol-specific and will not be discussed.

Add a bookmark at the top of configuration.h to connect to the UART configuration values.

```
9  Bookmarks:
10 ##### Communication peripheral board-specific parameters
```

**Figure 10-17** Bookmark added to the top of configuration.h

The BLADE UART register values are chosen identically to the DEBUG UART. They have been added in without further discussion.

#### 10.6.1 • EiE UART Driver

Using a UART peripheral directly is fairly simple. Transmission involves loading THR and waiting until the flag is set (or the interrupt occurs) to indicate the data has been sent before loading the next byte or being complete. Reception is very similar, where receive flags or interrupts can be monitored and the program reads RHR to get the received byte. As long as this is done quickly enough, incoming bytes will not be overwritten. Robust systems will make use of all the available error checking and status flags to make the best system possible.

Using the PDC to directly load from or read to the USART registers and RAM is just as simple but with an added layer of conceptual difficulty. However, as soon as you understand the PDC mechanism of operation and see how to use it effectively then you can write a very efficient system.

When a communication peripheral is used in a multi-tasking system then things start to get difficult because the processor cannot dedicate itself to one message and one peripheral. While DMA and interrupts can get around this problem almost entirely for a single task, the problem of multiple tasks wanting to use the same peripheral is not as trivial to solve. An easy way is to use a semaphore to lock out other tasks if the peripheral resource is busy. This puts the onus on the client tasks to take action and wait, retry, or give up. A high-level task may not have enough information about how the low-level peripherals work, so expecting a task to run properly may be asking a lot. In an RTOS-based system, this is much easier because a task can just block until it can take the semaphore and the scheduler will still run the rest of the system.

Since each UART is physically connected to a hardware resource, there is often just one task that accesses it. However, for a hardware resource like an LCD or debug output, it is actually very likely that multiple tasks will want to use that same resource. The system must be designed accordingly. In the case of the EiE firmware, we will insert an additional task between the UART peripheral for debug output and the high-level tasks that may want to use it to solve this problem.

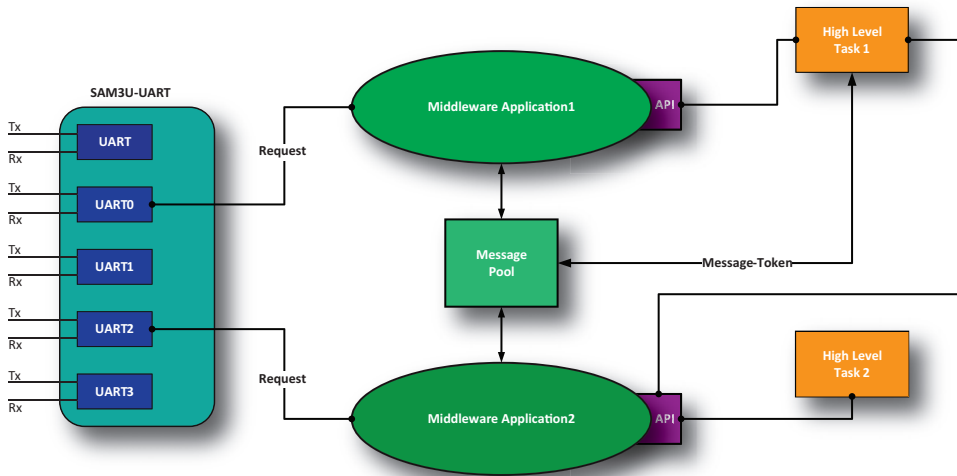


Figure 10-18 Multitasking to system UART resources

This is where the message queue comes in. By allowing any task to queue a message it wants to send (or receive) to the debug UART, the task can operate independently of the peripheral. The big problem, in this case, is knowing when a message has been sent. Now you will see how the token system will be used to assign values to unique messages and therefore allow the owner of a message to check the status.

The framework that we set up for our UART driver is distinctly different for transmit vs. receive. However, each of the available UART peripherals will work identically. The middleware task will take control of the peripheral through a Request function and be in charge of configuring the peripheral. It needs to link transmit and receive buffers to the messaging task to close the loop and provide API functions to get data in and out.

### 10.6.2 • UART Task Data Structures



We define two new struct types to help. The first will be used by client tasks to specify their data buffers and an Rx callback function.

```

/*!
@brief Task-provided parameters for a UART
*/
typedef struct
{
    PeripheralType UartPeripheral; /* Easy name of peripheral */
    u16 u16RxBufferSize;          /* Size of receive buffer in bytes */
    u8* pu8RxBufferAddress;       /* Address to circular receive buffer */
    u8** pu8RxNextByte;           /* Pointer to next byte in buffer */
    fnCode_type fnRxCallback;     /* Callback function for receiving data */
} UartConfigurationType;
  
```



The second new type will be a configuration frame for a USART peripheral. Each USART will get assigned one of these structs when the UART driver initializes.

```

/*!
@struct UartPeripheralType
@brief Complete configuration parameters for a UART resource
*/
typedef struct
{
    AT91PS_USART pBaseAddress;           /* Base address of the associated peripheral */
    u32 u32PrivateFlags;                 /* Flags for peripheral */
    MessageType* psTransmitBuffer;       /* Pointer to transmit linked list */
    u32 u32CurrentTxBytesRemaining;       /* Bytes remaining in current transfer */
    u8* pu8CurrentTxData;                 /* Current location in Tx buffer */
    u8* pu8RxBuffer;                     /* Circular receive buffer in user task */
    u8* pu8RxNextByte;                   /* Next received byte target */
    fnCode_type fnRxCallback;             /* Callback function for receiving data */
    u16 u16RxBufferSize;                 /* Size of receive buffer in bytes */
    u8 u8PeripheralId;                   /* Simple peripheral ID number */
    u8 u8Pad;
} UartPeripheralType;

```


The `u32PrivateFlags` member will be used for events or status specific to the assigned UART including a bit-wise implementation of the semaphore that prevents other tasks from directly using the peripheral.

```

/* u32PrivateFlags in UartPeripheralType */
#define _UART_PERIPHERAL_ASSIGNED (u32)0x00000001 /* Set when peripheral in use */
#define _UART_PERIPHERAL_TX      (u32)0x00200000 /* Set when transmitting */
/* end u32PrivateFlags */

```

When a UART is requested, the task data will be copied into the existing frame and the semaphore bit set. When released, the task-specific configuration information will be reset, and the semaphore bit cleared.

 The UART driver will also need some flags to communicate information inside the driver and for debugging. These will be expanded as they are encountered in the driver design. Add the variable in the static local global section of `sam3u_uart.c`:


```
static u32 Uart_u32Flags;           /*!< @brief Application flags for UART */
```

Add the following flags section to the Constants area in `sam3u_uart.h`:

```

/* Uart_u32Flags (local UART application flags) */
/* end of Uart_u32Flags */


```

 To keep the UART driver abstracted from a particular target, some associations are added to configuration.h. The enum `PeripheralType` provides simple self-documenting names for all the communication peripherals. “SPI” is included here to be ready for the next chapter.

```

/*!
@enum PeripheralType
@brief Short names used to identify peripherals in their configuration structs.
*/
typedef enum {SPI0, UART, USART0, USART1, USART2, USART3} PeripheralType;

```

 Now associate specific names to the general peripheral typedefs for our application. Again, we jump ahead to make all the assignments for the dev board. These are bookmarked as well.

```

/***** External device peripheral assignments *****/
!!!! External device peripheral assignments
*****/
/* Peripheral assignments */
#define BLADE_UART          UART
#define DEBUG_UART          USART0
#define SD_SSP              USART1
#define ANT_SPI             USART2
#define BLADE_SPI           SPI0

```

Specific initialization values must also be associated with generic names that the driver will use to set up all the peripheral registers. Doing this allows the `sam3u_uart` source code to never be touched while still promoting easy code portability. Alternate symbols for the intended interrupt handler and the peripheral ID are set also.

```

/* Debug UART Peripheral Allocation (USART0) */
#define USART0_US_CR_INIT    DEBUG_UART_US_CR_INIT
#define USART0_US_MR_INIT    DEBUG_UART_US_MR_INIT
#define USART0_US_IER_INIT   DEBUG_UART_US_IER_INIT
#define USART0_US_IDR_INIT   DEBUG_UART_US_IDR_INIT
#define USART0_US_BRGR_INIT  DEBUG_UART_US_BRGR_INIT

#define USART0_IrqHandler    USART0_IrqHandler
#define DEBUG_UART_PERIPHERAL AT91C_ID_US0

```

### 10.6.3 • UART Driver Functions

With these definitions in place, some of the UART driver functions can be written. Look closely at the functions to get a good understanding of the data structures and how everything is getting set up to be used. Different private, protected, and public functions are introduced in an order that makes the most sense. It is more appropriate to describe a driver like this in hindsight than to attempt to explain the whole design concept and then implement it. The way that the UART peripheral driver works with its API, the messaging task, and the peripheral hardware itself is a reasonably complicated system. The good news is that the SPI and TWI (I<sup>2</sup>C) drivers work nearly identically. It is therefore very important to understand what is written here as the other communication drivers will not have this detail.

#### UartInitialize()



All that's required when the task starts is to initialize the `UartPeripheralType` parameters for each peripheral instance. Since the driver is meant to be generic, a variable for every USART is created. The first one is done for you. Repeat for the remaining.

```

/* Initialize all the UART peripheral structures */
Uart_sPeripheral.pBaseAddress = (AT91S_USART*)AT91C_BASE_DBGU;
Uart_sPeripheral.psTransmitBuffer = NULL;
Uart_sPeripheral.pu8RxBuffer = NULL;
Uart_sPeripheral.u16RxBufferSize = 0;
Uart_sPeripheral.pu8RxNextByte = NULL;
Uart_sPeripheral.u32PrivateFlags = 0;
Uart_sPeripheral.u8PeripheralId = AT91C_ID_DBGU;

```

Since each UART is treated identically, the code will be written with pointer-based access to the different `UartPeripheralType` structures. Define this pointer in the local global section and initialize it to the first UART. For some safety, declare a local global `Uart_u8ActiveUarts` that will increment every time a UART is requested and decrement when it





is released to help ensure the system is functioning properly. Clear this value, the UART flags, and set the state machine pointer to Idle.

```
/* Select the first UART peripheral and initialize flags and application pointer */
Uart_psCurrentUart = &Uart_sPeripheral;
Uart_u32Flags = 0;
Uart_u8ActiveUarts = 0;
Uart_pfnStateMachine = UartSM_Idle;

} /* end UartInitialize() */
```

### UartRequest ()

The first API function allows a task to claim a UART peripheral. The calling task needs to have a good understanding of the peripheral connections on the board to know what parameters to specify, which is another reason why a middleware task written in the EiE system makes sense. The purpose of the Request function is to check if the desired peripheral is available and then correctly fill in the peripheral configuration values.



The function starts simply by loading variables that will be used to set the appropriate registers based on the specified peripheral. Only the USART0 case is shown here. Remember that symbol names were assigned to connect the application-specific initialization values defined in configuration.h to the generic INIT names used below.

```
UartPeripheralType* UartRequest(UartConfigurationType* psUartConfig_)
{
    UartPeripheralType* psRequestedUart;
    u32 u32TargetCR;
    u32 u32TargetMR;
    u32 u32TargetIER;
    u32 u32TargetIDR;
    u32 u32TargetBRGR;

    /* Set-up is peripheral-specific */
    switch(psUartConfig_>UartPeripheral)
    {
        case USART0:
        {
            psRequestedUart = &Uart_sPeripheral0;

            u32TargetCR = USART0_US_CR_INIT;
            u32TargetMR = USART0_US_MR_INIT;
            u32TargetIER = USART0_US_IER_INIT;
            u32TargetIDR = USART0_US_IDR_INIT;
            u32TargetBRGR = USART0_US_BRGR_INIT;

            break;
        }
    }
}
```

The remainder of the function can then be generic. First, the availability of the UART is checked.

```
/* If the requested peripheral is already assigned, return NULL now */
if(psRequestedUart->u32PrivateFlags & _UART_PERIPHERAL_ASSIGNED)
{
    return(NULL);
}
```



Next, the power control bit is set, the application-specific parameters are copied to the UART configuration structure, and the INIT values are loaded to the peripheral registers.

```
/* Activate and configure the peripheral */
AT91C_BASE_PMC->PMC_PCER |= (1 << psRequestedUart->u8PeripheralId);

psRequestedUart->pu8RxBuffer      = psUartConfig->pu8RxBufferAddress;
psRequestedUart->u16RxBufferSize = psUartConfig->u16RxBufferSize;
psRequestedUart->pu8RxNextByte    = psUartConfig->pu8RxNextByte;
psRequestedUart->fnRxCallback     = psUartConfig->fnRxCallback;
psRequestedUart->u32PrivateFlags |= _UART_PERIPHERAL_ASSIGNED;

psRequestedUart->pBaseAddress->US_CR   = u32TargetCR;
psRequestedUart->pBaseAddress->US_MR   = u32TargetMR;
psRequestedUart->pBaseAddress->US_IER  = u32TargetIER;
psRequestedUart->pBaseAddress->US_IDR  = u32TargetIDR;
psRequestedUart->pBaseAddress->US_BRGR = u32TargetBRGR;
```



Although we have not yet discussed how the interrupt system will work with the UART driver, add the following code to set up and enable the DMA and interrupts. At this point, you are not expected to understand these settings, but you should understand the registers being set based on the previous discussion of the PDC.

```
/* Preset the receive PDC pointers and counters */
psRequestedUart->pBaseAddress->US_RPR =
    (unsigned int)psUartConfig->pu8RxBufferAddress;
psRequestedUart->pBaseAddress->US_RNPR =
    (unsigned int)((psUartConfig->pu8RxBufferAddress) + 1);
psRequestedUart->pBaseAddress->US_RCR  = 1;
psRequestedUart->pBaseAddress->US_RNCR = 1;

/* Enable the receiver and transmitter requests */
psRequestedUart->pBaseAddress->US_PTCR = AT91C_PDC_RXTEN | AT91C_PDC_TXTEN;

/* Enable UART interrupts */
NVIC_ClearPendingIRQ( (IRQn_Type)psRequestedUart->u8PeripheralId );
NVIC_EnableIRQ( (IRQn_Type)psRequestedUart->u8PeripheralId );

return(psRequestedUart);
} /* end UartRequest() */
```

The calling application looks to ensure a pointer is returned. If NULL is returned, that is an indication that the requested peripheral is not available. If this occurs, the task would have to try again later. This seemed like a reasonable point to require a task to confirm it has the desired resource.

### UartRelease ()

Releasing a peripheral resource is much easier. There is nothing in the system that requires a task to release a resource. The driver is documented to suggest that tasks that no longer need the peripheral return it, but nothing enforces that. It would be easy to add a timer based on an absolute time or use a counter that would track how often the peripheral was being used. An under-utilized resource could be forced back to the pool. This could be further enhanced by flagging if another task was denied a peripheral request. For now, the system is kept simple.



Write the release function based on the header information:

```

/*!-----
@fn void UartRelease(UartPeripheralType* psUartPeripheral_)

@brief Releases a UART resource.

Requires:
@param psUartPeripheral_ has the UART peripheral number to be released

Promises:
- Disables the associated interrupts
- Resets peripheral object's pointers
- Any unsent messages are dumped and set to ABANDONED status
- Main SM reset to Idle
*/

```

Once you have written your solution, compare your code to the implementation below. It is important to disable interrupts before any of the buffer pointers are disconnected. Though external hardware trying to send data to the SAM3U2 would not know that the peripheral is no longer operating, at least the firmware would not crash due to invalid pointers.

```

void UartRelease(UartPeripheralType* psUartPeripheral_)
{
    /* Check to see if the peripheral is already released */
    if(psUartPeripheral_>pu8RxBuffer == NULL)
    {
        return;
    }

    /* First disable the interrupts */
    NVIC_DisableIRQ( (IRQn_Type)(psUartPeripheral_>u8PeripheralId) );
    NVIC_ClearPendingIRQ( (IRQn_Type)(psUartPeripheral_>u8PeripheralId) );

    /* Now it's safe to release all the resources in the target peripheral */
    psUartPeripheral_>pu8RxBuffer = NULL;
    psUartPeripheral_>pu8RxNextByte = NULL;
    psUartPeripheral_>fnRxCallback = NULL;
    psUartPeripheral_>u32PrivateFlags = 0;

    /* Empty the transmit buffer if there were leftover messages */
    while(psUartPeripheral_>psTransmitBuffer != NULL)
    {
        UpdateMessageStatus(psUartPeripheral_>psTransmitBuffer->u32Token, ABANDONED);
        DeQueueMessage(&psUartPeripheral_>psTransmitBuffer);
    }

    /* Ensure the SM is in the Idle state */
    Uart_pfnStateMachine = UartSM_Idle;
} /* end UartRelease() */

```

We're assuming this function isn't called when a transmission or reception is in progress, but this could be further improved by checking for that instead of leaving it to the task. How this would be handled has many options which would have to be designed in.

Since we are discussing the driver's public functions, we can also cover the two functions used to queue a message to the driver to send.

**UartWriteData ()**

The API function to send UART data is simple because all it does is queue the data to the messaging task. Care is necessary to get the pointer notation correct when calling `QueueMessage` since `UartWriteData()` is called with a `UartPeripheralType` parameter pointer from the calling task. Try to write the function based on the header and comments below.

```

/*!-----
@fn u32 UartWriteData(UartPeripheralType* psUartPeripheral_,
                    u32 u32Size_, u8* pu8Data_)

@brief Queues an array of bytes for transfer on the target UART peripheral.

Requires:
@param psUartPeripheral_ holds a valid pointer to a transmit buffer
@param u32Size_ is the number of bytes in the data array; should not be 0
@param pu8Data_ points to the first byte of the data array

Promises:
- adds the data message at psUartPeripheral_>pTransmitBuffer
- Returns the message token assigned to the message or NULL
  G_u32MessagingFlags can be checked for the reason
*/
u32 UartWriteData(UartPeripheralType* psUartPeripheral_,
                  u32 u32Size_, u8* pu8Data_)
{
    /* Check for a valid size */

    /* Attempt to queue message and get a response token */

} /* end UartWriteData() */

```

Only a few lines of code are required. Check that the pointer code is correct and if you did it differently make sure you understand why. When writing a full system, code like this should be checked carefully with the debugger to ensure that what you think is happening is actually happening. Pointer operations are notorious for being syntactically correct and able to compile into something that is entirely not what you wanted (and can sometimes dangerously work).

```

u32 UartWriteData(UartPeripheralType* psUartPeripheral_,
                  u32 u32Size_, u8* pu8Data_)
{
    u32 u32Token;

    /* Check for a valid size */
    if(u32Size_ == 0)
    {
        return NULL;
    }

    /* Attempt to queue message and get a response token */
    u32Token = QueueMessage(&psUartPeripheral_>pTransmitBuffer, u32Size_, pu8Data_);
    if(u32Token)
    {
        /* If the system is initializing, manually cycle the UART task */
        if(G_u32SystemFlags & _SYSTEM_INITIALIZING)
        {
            UartManualMode();
        }
    }
}

```

```

}

    return(u32Token);

} /* end UartWriteData() */

```


### UartWriteByte ()

We also wrote a single-byte version of `UartWriteData()`. This costs some flash resources for the extra function definition but could save flash and processor time in the overall application especially for the Debug task.

### UartManualMode()

What you should notice in `UartWriteData()` is the section of code that calls `UartManualMode()`. The communication drivers in the system rely on the 1ms system tick just like any other application. However, many of these drivers are needed by higher-level tasks during the Initialization section of the firmware system to properly set up when the main system loop is not running. For example, the LCD initialization needs the TWI driver to be able to communicate with the LCD screen.

A solution was created to fit into certain drivers to allow API functions to trigger an emulated 1ms system loop to allow the peripheral driver to send and/or receive data. This is called “manual mode” and is a special function that runs all the necessary supporting tasks associated with what a task needs to do. The processor is still blocked in the API function that triggers manual mode, but the necessary task calls are accessed directly in the manual mode loop. Trying to sleep does not make any sense, so the 1ms loop period is just counted out using `IsTimeUp()`. By definition of our system, it is fine to block like this during initialization.

 In the case of the UART driver, the UART task and Messaging task must run to properly send the message.

```

static void UartManualMode(void)
{
    Uart_u32Flags |= _UART_MANUAL_MODE;
    Uart_psCurrentUart = &Uart_sPeripheral;

    while(Uart_u32Flags & _UART_MANUAL_MODE)
    {
        UartRunActiveState();
        MessagingRunActiveState();

        Uart_u32Timer = G_u32SystemTime1ms;
        while( !IsTimeUp(&Uart_u32Timer, 1) );
    }
}

} /* end UartManualMode() */

```

Manual mode is triggered by the `_SYSTEM_INITIALIZING` flag as part of `G_u32SystemFlags`. Add this flag in `main.h` and wrap the whole initialization section of code in it. Other tasks will make use of this for various reasons like their own requirements for manual mode.

```

void main(void)
{
    G_u32SystemFlags |= _SYSTEM_INITIALIZING;

```

```
<all initialization calls not shown>

/* Exit initialization */
G_u32SystemFlags &= ~_SYSTEM_INITIALIZING;
```

### 10.7 • UART Interrupts

By now you should have a good understanding of the general structure of the UART driver. Transmit data is queued through the driver and will wait in the Messaging Task until the driver can send the data. An application receive buffer pointer is configured where any received bytes will arrive. The PDC will be used in conjunction with send and receive interrupts to move all the data.



Since the UART interrupts are completely dedicated to the UART driver code, the interrupt service routines will be coded in `sam3u_uart.c`. Each UART/USART has its own NVIC input and thus its own vector and handler, so we can start by adding all the handler definitions. The simple UART handler is shown, and the others are identical. We very purposefully name the USARTx handlers UARTx without the 'S' since in this task the interrupts are always generated from a UART configuration. Add UART, UART0, UART1, and UART2 in the same way.

```
/*-----
@fn ISR void UART_IRQHandler(void)
@brief Handles the enabled UART interrupts for the basic UART.

Requires:
- NONE

Promises:
- Gets the current interrupt context and proceeds to the Generic handler

*/
void UART_IRQHandler(void)
{
    /* Set the current ISR pointers to UART targets */
    Uart_psCurrentISR = &Uart_sPeripheral;
    Uart_u32IntCount++;

    /* Go to common UART interrupt */
    UartGenericHandler();
} /* end Uart_IRQHandler() */
```

Based on the code, you can see that our intent is to vector to the specific handler but then launch to a common generic handler. This is possible because all the UARTs work the same and our driver will use them in exactly the same way. All that's necessary for each individual ISR is to grab the applicable UART configuration structure based on the vector to feed to the generic handler. The UART interrupts must all have the same priority levels for this to work.

A counter for each UART is included for debugging purposes that keeps track of how many times each interrupt occurs. Little tricks like this are simple but can be invaluable to check how your firmware is running. Even without a debugger running, a system can be designed with "service" or "developer" modes that can be enabled and set up to view debug variables. An ISR with a very high counter might suggest that something in the system is wrong.



Set up the GenericHandler as a private function so it is ready to code as we explain how the driver works.

```
/*!-----
@fn static void UartGenericHandler(void)
@brief Common handler for all expected UART interrupts regardless of base peripheral
*/
static void UartGenericHandler(void)
{
} /* end UartGenericHandler() */
```

Only two interrupt signals within each peripheral will be handled: ENDRX and ENDTX. These correspond directly to receive and transmit functions, respectively. With the handler structures ready, we can go through the driver design to understand what each of them must do.

## 10.8 • UART Driver Design

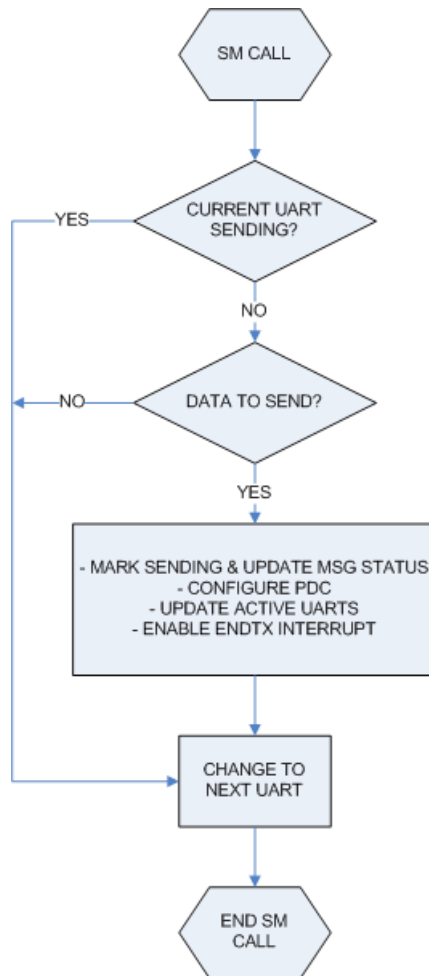
A challenge with writing a driver that transmits data from several different UART peripherals is to write generic functionality that will work with any of the UARTs just based on the UartPeripheralType data. If you remember writing UartInitialize, all the peripherals were configured in the same way and the overall intention of the driver is to avoid repeating peripheral-specific code for every UART.

Of course, there are some things that still must be specific to each UART. The trade-off to writing a generic driver is some added complexity to use switch statements, arrays, and/or pointers to access specific information in a generic routine. Transmitting data and receiving data work very differently, so we will look at each separately.

### 10.8.1 • Data Transmit

Transmitting data is probably simpler than receiving because the system usually knows how many bytes need to be sent. The system also has control over when transmission occurs. Transmitting data can, therefore, be less interrupt-intensive and can maximize efficiency with the PDC. The driver only has to worry about detecting when data is ready to be sent, then setting up the PDC to do the rest.

Transmits will be initiated by the UART state machine. The UART state machine only has an Idle state which will be used to check if any of the available UARTs in the system has data to send. Every requested UART has a transmit buffer and control flags including `_UART_PERIPHERAL_TX` that is set when the peripheral is already sending data. The Idle state checks to see if a peripheral has anything in its transmit buffer by looking if the transmit buffer pointer is NULL or not. Remember that the Tx buffer pointers point at message structures loaded there by the messaging task. It checks `_UART_PERIPHERAL_TX` to make sure that a transmission is not already in progress. Every call to the UART driver checks one of the peripherals. It cycles through the different peripherals, so in our case, it takes four main loop cycles to check every UART. As few as zero or as many as four UARTs can be busy sending at any given time. A flow chart of the system is shown.



**Figure 10-19** UART communications driver flowchart

Following the comments below, try to code the Idle state.

- Check the current UART peripheral for message activity
- If a message is ready, update the message's status and flag that the peripheral is now busy
- Load the PDC counter and pointer registers
- When TCR is loaded, the ENDTX flag is cleared so it is safe to enable the interrupt
- Update active UART count and enable the transmitter to start the transfer



Follow the flowchart and comments and trace through the code. Our goal is to have a very clear connection between the flowchart, the comments, and the code.



```

if( (Uart_psCurrentUart->psTransmitBuffer != NULL) &&
    !(Uart_psCurrentUart->u32PrivateFlags & _UART_PERIPHERAL_TX ) )
{
    /* Transmitting: update the message's status and flag peripheral is now busy */
    UpdateMessageStatus(Uart_psCurrentUart->psTransmitBuffer->u32Token, SENDING);
    Uart_psCurrentUart->u32PrivateFlags |= _UART_PERIPHERAL_TX;

    /* Load the PDC counter and pointer registers */
    Uart_psCurrentUart->pBaseAddress->US_TPR =
        (unsigned int)Uart_psCurrentUart->psTransmitBuffer->pu8Message;
    Uart_psCurrentUart->pBaseAddress->US_TCR =
        Uart_psCurrentUart->psTransmitBuffer->u32Size;

    /* When TCR loaded, ENDTX flag cleared so safe to enable interrupt */
    Uart_psCurrentUart->pBaseAddress->US_IER = AT91C_US_ENDTX;

    /* Update active UART count and enable the transmitter to start the transfer */
    Uart_u8ActiveUarts++;
    if(Uart_u8ActiveUarts > U8_MAX_NUM_UARTS)
    {
        /* Alert that the number of actual UARTs has been exceeded */
        Uart_u32Flags |= _UART_TOO_MANY_UARTS;
    }
    Uart_psCurrentUart->pBaseAddress->US_PTCR = AT91C_PDC_TXTEN;
}

```

The transmit interrupt will do the rest. The only thing left for the Idle state is to proceed to the next UART in the system for the next loop iteration. The manual mode adjustments are also made. Uart\_u8ActiveUarts is important to keep the task running in manual mode regardless of how long a message takes to transmit. At 115,200 baud, most messages will be completely sent in less than one loop cycle. However, if the message is very long or a UART is configured with a slow baud rate, then it could take many iterations of the UART task before the message is finally sent.



```

/* Adjust to check the next peripheral next time through */
switch (Uart_psCurrentUart->u8PeripheralId)
{
    case AT91C_ID_DBGU:
    {
        Uart_psCurrentUart = &Uart_sPeripheral0;
        break;
    }
    case AT91C_ID_US0:
    {
        Uart_psCurrentUart = &Uart_sPeripheral1;
        break;
    }
    case AT91C_ID_US1:
    {
        Uart_psCurrentUart = &Uart_sPeripheral2;
        break;
    }
    case AT91C_ID_US2:
    {
        Uart_psCurrentUart = &Uart_sPeripheral;
        /* Only clear _UART_MANUAL_MODE if all UARTs are done sending */
        if( (G_u32SystemFlags & _SYSTEM_INITIALIZING) && !Uart_u8ActiveUarts)
        {
            Uart_u32Flags &= ~_UART_MANUAL_MODE;
        }
        break;
    }
}

```

```

}

default:
{
    Uart_psCurrentUart = &Uart_sPeripheral;
    break;
} /* end switch */

```

The number of bytes to transmit is a known value, so only one interrupt will occur for each complete UART transmission. The messaging task provides the status of the message for the task that queued the transmission, so there does not have to be any other connections between the UART task, the ENDTX interrupt, and the client task.

Remember that both the ENDRX and ENDTX interrupts could be active and will share the UartGenericHandler ISR. The framework around each should, therefore, check that the interrupt is actively enabled in the UART mask register (IMR) and that the interrupt has actually occurred in the status register (CSR).



```

/* ENDTX Interrupt when all requested transmit bytes have been sent (if enabled) */
if( (Uart_psCurrentISR->pBaseAddress->US_IMR & AT91C_US_ENDTX) &&
    (Uart_psCurrentISR->pBaseAddress->US_CSR & AT91C_US_ENDTX) )
{
} /* end of ENDTX interrupt processing */

```

If ENDTX has occurred, that means the message was sent so the message status should be updated to COMPLETE. The correct message is accessed in the Messaging task using the message token which is saved in the transmit data structure of the message that was just sent and caused the interrupt. The message structure and data are no longer needed so can be dequeued. If more messages have been queued in the meantime, the transmit buffer pointer will now point to the next message. If no messages are waiting, the pointer will be NULL. This is also a good time to clear \_UART\_PERIPHERAL\_TX which will allow the UART state machine to start a new transmit the next time it runs since both conditions for initiating a new message transfer have been updated.



```

/* Update this message's token status and then DeQueue it */
UpdateMessageStatus(Uart_psCurrentISR->psTransmitBuffer->u32Token, COMPLETE);
DeQueueMessage( &Uart_psCurrentISR->psTransmitBuffer );
Uart_psCurrentISR->u32PrivateFlags &= ~_UART_PERIPHERAL_TX;

```

Next, we disable the UART transmitter and turn off the ENDTX interrupt so that the peripheral does not keep running and generating interrupts since the transmit buffer is now empty. This is just the opposite of what is done when the transmission is first started. It is a good idea to make sure everything that is turned on in one part of the code is turned off in the other. The connection between the ISR and the Idle function is admittedly obfuscated, but we can at least provide some clarity with a comment.



```

/* Disable the transmitter and interrupt sources that were enabled in UART idle to
start the transmission sequence */
Uart_psCurrentISR->pBaseAddress->US_PTCR = AT91C_PDC_TXTDIS;
Uart_psCurrentISR->pBaseAddress->US_IDR  = AT91C_US_ENDTX;

```

Lastly, Uart\_u8ActiveUarts is decremented. This was also adjusted in the Idle state, so we add a comment.



```

/* Decrement # of UARTs that are currently sending (incremented in UART Idle when
the transmission started) */
if(Uart_u8ActiveUarts != 0)
{

```

```

    Uart_u8ActiveUarts--;
}
else
{
    /* If Uart_u8ActiveUarts is already 0, then we are not properly synchronized */
    Uart_u32Flags |= _UART_NO_ACTIVE_UARTS;
}

```

This is a good time to take a moment to review the UART driver system diagram at the start of this chapter (Figure 10-18 on page 327). Make sure you understand all the connections that work together to facilitate a successful transfer and communicate status back to the task that queued the message.

### 10.8.2 • Data Receive

Receiving bytes is, especially in an asynchronous system like UART, a more difficult problem to solve for two reasons:

- The number of bytes coming in is unknown.
- The time when bytes come in is random unless flow control is being implemented.

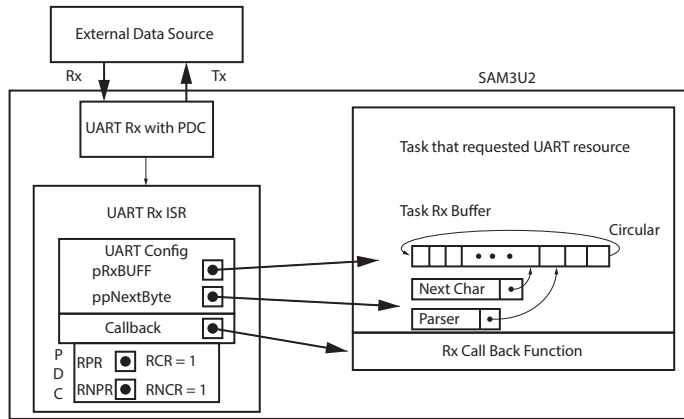
Additional overhead or protocol definition between two pieces of hardware could be defined to mitigate the two problems, but many UART systems don't have that luxury or don't want the complication. The best example comes in the next section of this chapter as we build the Debug driver that will accept keyboard input from the terminal. Any person typing data into the development board is not going to adhere to a firmware-defined protocol, and thus inputs will be entirely unsolicited and uncontrolled in speed and length.



In the EiE system, the UART driver state machine has nothing to do with data reception. We only consider the data source, the ENDRX interrupt handler, and the task that ultimately needs to get the data. Sketch out the system that we have to receive asynchronous data. See Figure 10-20 on page 341.

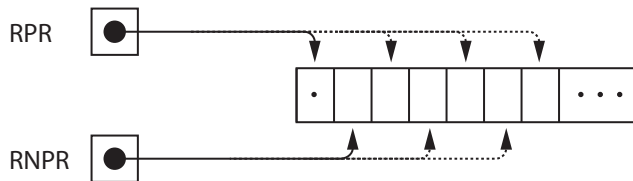
If you remember the PDC initialization values, you will have a hint for the way we decided to implement the system. The UART driver does not need to know anything about the task that is using the UART resource other than having a pointer to the task's receive buffer and a pointer to a pointer in the task that directs the ENDRX interrupt where to put the next character. The intent is that the task buffer is circular and large enough to allow data to be streaming in but still give the task enough time to process it. Because of the random (unsolicited) nature of incoming data, using a circular buffer is almost essential. The task is responsible for properly wrapping the NextChar pointer that the ISR uses. Your caution bells should be ringing at the mention of a pointer that both the task and the ISR accesses.

Since the task must see some signal when new data arrives to know it's there, it makes sense to have a callback function that the ENDRX ISR can use to communicate that the interrupt has occurred, and a new byte has been dropped into the task's receive buffer. Callbacks need to be short since they are accessed from the ISR, but that also gives them the power to operate knowing that the interrupt source is disabled while the function is running. In the Debug example, you will see that the only thing our callback function does is increment the NextChar pointer.



**Figure 10-20** System to receive asynchronous data

Managing pointers to circular buffers is one thing but doing that with DMA is a bit more challenging. To solve this in the EiE design, we decided to use 1-byte DMA transfers for all incoming serial data. While that might seem inefficient, the majority of the work is done in hardware, so it is a decent solution. The receiving DMA channel even embraces this because it offers two pointers that are used cooperatively to receive data. While one pointer is busy adjusting, the other pointer can actively receive the next byte. Since the PDC thinks that the space in the buffer it points to is only 1 byte in length, the visualization in your mind should end up looking a lot like “pointer leapfrog.”



**Figure 10-21** RPR and RNPR hop over each other

To be clear, both PDC receive pointers are pointing to the same overall receive buffer provided in the task and connected to the PDC with `UartRequest()`. However, they are configured and behave like they are pointing to their own single-byte buffers. Initializing the PDC pointers is first done when the task requests the UART resource before the `ENDRX` interrupts are enabled. Therefore, everything is ready to go on the receive side by the time the Request function returns.

Let’s start coding the `ENDRX` portion of the UART generic handler in the same way we started the `ENDTX` description. This time the `ENDRX` bits in the `IMR` and `CSR` registers need to both be set. Note that it is perfectly acceptable if both `ENDRX` and `ENDTX` flags are set inside the `ISR`.



```
/* ENDRX Interrupt when a byte has been received */
if( (Uart_psCurrentISR->pBaseAddress->US_IMR & AT91C_US_ENDRX) &&
    (Uart_psCurrentISR->pBaseAddress->US_CSR & AT91C_US_ENDRX) )
{
    /* end of ENDRX interrupt processing */
}
```

The most difficult part of the ENDRX ISR is correctly updating the DMA pointer knowing that the target buffer is circular. The concept is the same as wrapping an array pointer around, but you need to do it explicitly with pointers. This is why the task's Rx buffer size is included in the UART peripheral configuration.



```
/* Update the "next" DMA pointer to the next valid Rx location */
Uart_psCurrentISR->pBaseAddress->US_RNPR++;
if(Uart_psCurrentISR->pBaseAddress->US_RNPR ==
    (u32)(Uart_psCurrentISR->pu8RxBuffer +
    ( (u32)(Uart_psCurrentISR->u16RxBufferSize) & 0x0000FFFF ) ) )
{
    Uart_psCurrentISR->pBaseAddress->US_RNPR = (u32)Uart_psCurrentISR->pu8RxBuffer;
}
```

As the Rx handler exits, it triggers the task's Rx callback function and resets RNCR to 1 which clears the ENDRX flag and ensures the system is ready for the next received byte.



```
/* Invoke the callback */
Uart_psCurrentISR->fnRxCallback();

/* Write RNCR to 1 to clear the ENDRX flag */
Uart_psCurrentISR->pBaseAddress->US_RNCR = 1;

} /* end of ENDRX interrupt processing */
```

The reception code is simpler than the transmission code if you just consider its length and syntax, but reception involves much more initial thinking and planning. A lot of problems came up and needed to be solved to make the system handle the multitasking environment properly and to integrate with both the messaging task and any user tasks that would use the system. On a commercial design, we would build some test firmware to really stress this system to ensure that everything worked properly under worst-case conditions.

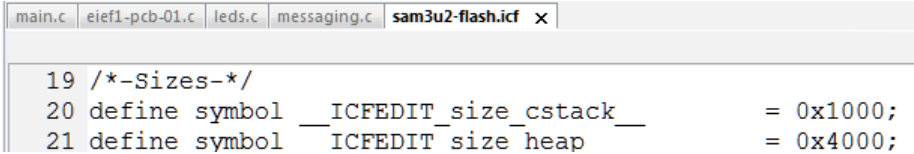
## 10.9 • Dynamic Memory Allocation

We've waited as long as possible to talk about dynamic memory allocation. We need to cover it now so we can use it in the Debug application. If you have taken a C programming course or done any C programming, you will have likely come across the malloc() function. Malloc is how memory is allocated on the "heap" which is a separate area of RAM reserved for this purpose.



*While the first letters of "dynamic memory allocation" are the same as "direct memory access," the two are totally different and unrelated concepts. Dynamic memory allocation is never abbreviated DMA.*

There is no physical difference between variables created on the stack or variables created on the heap. Both are just allocated space in the processor's available RAM. If you look at sam3u2-flash.icf, you can see the amount of memory allocated to each.



```

19 /*-Sizes-*/
20 define symbol __ICFEDIT_size_cstack__ = 0x1000;
21 define symbol ICFEDIT_size_heap = 0x4000;

```

Figure 10-22 sam3u2-flash.icf file

The size of these allocations will depend on the design of the system. We will point out right now that some companies absolutely forbid the use of dynamic memory allocation. The MISRA C standard also forbids it. That should give you an idea about what we will be saying here. There are only two tasks in the EiE system that use malloc(), the first of which we are about to introduce.

All the “normal” variables that are created in functions are kept on the stack. They are given space on the stack when the function is called, and that space is returned when the function exits. If a function tries to allocate too many variables or calls too many other functions, then eventually the stack will run out of space and you will have severe issues. In most cases this is deterministic. Recursive functions are a big problem, and interrupts can throw a wrench into the system as well if they use a lot of resources and there are a lot of them that could run in the absolute worst case of priorities and events.



“Absolute worse cases” are a very real thing and **MUST** be quantified and designed for. You have to do this for any real product that you design.

Devices running operating systems are also a problem for a slightly different reason because every task requires its own stack, and thus can consume the available resources quite quickly. For example, a processor that has 16kB of RAM available to tasks could have a maximum of 16 tasks if each task required a 1kB stack (which is a minimally safe amount of memory to allocate if it is automatically assigned).

On a side note, static variables are allocated to another part of RAM and are never destroyed. Their size does not change so they are allocated by the compiler and as long as there is space when the code is compiled (in addition to the space allocated for the stack and heap) then everyone is happy.

There are obviously a lot of considerations in the memory portion of the system design and we won’t cover that further. In short, you need to know what you are going to need and how the code will be written to ensure you have enough RAM. In pictures, it generally looks like this:

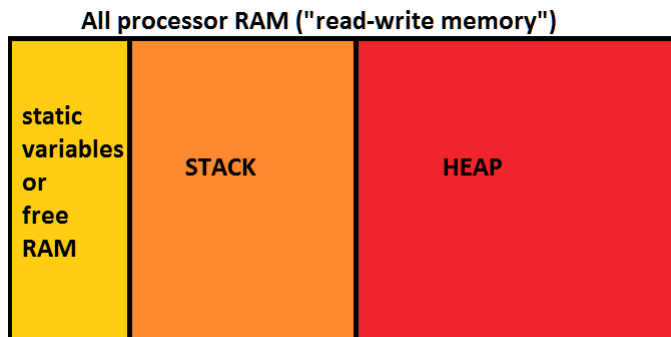


Figure 10-23 Processor RAM allocation

Dynamic memory allocation in C has always been a massive source of crazy bugs. At the most basic level of the C language, malloc uses a simple algorithm to keep track of how

much heap space is available. If there is a large enough chunk of space available, then malloc works.

The basic problem of malloc is running out of heap because there is simply not enough memory available for the worst-case scenario. This is 100% avoidable by design so it shouldn't happen unless you are careless (just like running out of stack space).

The second and very common problem with dynamic allocation is "memory leaks." It is up to the programmer to "free" any memory allocated on the heap with malloc once that memory is no longer used. If a function uses malloc and does not free the memory when it is done, every time that function is called it will allocate more space. A variation of this problem is losing the pointer to the memory area by accidentally moving it from what malloc provided and thus there is no way to free the memory.

Eventually, the heap will fill up and malloc will fail because it has no more space to give. This is a programming problem and slightly more difficult to solve especially with a lot of developers contributing code. However, it is still manageable and programs that fail due to this problem are not acceptable.

The third problem is called "heap fragmentation" and is the biggest issue with malloc. It is barely any fault of the programmers in the system because the programmer can code perfectly and never forget to free memory. However, the basic malloc function only allocates memory sequentially in the heap. If there are big blocks of the heap allocated for certain functions, other functions that are running and allocating memory must use space after the big blocks. If the big block goes away, even a single byte allocated into the resulting vacancy makes that whole block of space unavailable if the other functions that have space at the end are still running. If the heap isn't big enough, the function that just worked fine to allocate a big block of the heap no longer works. The following simple illustration should clarify:

1. The required heap size is allocated based on the maximum space required for 5 tasks plus a little extra. All the tasks startup and run no problem.
2. Task 3 and task 5 end and properly free their memory.
3. Task 5 runs again and is given the next space. Task 3 task tries to run, but there is not enough contiguous space for it to fit, so malloc fails and the world explodes.

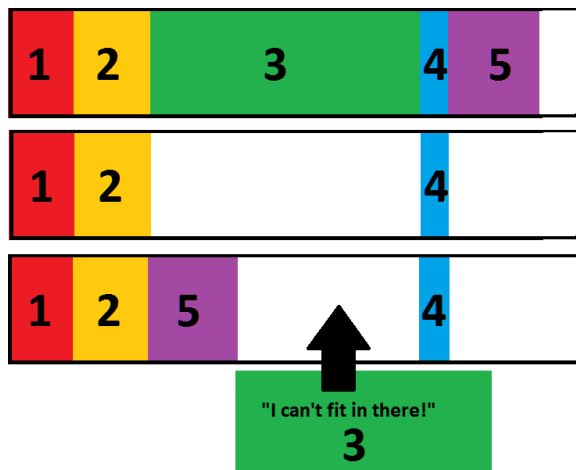


Figure 10-24 Heap fragmentation

This is a very simplistic case, but the problem is very, very real. Even one byte of dynamic allocation can make a big block of heap space unavailable to a large allocation. With a small heap and several tasks using it, fragmentation to the point of failure can occur remarkably quickly. That's a good thing because it is likely to appear during those overnight tests that you should be running all the time. The worst case is when it takes days, weeks, months, etc. to slowly fragment the heap until it is only your customers that use the product for enough time who discover the issues. What makes it even more complicated is a typical troubleshooting step is to restart the device which means everything goes back to a properly functioning system.

There are ways to get around this problem, but that involves writing a lot of code to manage memory (or you could change languages and take advantage of ways this problem has already been solved). Alternatively, you could just outlaw dynamic allocation. Many companies will not allow developers to use it.

A big question is, do you really need dynamic allocation in the first place? If you allocate enough space on the heap to handle the worst-case heap requirements in the program, why bother using the heap? Why not use static variable space so the resources are allocated specifically to the tasks that need them and the tasks don't need to worry about sharing? It is analogous to taking a room full of 10 toddlers and dropping in ten identical toys and telling everyone to get a toy. The main value is the guarantee that everyone has a toy. Ok, at least two kids will fight over the same toy, but hopefully, you get the point. Even if the child decides they're not interested in the toy, you already bought it, so there's no difference if it's being used or not.

Buying the same toy for every child may not be possible. Resources are finite, whether in the context of money to buy toys or memory in digital systems. Once again it boils down to designs and specific needs of each system. In an unbounded system where new tasks can be added with resource requirements that are not known when the system was first designed, it is essentially impossible to provide a system that will always have enough memory. Most embedded systems do not work this way and operate in a much more controlled environment where the total resource requirements are deterministic.

What we can say is that the same data structures can be built using static or dynamic memory. It might not be as fancy or convenient to use static memory, but with some effort it is successful. Dynamic allocation is minimized in the EiE system as there are enough resources to allocate memory for all the tasks in the system. The downside is that some of the buffers used could be bigger, and of course, the total requirements as the system grows could change one day. However, given the expected scope of the development board and available resources of the SAM3U2, it's a good choice. You will see a few tasks use some "local" dynamic allocation to simplify certain operations, but the memory is allocated and freed within a single function. So other than interrupts that might fire while a function is using malloc, the heap is untouched from the rest of the system's perspective. Let's make a hard rule that interrupt service routines shall not use dynamic allocation.

### 10.10 • Debug Task

The final part of this chapter introduces a middleware task that will take control of USART0 for the entirety of the system operation. Thanks to this task, we will finally get "printf-like" functionality on the development board.

Getting data in and out of an embedded system is important. Sending data out to a computer is very useful for status, debugging and data logging. Reading information is a great way to attach a much more powerful interface to the embedded system (e.g. a keyboard or script files) and can be used to control the system or send configuration data.



The Debug application does exactly this by taking control of the UART and handling all input and output. It provides a simple API to print strings out to the terminal, and read characters typed into the terminal. It does not provide any special number formatting like the C-standard `printf()` does as that becomes very complicated. The `scanf()` functionality in Debug is also very basic but has been sufficient for all applications of the EiE firmware. If you don't immediately see why those two functions are complicated, take a moment to think about what must happen in the system to make them work. Also consider that IAR has project settings dedicated to configuring `printf()` and `scanf()` from the standard library, where you can select the resources used vs. the features available.

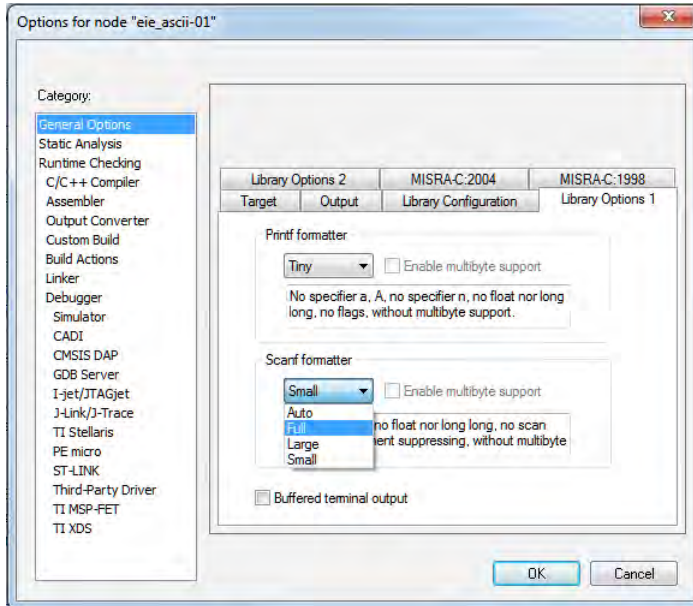


Figure 10-25 Printf and Scanf options

This task will be described starting with the main API functions because they are important for the rest of the task including initialization.

### 10.11 • Debug API

Writing and Reading data with a terminal is a straight-forward exchange of ASCII bytes. Writing data is easier to code since the size of the outgoing data is predictable and nothing needs to happen once the function is called.

The main difficulty we observe with people using the EiE Debug system is trying to queue more messages than what is available from the messaging task. This was mentioned in the Messaging chapter and some decisions were made about the number of messages and the size of the messages that would be available. Dynamic allocation would be a possible way to solve this problem, but the chances of fragmentation would always be a ticking time bomb if the system only used `malloc()` and `free()` for heap management.

#### 10.11.1 • DebugPrintf()

The UART driver and Messaging task already give us everything needed to make our version of `printf`. In fact, the function ends up being trivial. Note we chose to have

the function count the size of the string instead of asking the client to provide it as a parameter. This makes it much easier to use in the API. Also, note that the length of the string is not bounded. The messaging task will automatically split messages that are too long into multiple message slots. A production system would absolutely set either a hard limit or query the current available space. Messages deemed too long could either be truncated or simply refused.



From the declaration, you should be able to code this easily.

```

/*!-----
@fn u32 DebugPrintf(u8* u8String_)
@brief Queues the string pointed to by u8String_ to the Debug port.

The string must be null-terminated. It may also contain control characters
like newline (\n) and linefeed (\f)

Requires:
- The debug UART resource has been set up for the debug application.
- The size of the string will not exceed the total available message
  slots in the system. As a guideline, this should be less than
  10 x U16_MAX_TX_MESSAGE_LENGTH but this is not enforced.

@param u8String_ is a NULL-terminated C-string

Promises:
- The string is queued to the debug UART.
- The message token is returned

*/
u32 DebugPrintf(u8* u8String_)
{
    u8* pu8Parser = u8String_;
    u32 u32Size = 0;

    while(*pu8Parser != '\0')
    {
        u32Size++;
        pu8Parser++;
    }

    return( UartWriteData(Debug_Uart, u32Size, u8String_) );
} /* end DebugPrintf() */

```


Just like that, we have printf functionality. Having terminal output now gives a ton of new practicality to the EiE system that we can start to make use of immediately. For convenience, we also add DebugLineFeed() which is a parameter-free function that simply queues \n\r to the UART.

### 10.11.2 • DebugPrintNumber()

Sending numbers to print on the terminal involves a lot of work. Numbers stored and used in the embedded system are all binary values packed into 1, 2, or 4 bytes. These binary values mean nothing to a terminal program that is expecting ASCII characters. Therefore, any number that needs to be printed on the display must first be converted to ASCII.

DebugPrintNumber is designed to take any unsigned value up to 32-bits in length and print it to the terminal in ASCII without any leading zeroes. It is essentially a custom


version of `itoa()`. To help us out, `malloc()` will be used to get a temporary array of the right size that is needed to pass to `UartWriteData` and then be immediately freed. This meets the usage conditions for dynamic allocation that were decided as acceptable.

 The function starts by counting the number of digits in the number. The maximum number in 32-bits is just over 4 billion, so a maximum of 10 digits are needed. 0 is a special case to be handled.

```
void DebugPrintNumber(u32 u32Number_)
{
    bool bFoundDigit = FALSE;
    u8 au8AsciiNumber[10];
    u8 u8CharCount = 0;
    u32 u32Temp, u32Divider = 1000000000;
    u8 *pu8Data;

    /* Parse out all the digits, start counting after leading zeros */
    for(u8 index = 0; index < 10; index++)
    {
        /* Get the digit and add offset to get ASCII character */
        au8AsciiNumber[index] = (u32Number_ / u32Divider) + NUMBER_ASCII_TO_DEC;
        if(au8AsciiNumber[index] != '0')
        {
            bFoundDigit = TRUE;
        }
        if(bFoundDigit)
        {
            u8CharCount++;
        }
        u32Number_ %= u32Divider;
        u32Divider /= 10;
    }

    /* Handle special case where u32Number == 0 */
    if(!bFoundDigit)
    {
        u8CharCount = 1;
    }
}
```

 Once the number of digits is known, an array can be allocated dynamically for the correct size needed for the number. If we used a static array with 10 digits, handling leading zeroes would be slightly awkward. The ASCII values are written in, queued to the UART, then deallocated with `free()`.

```
/* Allocate memory for the number and copy the array */
pu8Data = malloc(u8CharCount);
if (pu8Data == NULL)
{
    Debug_u8ErrorCode = DEBUG_ERROR_MALLOC;
    Debug_pfnStateMachine = DebugSM_Error;
}

u32Temp = 9;
for(u8 index = u8CharCount; index != 0; index--)
{
    pu8Data[index - 1] = au8AsciiNumber[u32Temp--];
}

/* Print the ascii string and free the memory */
UartWriteData(Debug_Uart, u8CharCount, pu8Data);
free(pu8Data);
```

```
} /* end DebugPrintNumber() */
```

### 10.12 • Reading Character Input

Reading characters from the terminal is more difficult than writing them. More design decisions are required up front to ensure the system interacts properly with the terminal and the data coming in. The Debug state machine will be in charge of this and operate following the state diagram shown.

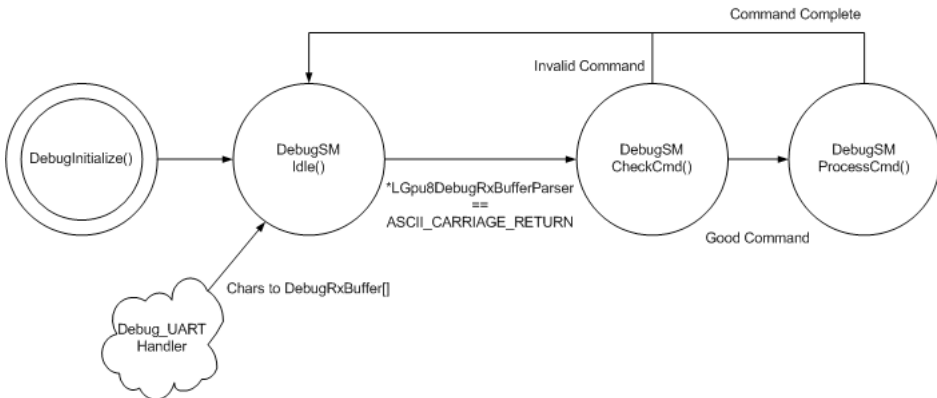


Figure 10-26 State diagram

When a character is typed in a terminal, the ASCII code is sent out the computer's serial port. In the most basic terminal programs, that's all that happens. The character is not printed in the terminal window automatically. There is usually a "local echo" option that can be enabled which causes the character that is sent out to also be displayed on the screen by the terminal. This is good because the user typing at the keyboard is expecting to see the characters being displayed as they are typed.

Unfortunately, this can be deceiving because it gives no indication that the sent characters are getting received by the device downstream. So instead of local echo, the connected system should echo the character back to the terminal. This immediately indicates that the character has been received and that the target system is functioning enough to detect the character and write it back. This is how the EiE system works.



The Debug task takes the USART0 peripheral during system initialization. All characters from the terminal will be captured and handled by the Debug task. If they are intended to go to other applications, then the API offers DebugScanf(). Before that happens, the Debug task needs to be coded to correctly work with the UART driver. First, define the local global variables that will be needed.

```
static UartPeripheralType* Debug_Uart; /* Debug UART peripheral object */
static u8 Debug_u8ErrorCode; /* Error code */

static u8 Debug_au8RxBuffer[DEBUG_RX_BUFFER_SIZE]; /* Debug's Rx buffer */
static u8 *Debug_pu8RxBufferNextChar; /* Pointer to next spot in the Rx buffer */
static u8 *Debug_pu8RxBufferParser; /* Pointer to loop through the Rx buffer */
```



Also define global flags, a buffer, and a character counter that will be used as the interaction between the Debug task and client applications. Though globals are good to avoid, the benefits of using globals for this application are greater than the downsides. An API function to access these will still be provided, but direct access proved to be very

useful for some applications. Trust that these are needed at this point, and their usage will be explained further shortly.

```

/*****
Global variable definitions with scope across entire project.
All Global variable names shall start with "G_<type>Debug"
*****/
/* New variables */
u32 G_u32DebugFlags;                /* Debug flag register */

u8 G_au8DebugScanfBuffer[DEBUG_SCANF_BUFFER_SIZE];
u8 G_u8DebugScanfCharCount = 0; /* Counter for characters in Debug_au8ScanfBuffer */

```

### 10.12.1 • DebugInitialize()



Now is a good time to add DebugInitialize(). This is the first initialize function where we can print out information to the terminal to tell the user what is going on. We will add an introduction message in debug.c and encode a firmware version message in main.h (not shown).

```

static u8 Debug_au8StartupMsg[] =
"\n\n\r*** RAZOR SAM3U2 ASCII LCD DEVELOPMENT BOARD ***\n\n\r";

```

The globals that were just defined will be initialized and the Debug UART will be requested.

```

void DebugInitialize(void)
{
    u8 au8FirmwareVersion[] = FIRMWARE_VERSION; /* from main.h */
    UartConfigurationType sUartConfig;

    /* Clear the receive buffer and initialize pointers */
    for (u16 i = 0; i < DEBUG_RX_BUFFER_SIZE; i++)
    {
        Debug_au8RxBuffer[i] = 0;
    }

    Debug_pu8RxBufferParser = &Debug_au8RxBuffer[0];
    Debug_pu8RxBufferNextChar = &Debug_au8RxBuffer[0];

    /* Clear the scanf buffer and counter */
    G_u8DebugScanfCharCount = 0;
    for (u8 i = 0; i < DEBUG_SCANF_BUFFER_SIZE; i++)
    {
        G_au8DebugScanfBuffer[i] = 0;
    }

    /* Request the UART resource to be used for the Debug application */
    sUartConfig.UartPeripheral = DEBUG_UART;
    sUartConfig.pu8RxBufferAddress = &Debug_au8RxBuffer[0];
    sUartConfig.pu8RxNextByte = &Debug_pu8RxBufferNextChar;
    sUartConfig.u16RxBufferSize = DEBUG_RX_BUFFER_SIZE;
    sUartConfig.fnRxCallback = DebugRxCallback;

    Debug_Uart = UartRequest(&sUartConfig);
    /* Go to error state if the UartRequest failed */
    if(Debug_Uart == NULL)
    {
        Debug_pfnStateMachine = DebugSM_Error;
    }
}

```

```

}
/* Otherwise send the first message, set "good" flag and head to Idle */
else
{
    DebugPrintf(Debug_au8StartupMsg);
    DebugPrintf(au8FirmwareVersion);

    G_u32ApplicationFlags |= _APPLICATION_FLAGS_DEBUG;
    Debug_pfnStateMachine = DebugSM_Idle;
}
} /* end DebugInitialize() */

```

### 10.12.2 • DebugRxCallback()



The callback function for the UART only needs to advance the Rx buffer pointer. In a moment we'll look at how the Debug task detects and reacts to newly received data.

```

void DebugRxCallback(void)
{
    /* Safely advance the NextChar pointer */
    Debug_pu8RxBufferNextChar++;
    if(Debug_pu8RxBufferNextChar == &Debug_au8RxBuffer[DEBUG_RX_BUFFER_SIZE])
    {
        Debug_pu8RxBufferNextChar = &Debug_au8RxBuffer[0];
    }
}

} /* end DebugRxCallback() */

```

### DebugScanf()



Now we have all the variables and data structures in place to write the Scanf() function. The function itself ends up being quite simple because it does not have anything to do with reading and interpreting the incoming characters. The purpose of this function is to provide a client task with the data that has been entered and somewhat vetted.

It is assumed that the global `G_u8DebugScanfCharCount` is being incremented when valid characters are received. It is also assumed that those characters are available in `G_au8DebugScanfBuffer`. The act of calling `DebugScanf()` will send the characters to the client and return the number of characters sent. It is limited to 255. The function then clears `G_au8DebugScanfBuffer` and zeroes the character counter which fits into how the Debug task is designed to operate.

```

u8 DebugScanf(u8* au8Buffer_)
{
    u8 u8Temp = G_u8DebugScanfCharCount;

    /* Copy the characters, clearing as we go */
    for(u8 i = 0; i < G_u8DebugScanfCharCount; i++)
    {
        *(au8Buffer_ + i) = G_au8DebugScanfBuffer[i];
        G_au8DebugScanfBuffer[i] = '\0';
    }

    G_u8DebugScanfCharCount = 0;
    return u8Temp;
} /* end DebugScanf() */

```

### 10.13 • Debug Programmer Access

The Debug task has two roles. In addition to the API functions and data access, the Debug application is our developer task that allows various test routines and backdoor access to anything we want. The programmer's access is provided with a menu system that is available through a custom escape sequence that the EiE firmware is built to recognize when sent from the terminal. This sequence is:

```
en+cxx
```

where xx is the command number valid from 00 to 99. Leading zeroes on the first 10 are required. "en+c" is a sequence of characters unlikely to occur in a regular input stream. It stands for "**E**ngenuics **c**ommand." Entering en+c00 prints the menu. Additional menu functions can be coded as needed depending on the development board.



The data structure for the command list is a local-global array of DebugCommandType structs. The struct is a string with the name of the command and a function pointer for the command to execute. Add the typedef in debug.h:

```
/*!
@struct DebugCommandType
@brief Required members of a Debug command.
*/
typedef struct
{
    u8 *pu8CommandName;
    fnCode_type DebugFunction;
} DebugCommandType;
```

To standardize the command structure for the Debug task, define the following constants.

```
#define DEBUG_CMD_PREFIX_LENGTH    (u8)4    /* Size of command list prefix "00: " */
#define DEBUG_CMD_NAME_LENGTH     (u8)32    /* Max size for command name */
#define DEBUG_CMD_POSTFIX_LENGTH  (u8)3     /* Size of command list postfix "\n\r\0 */
```



Define the available commands for the EiE ASCII development board. Notice the various comments that guide the user to use the system correctly. It is expected that only developers working on the core EiE firmware system would change the Debug task code, so this is very direct and low-level access. If a user application wanted its own special debugging functionality, that would typically be coded at the application level.

```
/* New commands must update the definitions below. Valid commands are in the range
00 - 99. Command name string is a maximum of DEBUG_CMD_NAME_LENGTH characters. */

#define DEBUG_COMMANDS            (u8)8      /* Total number of debug commands */

/*
#define DEBUG_CMD_NAME00          "0123456789ABCDEF0123456789ABCDEF" size reference */
#define DEBUG_CMD_NAME00          "Show debug command list"           /* Command 0 */
#define DEBUG_CMD_NAME01          "Toggle LED test"                   /* Command 1 */
#define DEBUG_CMD_NAME02          "Toggle system timing warning"      /* Command 2 */
#define DEBUG_CMD_NAME03          "Dummy3"                            /* Command 3 */
#define DEBUG_CMD_NAME04          "Dummy4"                            /* Command 4 */
#define DEBUG_CMD_NAME05          "Dummy5"                            /* Command 5 */
#define DEBUG_CMD_NAME06          "Dummy6"                            /* Command 6 */
#define DEBUG_CMD_NAME07          "Dummy7"                            /* Command 7 */
```



Add the command array in `debug.c`. By default, 8 commands are entered. Any command that is not in use is filled with a “dummy” command since something must be present.

```
/*! Add commands by updating debug.h in the Command-Specific Definitions section,
then update this list
with the function name to call for the corresponding command: */
#ifdef eie1
DebugCommandType Debug_au8Commands[DEBUG_COMMANDS] =
{ {DEBUG_CMD_NAME00, DebugCommandPrepareList},
  {DEBUG_CMD_NAME01, DebugCommandLedTestToggle},
  {DEBUG_CMD_NAME02, DebugCommandSysTimeToggle},
  {DEBUG_CMD_NAME03, DebugCommandDummy},
  {DEBUG_CMD_NAME04, DebugCommandDummy},
  {DEBUG_CMD_NAME05, DebugCommandDummy},
  {DEBUG_CMD_NAME06, DebugCommandDummy},
  {DEBUG_CMD_NAME07, DebugCommandDummy}
};
```

A command buffer and a few variables are needed as well.



```
/* Space to store chars as they build up to the next command */
static u8 Debug_au8CommandBuffer[DEBUG_CMD_BUFFER_SIZE];

/* Pointer to incoming char location in the command buffer */
static u8 *Debug_pu8CmdBufferNextChar;

/* Number of characters in the command buffer */
static u16 Debug_u16CommandSize;

/* A validated command number */
static u8 Debug_u8Command;
```

Understanding the implementation of the command functions is left as an exercise. Remember that any Debug function should still adhere to the 1ms system loop rule, so you may have to limit the command function in `debug.c` to just setting a flag in `G_u32DebugFlags` or calling an API function of a different task. Since this is low-level programmer access, breaking some system rules is much more acceptable. Remember that the rest of the board functionality still depends on the 1ms system loop running to execute all the board services.

### Passthrough Mode

Some of the applications you write need to receive all characters directly and may also want to use the terminal as part of the task. In this case, you would not want the Debug command processor to be complaining of invalid input or limiting the characters that can be entered or passed on. Therefore, the command-processing portion of the Debug application can be disabled to allow terminal data to pass through directly to the task. This kind of functionality is thus typically referred to as “passthrough mode.”

User tasks can manage passthrough mode via two API functions, `DebugSetPassthrough()` and `DebugClearPassthrough()`. The “Set” function is shown here.

```
void DebugSetPassthrough(void)
{
    G_u32DebugFlags |= _DEBUG_PASSTHROUGH;

    DebugPrintf("\n\n\r***Debug Passthrough enabled***\n\n\r");
} /* end DebugSetPassthrough */
```





The code simply sets a flag that the Debug task uses to skip certain functions that you have already seen in the Debug task. `DebugClearPassthrough()` is identical but clears the flag. Make sure both functions are added in.

### DebugSM\_Idle

Most of the work for the Debug task is done in the Idle state. The state knows when a character has been received because the local `Debug_pu8RxBufferParser` pointer will not be aligned with `Debug_pu8RxBufferNextChar` whenever the latter is moved by the `ENDRX` handler. Any new input characters will be processed immediately ensuring that even high input data rates will not lead to missed characters due to the 1ms loop time delay between buffer checks.



Set up the Idle state including the four variables shown that will be needed for the function.

```
void DebugSM_Idle(void)
{
    bool bCommandFound = FALSE;
    u8 u8CurrentByte;
    static u8 au8BackspaceSequence[] = {ASCII_BACKSPACE, ' ', ASCII_BACKSPACE};
    static u8 au8CommandOverflow[] = "\r\n*** Command too long ***\r\n\n";
} /* end DebugSM_Idle() */
```

Checking the characters is done char-by-char as long as there are unread characters in the buffer, and as long as the characters processed do not form an allowable command in the debug system at which point character processing will stop and the task will advance to another state. Most characters are simply saved in a buffer, but there are a few special characters that must be handled differently. Since there might be future characters that need to have special action, the special characters are handled with a switch statement. Set up the processing framework first with the two special cases (backspace and carriage return) and default case.



```
/* Parse any new characters until no more chars or a command is found */
while( (Debug_pu8RxBufferParser != Debug_pu8RxBufferNextChar) &&
        (bCommandFound == FALSE) )
{
    /* Grab a copy of the current byte */
    u8CurrentByte = *Debug_pu8RxBufferParser;

    /* Process the character */
    switch (u8CurrentByte)
    {
        /* Backspace: update command buffer and send delete sequence to terminal */
        case(ASCII_BACKSPACE):
        {
        }

        /* Carriage return: change states to process new command */
        case(ASCII_CARRIAGE_RETURN):
        {
        }

        /* Add to command buffer and echo */
        default:
        {
        }
    }
}
```

```
    } /* end switch (u8RxChar) */
```



Each of the above cases involves important discussion. Backspace is a control character that comes in as 0x08. If the Debug task is running in Passthrough mode, then this character should just be put into the Scanf buffer.

```
    if( G_u32DebugFlags & _DEBUG_PASSTHROUGH )
    {
        if(G_u8DebugScanfCharCount < DEBUG_SCANF_BUFFER_SIZE)
        {
            G_au8DebugScanfBuffer[G_u8DebugScanfCharCount] = u8CurrentByte;
            G_u8DebugScanfCharCount++;
        }
    }
}
```

When the regular Debug task is running, backspaces sent from the terminal should do what you expect backspace to do: delete the last character you typed. Take a moment to consider what that means for both the Debug task and the terminal.



In the Debug task, the buffer holding incoming characters must have the latest character removed. There are two buffers being updated: the `G_au8DebugScanfBuffer`, and `Debug_au8CommandBuffer`. We keep NULLs in the scanf buffer when no other characters are present since a NULL is as close to empty as you can get. It's also non-printable and if the buffer is considered a string, would make sure that any string functions would work.

```
else
{
    /* Process for scanf */
    if(G_u8DebugScanfCharCount != 0)
    {
        G_u8DebugScanfCharCount--;
        G_au8DebugScanfBuffer[G_u8DebugScanfCharCount] = '\0';
    }
}
```



The command buffer is also processed. The command pointer and the command size counter need to be adjusted if they're not already at the beginning of the buffer.

```
/* Process for command */
if(Debug_pu8CmdBufferNextChar != &Debug_au8CommandBuffer[0])
{
    Debug_pu8CmdBufferNextChar--;
    Debug_u16CommandSize--;
}
}
```



In all cases, inputting a backspace character needs to make the last terminal character disappear. Not even this is easy, which is why the `au8BackspaceSequence` array exists. Deleting a character on the terminal involves moving the cursor backward, writing a space, then moving the cursor back again. The terminal “should” automatically stop the cursor from moving back if it is already at the start of a new line.

```
/* Send the Backspace sequence to clear the character on the terminal */
Debug_au32MsgTokens[Debug_u8TokenCounter] =
    DebugPrintf(au8BackspaceSequence);
AdvanceTokenCounter();
break;
} /* end case(ASCII_BACKSPACE) */
```

A small subsystem in the Debug task runs to clean up the message tokens that are generated by the Debug function itself. This is a small circular array that will collect tokens as they are generated. At the end of the Idle task, the array is used to clear out any COMPLETED messages from the message status buffer. Although this is not required, it is good practice. `AdvanceTokenCounter()` is a simple inline function to make the system circular.

```
inline static void AdvanceTokenCounter(void)
{
    Debug_u8TokenCounter++;
    if(Debug_u8TokenCounter == DEBUG_TOKEN_ARRAY_SIZE)
    {
        Debug_u8TokenCounter = 0;
    }
}
```

The carriage return case is simpler. For passthrough mode, this doesn't have to do anything but fall through to the default case. For the Debug task, this is the character that signals a command has been entered by the user in which case the task will change states to go and check the command. The Boolean `bCommandFound` gets set to prevent any other characters from being processed at this time.



```
case(ASCII_CARRIAGE_RETURN):
{
    if( !( G_u32DebugFlags & _DEBUG_PASSTHROUGH) )
    {
        bCommandFound = TRUE;
        Debug_pfnStateMachine = DebugSM_CheckCmd;
    }

    /* Fall through to default */
} /* end case(ASCII_CARRIAGE_RETURN) */
```



The default case handles every other character. As long as the scanf buffer is not full, the character is added, and the character count is incremented. If the buffer is full, it is not added. This is a strange case and difficult to decide how to handle. Since the command buffer will report if too many characters have been entered, we chose to leave the scanf buffer as-is. The input pointer will wrap around and start over-writing characters in the buffer, but of course, the size of the buffer will not change. The character is also queued to the UART for echo back to the terminal.

```
/* Process for scanf */
if(G_u8DebugScanfCharCount < DEBUG_SCANF_BUFFER_SIZE)
{
    G_au8DebugScanfBuffer[G_u8DebugScanfCharCount] = u8CurrentByte;
    G_u8DebugScanfCharCount++;
}

/* Echo the character back to the terminal */
Debug_au32MsgTokens[Debug_u8TokenCounter] =
    UartWriteByte(Debug_Uart, u8CurrentByte);

AdvanceTokenCounter();
```



If passthrough mode is not active, then the character is also added to the command buffer. When this happens, the character needs to be checked if the buffer is full or not but also if the character is a carriage return. If the buffer is now full but the character was not a carriage return, the buffer is cleared, the character counter is cleared, and an error

message is printed.

```
/* As long as Passthrough mode is not active, update command buffer */
if( !( G_u32DebugFlags & _DEBUG_PASSTHROUGH) )
{
    *Debug_pu8CmdBufferNextChar = u8CurrentByte;
    Debug_pu8CmdBufferNextChar++;
    Debug_u16CommandSize++;

    /* Command buffer full but last character not ASCII_CARRIAGE_RETURN */
    if( (Debug_pu8CmdBufferNextChar >=
        &Debug_au8CommandBuffer[DEBUG_CMD_BUFFER_SIZE]) &&
        (u8CurrentByte != ASCII_CARRIAGE_RETURN) )
    {
        Debug_pu8CmdBufferNextChar = &Debug_au8CommandBuffer[0];
        Debug_u16CommandSize = 0;

        Debug_au32MsgTokens[Debug_u8TokenCounter] =
            DebugPrintf(au8CommandOverflow);
        AdvanceTokenCounter();
    }
}
break;
} /* end default */
```

That might seem like a lot of work to process character input. If you had doubts about how a typical scanf function could be complicated, perhaps you are a believer now! Working with ASCII is a pain, but it is critically important since it is often the basis of human interactions where subtle misbehavior or unexpected actions can quickly degrade a user experience.



The Idle state is not done yet, either! We are still in the “while” loop processing characters. There is an available LED test function built into the Debug task so that is also processed since it’s a “real time” response. After the LED test, Debug\_pu8RxBufferParser is advanced and wrapped if necessary.

```
/* If the LED test is active, toggle LEDs based on characters */
if(G_u32DebugFlags & _DEBUG_LED_TEST_ENABLE)
{
    DebugLedTestCharacter(u8CurrentByte);
}

/* In all cases, advance the RxBufferParser pointer safely */
Debug_pu8RxBufferParser++;
if(Debug_pu8RxBufferParser >= &Debug_au8RxBuffer[DEBUG_RX_BUFFER_SIZE])
{
    Debug_pu8RxBufferParser = &Debug_au8RxBuffer[0];
}

} /* end while */
```

The final operation performed by the Debug Idle state is to clear any message status flags. This is not totally necessary, and there is a chance that some message statuses will be missed depending on how busy the system is. That’s ok because the Messaging task is designed to write over stale messages. However, explicitly leaving that for another task to take care of is not a very nice thing for a task to do.



Yes, every task that uses `DebugPrintf()` is likely going to be abandoning messages. It was considered to have the Debug task manage all the message tokens from `DebugPrintf()`, but if a task really did want to track its printf message statuses there would be a conflict. Keeping this local to the Debug task also removes potential issues with interrupts if any task ISR (ill-advisedly) decided to call `DebugPrintf()`. If you remember in the Messaging task description, a hook was left in place to do garbage collection, but because the message status buffer is circular, it is not necessary.

```
/* Clear out any completed messages (Query automatically removes if complete) */
u8Counter = 0;
while ( (u8Counter < DEBUG_TOKEN_ARRAY_SIZE) &&
        (Debug_au32MsgTokens[u8Counter] != 0) )
{
    QueryMessageStatus(Debug_au32MsgTokens[u8Counter]);
    u8Counter++;
}

} /* end DebugSM_Idle() */
```

### DebugSM\_CheckCmd

The Idle state watches the input stream for a `CARRIAGE_RETURN` which is the expected keyboard input when a user thinks they want to “submit” their text to the system. At that point, Debug changes states to look at the command and verify it is valid. This is a good function to write if you need practice parsing ASCII arrays. The function structure is shown below including the comments that detail what each section of the function does. If the command is deemed valid, then the SM advances to `DebugSM_ProcessCmd`. Otherwise, it returns to Idle. Write your version now.



```
void DebugSM_CheckCmd(void)
{
    static u8 au8CommandHeader[] = "en+c";
    static u8 au8InvalidCommand[] = "\nInvalid command. Use en+c##\n\n\r";
    bool bGoodCommand = TRUE;
    u8 u8Index;
    s8 s8Temp;

    /* Verify that the command starts with en+c */

    /* On good header, read the command number */

    /* If still good command, process it. Otherwise, return to Idle */

    /* Reset the command buffer next char pointer */

} /* end DebugSM_CheckCmd() */
```

Validating ASCII strings is tedious. Standard `string.h` library functions can be used if the strings are formatted correctly and null-terminated. In this case, it is not possible to directly use `strcmp()` because the “en+c” portion of the command buffer for a valid command would not be null-terminated. Therefore, we check each character.

```
/* Verify that the command starts with en+c */
u8Index = 0;
do
{
    if(Debug_au8CommandBuffer[u8Index] != au8CommandHeader[u8Index])
    {
```

```

        bGoodCommand = FALSE;
    }

    u8Index++;
} while ( bGoodCommand && (u8Index < 4) );

```

Once the preamble is verified, checking the number requires converting the ASCII digit to its corresponding numerical value so it can be range-checked and then used to index the command.

```

/* On good header, read the command number */
if(bGoodCommand)
{
    /* Make an assumption */
    bGoodCommand = FALSE;

    /* Verify the next char is a digit */
    s8Temp = Debug_au8CommandBuffer[u8Index++] - NUMBER_ASCII_TO_DEC;

    if( (s8Temp >= 0) && (s8Temp <= 9) )
    {
        Debug_u8Command = s8Temp * 10;

        /* Verify the next char is a digit */
        s8Temp = Debug_au8CommandBuffer[u8Index++] - NUMBER_ASCII_TO_DEC;
        if( (s8Temp >= 0) && (s8Temp <= 9) )
        {
            Debug_u8Command += s8Temp;

            /* Check command number is within range and the last char is CR */
            if( (Debug_u8Command < DEBUG_COMMANDS) &&
                (Debug_au8CommandBuffer[u8Index] == ASCII_CARRIAGE_RETURN) )
            {
                bGoodCommand = TRUE;
            }
        }
    }
}
}

```

Choosing the next state is simple. We chose to print an error message to alert users about invalid commands. This is obviously helpful in the majority of applications, though if this is a means to access a secure feature, you probably don't want to give any clues.

```

/* If still good command */
if( bGoodCommand )
{
    Debug_pfnStateMachine = DebugSM_ProcessCmd;
}
/* Otherwise print an error message and return to Idle */
else
{
    DebugPrintf(au8InvalidCommand);
    Debug_pfnStateMachine = DebugSM_Idle;
}

/* Reset the command buffer next char pointer */
Debug_pu8CmdBufferNextChar = &Debug_au8CommandBuffer[0];

```



We chose to use a separate state to process the command to mitigate any timing impact that the Debug task is having on the system. There is no rush to process human-input commands, so we prioritize consuming as little time as possible in each run of the EiE system over the few extra resources it takes to add an additional state.

```
void DebugSM_ProcessCmd(void)
{
    /* Setup for return to Idle state */
    Debug_pfnStateMachine = DebugSM_Idle;

    /* Call the command function in the function array (may change next state) */
    Debug_au8Commands[Debug_u8Command].DebugFunction();
} /* end DebugSM_ProcessCmd() */
```

### DebugSM\_Error()



The last state of the Debug task is set up to process global errors. It is by no means comprehensive or complete as it currently only manages a few errors from the Debug task. However, a large embedded system could benefit from a centralized error-handler with API functions that could help other tasks standardize information about an error. Some general system action could also be added by displaying simple error codes to the extreme of resetting the system.

```
void DebugSM_Error(void)
{
    static u8 au8DebugErrorMsg[] = "\n\nDebug task error: ";

    /* Flag an error and report it (if possible) */
    G_u32DebugFlags |= _DEBUG_FLAG_ERROR;
    DebugPrintf(au8DebugErrorMsg);
    DebugPrintNumber( (u32)(Debug_u8ErrorCode) );
    DebugLineFeed();

    /* Return to Idle state */
    Debug_ul6CommandSize = 0;
    Debug_pu8CmdBufferNextChar = &Debug_au8CommandBuffer[0];
    Debug_pfnStateMachine = DebugSM_Idle;
} /* end DebugSM_Error() */
```

The possibilities for error-handling are practically endless. One of the most robust error handling systems we have designed used a centralized error code database and every new product was given a range of error codes within the database. This gave engineering, production, service, and even sales easy access to information and documentation for any error code on any device. It also helped standardize device performance in the context of errors.



In the final EiE code, there is also a “System Status” function and related application flags. This is another part of the system design to facilitate communication of what is going on. It is not detailed in this book and left to the reader to check out. Each of the system tasks now has new code in their initialization sections to print out status and set its application flag.

### 10.14 • Terminal Control Codes

Since this chapter brings in terminal access to the embedded system, we wanted to mention that standard terminal programs have many built-in features and can respond to various “escape sequences” and “control codes” to do things like move the cursor or change colors.

For example, you can write a Tic-Tac-Toe game and display it in a strategically-sized terminal window, so it looks good just by sending control codes along with the displayable characters.

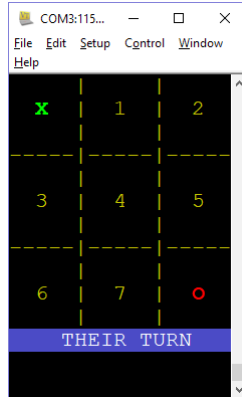


Figure 10-27 Tic-Tac-Toe game in Teraterm

Control characters can be sent in binary but can also be encoded to strings in `DebugPrintf()`. Common control codes are `\n` and `\r` for new line and carriage return, respectively. Every non-printable ASCII character has one. `\033` sends the “Esc” character (0x1B) which causes the terminal to pay attention to a few more subsequent characters. The correct sequence of characters will produce a result as long as the terminal program you use supports it. This is all part of a few terminal standards like “VT100” which sounds like a 1980s computer because it probably was.

A bunch of these sequences have been added to `utilities.h`. Tera Term supports all of these and they are what was used to make the Tic-Tac-Toe board.

```
main.c  messaging.c  utilities.c  configuration.h  sam3u_uart.c  debug.c  utilities.h x
27  /* Terminal escape sequences
28  ("033" converts to the single control character Esc (0x1B)
29  */
30  #define TERM_CLEAR_SCREEN      "\033[2J"
31  #define TERM_CUR_UP_LEFT      "\033[H"
32  #define TERM_CUR_R_C          "\033[r;cH"    /* Rows and columns
33  #define TERM_CUR_HOME         "\033[1;1H"
34  #define TERM_CUR_HIDE         "\033[?251"
35  #define TERM_DELETE_RIGHT     "\033[K"
```

Figure 10-28 Terminal escape sequences coding

Have a look and play around. The settings specified do not always work as you might expect and will sometimes conflict with the terminal’s own settings. With a little bit of effort, you should be able to get things looking the way you want. Multiple escape codes can be combined in a single string, so don’t hesitate to send a lot of configuration data out.



### 10.15 • Chapter Exercise

The online Debug exercise explores writing and reading data using the Debug API including how to properly interface a user task with it. With terminal access, there are so many things you can do now. Games like Tic-Tac-Toe, Battleship, and even Brick-Breaker or Pong can be implemented very well. There are many more practical applications, too, such as providing user interfaces, sending script files, or logging data from a running system.

Terminal access to an embedded system is really a fundamental and essential service to provide in a design, and it is very rare to find an embedded system that wouldn't have an interface (even though it might be secret).

## Chapter 11 • I SPI with my I2C

A lot of detail was provided in the last chapter to explain how the UART serial driver works in the EiE firmware system. That drastically reduces the amount of detail that needs to be included in this chapter about the main operation of an EiE communications driver. If you skipped reading the previous chapter and find something missing here, please reference back. The focus here will be to explain the SPI protocol and the peripheral implementation and then briefly cover the differences in the driver and API functions between UART and SPI. There will still be a lot of SPI-specific information.

### 11.1 • SPI Signaling

SPI is a synchronous communication system that uses a Master/Slave relationship between devices. Synchronous means that the clock signal is part of the communication signaling. SPI can be very simple and in its most basic form behaves as a simple shift register. A shift register is a sequence of bit locations where the previous bit is transferred to the next bit with every clock.

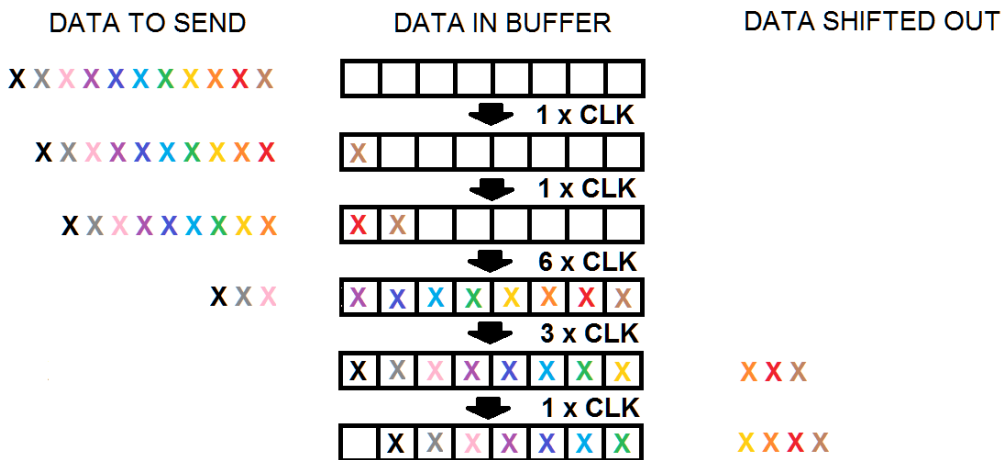


Figure 11-1 Shift register sequence

Systems like this may be more common than you might expect. For example, LED displays are made by mixing red, green, and blue light of various intensities. The LEDs are in groups of 3 (RGB) to make a color “pixel” and the whole display is a matrix of these pixels to form the image. A simple 10x10 pixel display is 100 pixels with 3 LEDs each or 300 total LEDs. The intensity of each LED is adjusted to achieve the desired color for each pixel. If each LED has 8 bits to configure how bright it is, 8 bits x 3 LEDs/pixel x 100 pixels = 2400 bits to define the whole display.

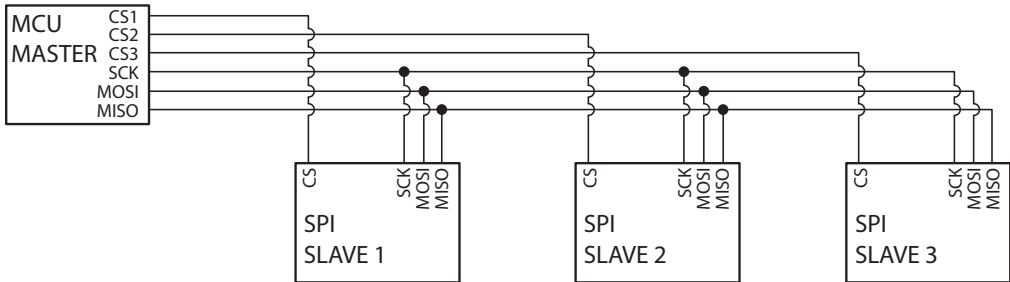
In many LED displays, the memory structure of those 2400 bits would simply be a 2400-bit long shift register. Each 8-bit location is daisy-chained to the next. The data for the last LED in the last pixel is clocked first, then each subsequent LED’s data is clocked which pushes the first data into the next location. If exactly 2400 bits are clocked, all the bits for all the LEDs end up in the correct location. Then a “latch” signal is sent which causes the new data to be applied to its respective LEDs and the image refreshes to the newly loaded values. It seems like a complicated amount of work, but for a microcontroller it’s simple.



An SPI peripheral could send the image data in 2400-bit bursts at whatever refresh rate was required for the display. Refreshing 30 times per second means that the SPI peripheral is clocking out 72,000 bits every second. The refresh time is periodic, so 30 times per second means the 2400 data bits must be updated every 33.3ms. Due to the simplicity of SPI, these systems can be clocked extremely quickly, so even the EiE system could burst out 2400 bits at 48 MHz. At this speed, it would take only 50us to clock the data and latch it, so it would be easy to be ready to update every 33.3ms. A basic driver system like this could support much larger displays, though there are limits that start to come into effect quite quickly. As an exercise, how many bits need to be updated for a 4K TV assuming 24-bit color?

For bi-directional SPI data communication, data is provided to the SPI peripheral as bytes and then the data is clocked out to the receiving device's buffer. The Master device provides the clock. Each node is part of a system bus that allows multiple device connectivity. SPI does not use any additional bits for message frame information, though two systems need to agree on the number of data bits being used. Rarely would a system use something other than 8 bits. An SPI transmission is 100% efficient at the protocol layer. Applications could optionally add on framing information if it was necessary.

Most embedded systems with SPI devices on board will have a single Master and one or more Slave devices. Usually, the microcontroller is the Master and it interfaces to other SPI Slave devices on the circuit board. The Master can address different devices on the bus using a "chip select" (CS) hardware line. The CS line might be called something else, but the functionality is always the same. Data transmission with SPI uses a 3-wire bus consisting of a Master-Out-Slave-In (MOSI) data line, a Master-In-Slave-Out (MISO) data line and a Serial Clock (SCK). The clock and data lines extend to all the devices on the bus, but each Slave has a dedicated chip select line connected to the Master. That means that the hardware overhead for SPI assuming  $n$  devices is  $3 + n$ .



**Figure 11-2 SPI Master and Slave devices**

If there is only one Master and one Slave on the bus, the CS line may be optional, though it is often still required for triggering purposes. If it is used, CS might be integrated into the SPI peripheral or managed manually by the Master. Multi-Masters are allowed, and in this case, the CS line is required. Some SPI peripherals on some processors require CS regardless, so it is a good idea to always plan for it when assigning GPIO. For simplicity's sake, only single-Master SPI systems will be considered in this chapter.

The most common SPI configurations use active low CS lines and keep data and clock lines in the high state when the system is idle. Since this is not exactly standardized, SPI peripherals on microcontrollers allow the designer to configure idle IO levels, rising or falling edge sampling, and even have a degree of customization for data lengths, address lengths and other things. The clock behavior setting is usually called "CPOL" for clock polarity, and the sampling edge is often referred to as the clock phase or "CPHA." If two systems are going to communicate properly, they both must have the same CPOL and CPHA settings. If you are writing an SPI driver and the data seems like it is getting shifted

by 1 bit, this is a good indication that the settings are not matched. Some systems will seem totally unresponsive but suddenly spring to life when CPHA is flipped to the other state. While this should be determined from the device documentation, it is often difficult to tell for sure. This is a case where a quick test might save a lot of time.

Master / Slave systems are easy to work with especially from the point of view of the Master that wants to send data. When the Master wants to send a message to a Slave, the Master asserts the Slave's CS and begins transmitting. The assumption is that the Slave is ready, but flow control could be added to the system as well. The clock signal is always provided by the Master and can practically be at any frequency, though the maximum for an SPI bus will depend on hardware limitations. Both Master and Slave are automatically synchronized because the clock is sent with the message. The clock rate can even change between messages or during the same message, though it is unlikely this would ever be done purposely. Extremely slow clocks can be used for special applications with very long data lines or high-noise environments.

The Slave SPI peripheral does not generate an SPI clock since the SCK signal is provided by the Master. Many SPI Slave peripherals will run even if their main clock in the device is off and the processor is sleeping. The incoming clock signal from the Slave takes care of moving all the data in the Slave's SPI peripheral – remember, it's just a shift register. If you add direct memory access (DMA) capability, then whole streams of data can be transferred to the Slave without ever waking up the core processor!

During a data transaction, the Slave samples data on the clock edge, which means that the processor must be sufficiently fast to detect the edge and grab the signal state of the MOSI line to capture the bit. If an SPI peripheral is used, the programmer's life is easier as this is all taken care of with an edge detector and the sample happens quickly and reliably. Really the only thing to confirm is the datasheet spec of the Master and Slave to make sure the SPI clock is not beyond the capability of the Slave.

There is no ACK or NACK at the protocol level after each byte. If mission-critical data is being sent, the application should verify data by adding an appropriately reliable method. In most cases of onboard messaging that requires perfect data, packets of bytes will feature checksum data of some sort that the receiver uses to get a go or no-go status of bytes received.

If you try to bit-bash an SPI protocol, you will have to optimize the code to ensure you make the SPI transfer a priority when it is occurring. It would be best to use an interrupt on the SCK clock edge to trigger the bit sample rather than trying to poll for the clock. A Master is easier to program than a Slave since the Master has control of the clock and therefore can take as much time as needed between bits. The Slave must be able to operate at the speed the system is rated for.

For the Master to receive data, bits are signaled on the MISO line and sampled on the clock edge. Having separate transmit and receive lines that operate simultaneously (full duplex) is a feature of SPI that makes it highly efficient. This is a good time to introduce what may be the most confusing part of an SPI transaction: both the Master and Slave always transmit, even if one side does not have meaningful data to send.

If both devices have data to send, great, each side can send its transmit data and read in the received data. If one side has nothing to send, it is common practice to just send a known "dummy" byte. This could be a byte that is unlikely to occur if that's possible given the data that is typical in the system. For example, if the data is all ASCII characters, sending a non-printable control character would be a good choice.

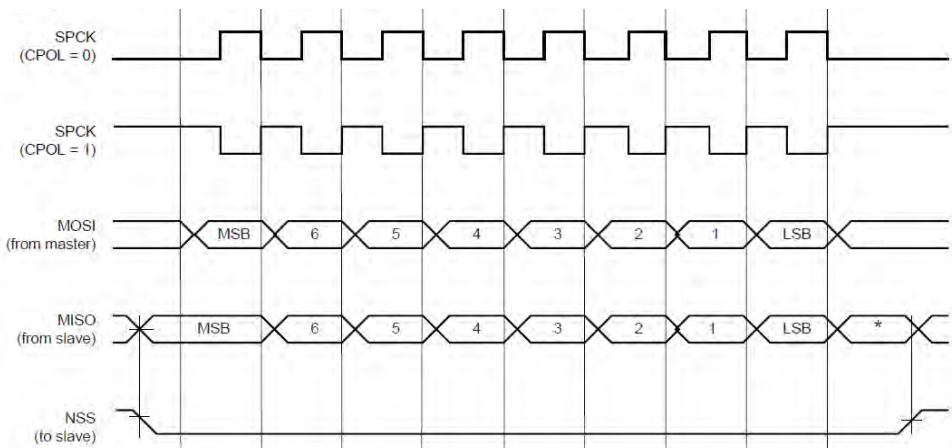


Think about various advantages and disadvantages of choosing values like 0xAA, 0x55, 0xA5, 0xFF, 0x0F, etc. to send as dummy bytes.

If an SPI device is being clocked for data transfer in one direction, but there is no data being sent in the other direction, there is no way to indicate this at the protocol level. Both systems send and receive every clock tick. If dummies are not being sent, the data line will be left in its idle state. But again, there is no way for the other side to know this by the SPI definition. If the idle state of an unused data line is low, then the other side will keep reading 0x00 for every byte. The physical data is no different than a device actively sending 0x00 as real data, or actively sending 0x00 because that is the defined dummy byte. The point is, it is up to the application to deal with this because the peripheral will just keep sending and receiving bytes. Exactly how to deal with this can be a conundrum, especially if a peripheral requires the application to write and read the SPI peripheral's transmit and receive buffers to operate correctly. Some peripherals care, others do not.

There are some SPI configurations that will sample data on both the rising and falling clock edges, which further doubles the available throughput of the signaling. Further to that, some devices are set up to use multiple SPI buses between the same devices to run parallel data streams for even faster net transfer. For the exploration here, we will focus only on single clock edge transfers over one connection pair.

The image below shows all signals including the chip select (in this case called NSS), the clock signal of both polarities, and the two data lines. The clock phase bit is 1 here. MOSI and MISO are sampled on the clock edge in the middle of the periods shown.



**Figure 11-3 SPI configuration signals**

Sometimes sending meaningful data from the Slave to the Master is a bit tricky since everything is initiated on the Master's side including the clock signal. Since only the Master can initiate communication, a Slave cannot just power up and send a message whenever it wants to. There are at least three ways around this:

1. The Slave must signal the Master in some way.
2. The Master must regularly read the Slave.
3. The Master knows when it needs information and it sends a message to the Slave that indicates the Slave should get ready to respond with certain data the next time the Master sends the clock. Sometimes the Slave can respond in the same transaction so a transmit operation changes to a receive operation without toggling CS between.

In the first case, a dedicated signaling line is the easiest way to go. You will see this in the ANT chapter as ANT defines a “message ready” (MRDY) signal that the Slave uses to alert the Master. If a dedicated GPIO was not available, the MISO line itself could be used to signal the Master. Both the Master and Slave would have to reconfigure this line from being connected to their SPI peripherals to being a standard GPIO. Once the signaling was complete, the lines could be repurposed back to their peripheral functions to complete the data transfer.

The second option is for the Master to regularly poll the Slave for data. This works well when the Slave is a sensor or a device with a large enough data buffer with room to store data while it is not being polled. Or maybe the frequency of the updates is not critical and whenever the Master gets around to polling the Slave the instantaneous data at that point is all that’s needed. If the Slave were a temperature sensor, for example, the Master might only care to know what the temperature is at an update rate of once every five seconds. It can ask the Slave for the latest temperature reading at this interval, and the Slave will report the current temperature whenever it is asked.

There exists the potential of missing information depending on the sampling rate, but that is up to the system designer to account for. In the case of temperature, it is probably physically impossible to have a multi-degree change up and then back down in a span of a few seconds, so the 5-second sampling rate would be plenty. If the Slave could accumulate data, then every five seconds the Master could poll it and either read a fixed number of bytes or agree in the firmware messaging protocol that the first byte it receives from the Slave is the number of new bytes the Slave has to send. Then the Master would provide enough clock signals to retrieve the available data.

In the third case, the application protocol is specified so the Slave knows what to expect. A common interface is for a Master to send the Slave a starting address of information that it wants to read and then start clocking. The Slave keeps sending data and sequentially increments the address. The Master knows what all the data means as it comes in based on the byte order.

SPI is capable of full-duplex transmission since data can be transferred in both directions with MOSI and MISO lines. In an embedded system, SPI data transfer still is typically implemented in a half-duplex way. This is just like the UART where both sides of the system take turns to transmit messages back and forth since the application is probably implemented with a send-message, get-response approach. Remember, though, that SPI always sends and receives at the same time. SPI Master peripherals usually do not even have means to activate their clocks just to receive data. A byte must be written into the transmit buffer to turn on the clock to receive the incoming byte from the Slave. The Slave will also receive the byte that was used to initiate the clock. Even if the Master send register did not have to be loaded with anything to trigger the clock, the Slave would still sample the MOSI line and simply read 0xff or 0x00 for the byte depending on the idle state of MOSI.

The application using the SPI interface must know what bytes have meaning and which ones are dummy bytes. Having a defined dummy byte can make this more intuitive, but in typical data, there is a 1 in 256 chance that real data is the same as the chosen dummy byte, so it cannot be used exclusively. Perhaps a protocol could be designed where both sides agree on a reserved dummy byte and agree that byte will never occur in regular data if that’s possible given the data content exchanged. These are details that must be worked out at the application level.

Obviously, it is important to know if the data being transferred means anything. Master-Slave systems typically work where a simple command protocol is implemented so that both sides understand some basic data exchange messages. As a general example, consider the scenario where a Master needs to get a bunch of data from the Slave that buffers data over a few seconds. The Master can send a request message command which

could prompt the Slave to respond with the number of bytes it has. The Master would clock a dummy byte to get this number, and then proceed to clock that number of bytes to get all the data. The send/receive message pairs that happen would go something like this (remember that both the Master and Slave send and receive a byte every cycle):

```
CHIP SELECT ASSERT

Master Tx (MOSI): Master sends byte that means, "How many bytes do you have?"
Master Rx (MISO): Garbage from Slave (unless Slave sends a defined DUMMY byte)

Master Tx (MOSI): SPI DUMMY
Master Rx (MISO): Number of bytes to send

while(number of bytes to send--)
    Master Tx (MOSI): SPI_DUMMY
    Master Rx (MISO): Slave data byte

CHIP SELECT DE-ASSERT
```

When CS is asserted, the system is in a state where the Slave knows that it is waiting for a command from the Master. The Master knows that the Slave is waiting for it to send a command. As soon as the command is sent, both the Master and the Slave know what is supposed to happen next because of the command byte. Immediately after that, both sides know how many bytes need to be exchanged. The Master sits in a loop clocking the required number of bytes. When the session is done, CS is de-asserted and the system returns to an idle state waiting for the next transaction. The significant assumption here is that there are no bit errors.

The actual information exchanged to make this happen will vary depending on the device you are talking to. If the Slave is a peripheral device of some sort, the datasheet will define the protocol used to access its information properly. There may need to be a defined delay between each byte exchange to give each side time to process the information being received from the other device. Many systems deassert and assert the CS line to change between Master-to-Slave and Slave-to-Master data exchange, though many also just do it on the fly during the same message.

11.2 • SPI Hardware

SPI clock and data lines use push-pull drivers, so they do not require pull-up resistors. This makes signaling substantially more robust over other synchronous, multi-device protocols like I<sup>2</sup>C that requires open-drain signals. Only the Master drives SCK and MOSI. Even though every SPI device on the bus connects to the MISO line, the line drivers are not active unless the chip select line is enabled to a Slave. The trade-off is that a firmware error could enable more than one SPI Slave which, if clocked, could cause shorting on the MISO line as two devices try to drive the line.

The SAM3U2 processor has one dedicated SPI peripheral, and all the USARTs are capable of SPI. There are three SPI connections on the EiE ASCII development board:

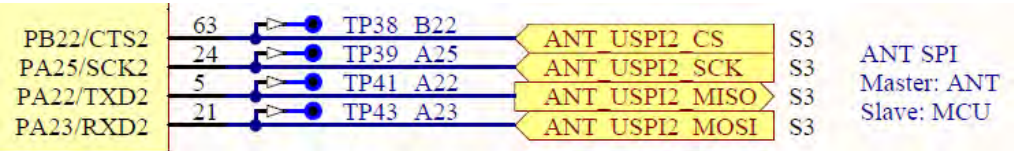


Figure 11-4 ANT (USART2)



This is an uncommon case where the main processor is the SPI Slave. The SAM3U2 and ANT processor communicate often, and both must initiate communication regularly. Choosing which should be Master and which should be Slave is a toss-up. The ANT protocol chose ANT as the Master but also defines two flow control lines to compensate. While this slightly complicates communication between the two devices, it is a good trade-off to address the requirements of the system.

The CS line (called the SEN line from ANT's perspective) is connected to the USART2 peripheral CS line, but it is configured in firmware as a standard GPIO with interrupt for better responsiveness.

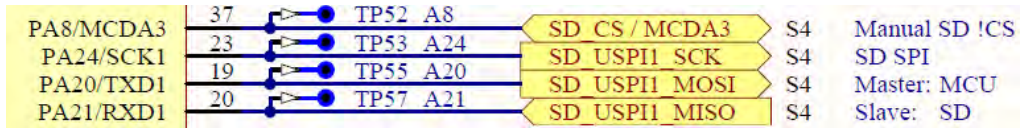


Figure 11-5 SD card (USART1)

The SD card connection should be a high-speed interface since there is potentially a lot of data being transferred. The SAM3U2 is the Master and the SD card is the Slave. The CS line will be driven by the application. There is an alternate high-speed connection to the SD card that can be used in lieu of the SPI connection to improve the read/write speed.

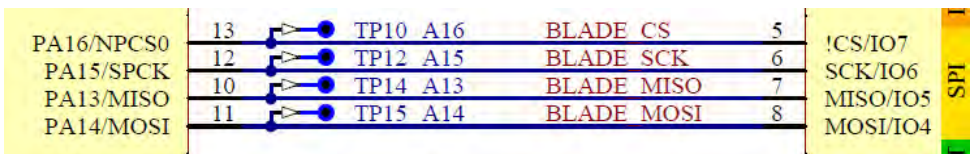


Figure 11-6 BLADE daughter board connector (dedicated SPI)

The Blade connection uses the dedicated SPI peripheral which is different enough from the USART SPI implementation that a separate driver will be written. The CS, SCK, MISO, and MOSI signaling are still identical across all the implemented SPI peripherals.

### 11.3 • SPI Registers

The driver development will focus on the more complicated case of the USART SPI implementation (denoted SSP for “Synchronous Serial Protocol”). Though Atmel does not explicitly use “SSP” in their naming convention, other vendors do.

Details of the simple SPI peripheral will also be shown where appropriate. The SD card (Master), ANT radio (Slave), and Blade (assumed Master) configurations are all included. Both SSP and SPI driver files have been added to the starting point in the project, so you can proceed to add in the INIT values discussed here.

Most of the USART peripheral information discussed in the UART chapter applies here with minor deviations. The only new base address to mention is for the dedicated SPI. There is no PDC connectivity for the simple SPI peripheral.



```
#define AT91C_BASE_SPI0 (AT91_CAST(AT91PS_SPI) 0x40008000) // (SPI0) Base Address
```

Comparing the two peripheral register sets shows similarities as expected, but there are too many differences in configuration and operation to apply the same driver code to both peripherals as was done for the UART mode. The lack of PDC connectivity is the biggest problem as our USART driver will use that for most modes of communication.



Offset	Register	Name	Offset	Register	Name
0x00	Control Register	SPI_CR	0x0000	Control Register	US_CR
0x04	Mode Register	SPI_MR	0x0004	Mode Register	US_MR
0x08	Receive Data Register	SPI_RDR	0x0008	Interrupt Enable Register	US_IER
0x0C	Transmit Data Register	SPI_TDR	0x000C	Interrupt Disable Register	US_IDR
0x10	Status Register	SPI_SR	0x0010	Interrupt Mask Register	US_IMR
0x14	Interrupt Enable Register	SPI_IER	0x0014	Channel Status Register	US_CSR
0x18	Interrupt Disable Register	SPI_IDR	0x0018	Receiver Holding Register	US_RHR
0x1C	Interrupt Mask Register	SPI_IMR	0x001C	Transmitter Holding Register	US_THR
0x20 - 0x2C	Reserved		0x0020	Baud Rate Generator Register	US_BRGR
0x30	Chip Select Register 0	SPI_CSR0	0x0024	Receiver Time-out Register	US_RTOR
0x34	Chip Select Register 1	SPI_CSR1	0x0028	Transmitter Timeguard Register	US_TTGR
0x38	Chip Select Register 2	SPI_CSR2	0x2C - 0x3C	Reserved	-
0x3C	Chip Select Register 3	SPI_CSR3	0xE4	Write Protect Mode Register	US_WPMR
0x4C - 0xE0	Reserved	-	0xE8	Write Protect Status Register	US_WPSR
0xE4	Write Protection Control Register	SPI_WPMR	0x5C - 0xFC	Reserved	-
0xE8	Write Protection Status Register	SPI_WPSR	0x100 - 0x128	Reserved for PDC Registers	-
0x00E8 - 0x00F8	Reserved	-			
0x00FC	Reserved	-			

Figure 11-7 SPI vs. USART peripheral user interface registers

Three new sections have been set up in configuration.h for all the SPI peripheral register definitions. The initialization values indicated below are shown together but would be placed in their appropriate locations. Be sure to compare the values with the bits in the registers to understand them.

**Control Register (SPI\_CR, US\_CR):** SPI\_CR uses only four bits. The peripheral can be Enabled, Disabled, and Reset. There is also a control bit for the SPI CS line. For US\_CR, the same reset, enable and disable bits for the transmitter and receiver are used like in the UART case. Remaining bits will not be used.



```
#define BLADE_SPI_CR_INIT (u32)0x00000002
#define SD_SPI_US_CR_INIT (u32)0x00000050
#define ANT_SPI_US_CR_INIT (u32)0x00000050
```

**USART Mode Register (SPI\_MR, US\_MR):** The SPI Mode register offers a single bit to select Master or Slave mode. Other bits provide interesting options that allow the SPI peripheral to interface efficiently to multiple Slave devices, though that is not of interest here.



```
#define BLADE_SPI_MR_INIT (u32)0x00000021
```

For US\_MR, all the SPI-specific bit selections are required to properly configure the USART. Go through the value to verify you understand the configurations. Remember that the SAM3U2 will be the Master for the SD application, but is the Slave for the ANT interface.



```
#define SD_SPI_US_MR_INIT (u32)0x004518CE
#define ANT_SPI_US_MR_INIT (u32)0x004118FF
```

**Receive Data Register, Receive Holding Register (SPI\_RDR, US\_RHR):** These registers have the same main function of receiving incoming data for both the SPI and USART peripherals. The SPI version can technically support 16-bit transfers although anything other than 8 bits is rare. It has 4 bits that capture the current chip select mode to further

support multiple Slaves. The RXSYNH bit in US\_RHR does not apply to basic SPI.

**Transmit Data Register, Transmit Holding Register (SPI\_TDR, US\_THR):** Both the transmit registers are written with the value to clock out during the next transfer. The additional bits match the receive versions, except SPI\_TDR can automatically de-assert the chip select.

**Interrupt Enable / Disable / Mask Registers (SPI\_IER / SPI\_IDR / SPI\_IMR, US\_IER / US\_IDR / US\_IMR):** Both the SPI and USART interrupt registers have very similar interrupt sources for main functions like transmit, receive, and errors. US\_IER has additional flags for more features and the PDC-related interrupts.

The interrupt start-up values for communication signals should be off because the SPI drivers will use the same Request/Release mechanism as the UART driver. The CS interrupt will be enabled for the ANT Slave SPI to start catching inputs from the nRF51422 ANT processor.



```
#define BLADE_SPI_US_IER_INIT (u32)0x00000000
#define SD_SPI_US_IER_INIT (u32)0x00000000
#define ANT_SPI_US_IER_INIT (u32)0x00080000
```

**SPI Chip Select Registers (SPI\_CSR0...3):** These registers apply only to the dedicated SPI peripheral. They hold the critical setup data including clock polarity, phase selection, and SPI clock speed. They also allow programming a delay between bytes that are sent and control the behavior of the CS line after transmission. From this, we see that the dedicated SPI peripheral is feature-rich when compared to the USART SPI functionality. It can definitely be optimized to achieve a very robust and high-speed connection to another device.

There are four CSR registers because the SPI peripheral can interface to four SPI Slaves. There is a note that indicates that all four must be written even if defaults are desired, but it is not clear if single-Slave mode requires this.

For initialization, set both CPOL and NCPHA. The CS behavior is complicated when described in words but you can reference back to the user guide section to get further explanation. The flow chart for Master Transmission provides the clearest picture of what bits should be set to make transmission go as expected. This is obviously different than the USART data flow, which reinforces the need for a separate driver. See Figure 11-8 on page 372

Since the Blade only supports a single SPI Slave directly on the peripheral, choose the “Fixed peripheral” mode and keep PCS as 0 which means the Blade’s CS signal will be used. The words in the user guide that we found confusing (or missing) were how CS is asserted and deasserted. CS is managed automatically by the peripheral. When TDRE is written, the SPI peripheral checks CSAAT which, when 0, tells the SPI peripheral to raise CS after transmission of a byte. This does not seem correct and implies that CS will rise after every byte. However, if you follow through the flowchart, you should see that as long as TDRE is not starved then the transfer will continue without de-asserting CS. The flowchart gives a good view of where the various delays are inserted as well. Though it may seem initially confusing, it is brilliantly simple.

To achieve this behavior, both CSAAT and CSNAAT are 0. BITS is cleared for 8-bit data transfer. Set the SPI baud rate to 1 MHz by loading SCBR with 48 (0x30).

$$SPCK \text{ Baudrate} = \frac{MCK}{SCBR} \Rightarrow SCBR = \frac{48MHz}{1MHz} = 48$$

DLBYS is a delay between the falling edge of CS and the start of the SPI clock. Let's choose 1us so DLYBS is 48 (0x30).

$$\text{Delay Before SPCK} = \frac{DLYBS}{MCK} \Rightarrow DLYBS = SPCK \times MCK = 1\mu s \times 48\text{MHz} = 48$$

Lastly, the delay between consecutive bytes sent is a function of DLYBCT. This can have a dramatic impact on the overall speed of transfer. For now, we'll target 3 SPI clock ticks (3us) and adjust if required. Since fractional values are not allowed, round up to set DLYBCT = 5.

$$\text{Delay Between Xfers} = \frac{32 \times DLYBCT}{MCK} \Rightarrow DLYBCT = \frac{3\mu s \times 48\text{MHz}}{32} = 4.5$$

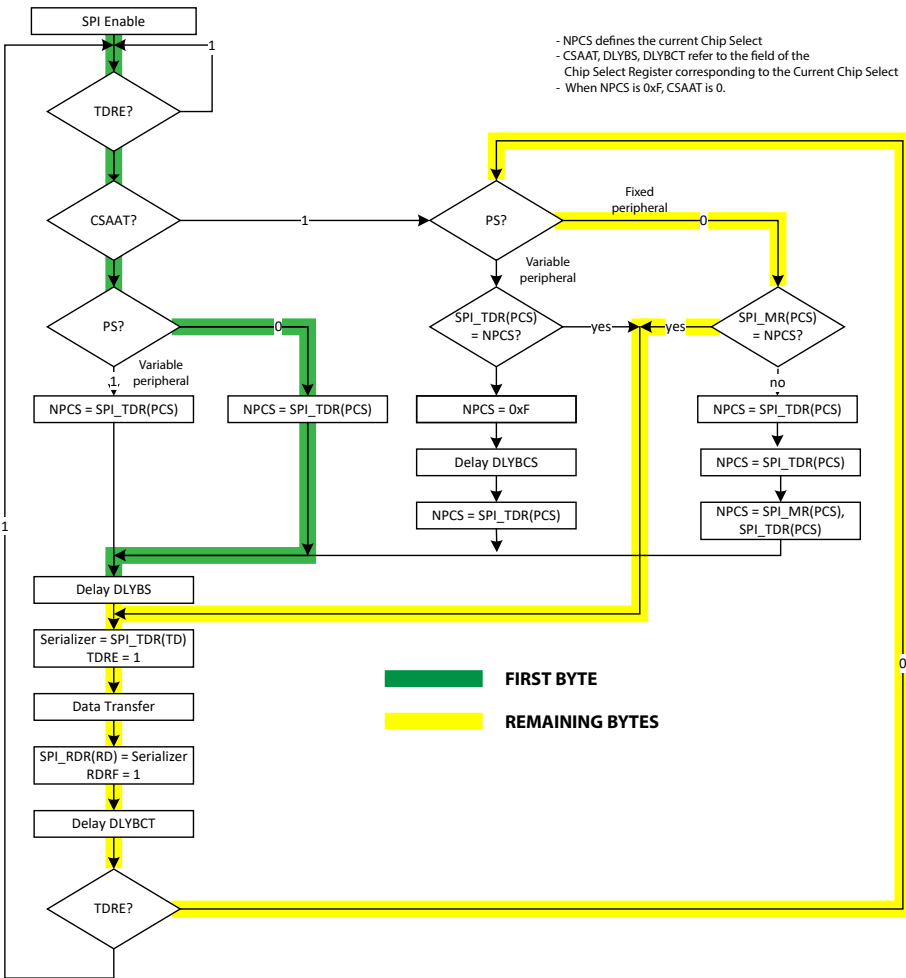


Figure 11-8 USART flowchart

Capture all these calculations in the code and be sure to comment them as you define the setup value for SPI\_CSR.



```
#define BLADE_SPI_CSR0_INIT (u32)0x05303001
#define BLADE_SPI_CSR1_INIT (u32)0x05303001
#define BLADE_SPI_CSR2_INIT (u32)0x05303001
#define BLADE_SPI_CSR3_INIT (u32)0x05303001
```

**USART Baud Rate Generator (US\_BRGR):** We examined this already in the UART chapter, though for SPI only the CD bits matter. The baud rate is  $MCK / CD$ . For the SD card, use 4 MHz as we know that to be a reliable speed on the EIE development board from verifying the rise and fall times using an oscilloscope. The value does not matter for the ANT Slave and can be set to 0 to disable the generator. The user guide notes that the incoming clock cannot exceed  $MCK / 6$  ( $48\text{MHz} / 6 = 8\text{MHz}$ ).



```
#define SD_SPI_US_BRGR_INIT (u32)0x00000030
#define ANT_SPI_US_BRGR_INIT (u32)0x00000000
```

**SPI Status Register, USART Channel Status Register (SPI\_SR, US\_CSR):** The behavior of US\_CSR in SPI mode is nearly identical to UART mode. The same bits will be used for the driver. SPI\_SR is essentially a subset of US\_CSR with the main difference being the lack of PDC-related status bits. Status registers do not require initialization.

**Write Protection Mode / Status Registers (SPI\_WPMR / SPI\_WPSR, US\_WPMR, US\_WPSR):** As with many other peripherals, these registers allow write protection to critical registers in the peripherals. Protection is left disabled for EIE.

#### 11.4 • EIE SPI Driver

The high-level view of the USART SPI driver is nearly identical to the UART driver. Interrupts, the peripheral DMA controller, and the Message task are all integral parts. The API to the user hides most of the work being done by the driver code. Though we are again describing a system that we have already built, the initial design process was very similar but with a lot more decisions requiring some testing and fitting into the final solution. If you compare the dedicated SPI driver to the USART SPI driver, you will also see many similarities but overall the SPI driver is substantially simpler.

A difficult part of designing a synchronous mode communication driver is making it work for both Master and Slave cases. Very often vendor-provided code will give separate source files for Master and Slave drivers. Our driver will support both. To further complicate the design, the driver will also support Slave devices that use flow control so that the driver will interface to ANT without further modifications. Therefore, there are three modes that must be handled, each with transmit and receive cases. This provides a great opportunity to demonstrate a lot of different considerations that must be made. The table below summarizes the main design decisions made for each scenario.

SPI Mode Behavior		
	Transmit	Receive
MASTER	<ul style="list-style-type: none"> <li>- Start in SspSM_Idle</li> <li>- DMA all bytes</li> <li>- Rx dummies to application</li> </ul>	<ul style="list-style-type: none"> <li>- Start in SspSM_Idle</li> <li>- DMA all bytes</li> <li>- Tx dummies from Rx buffer</li> </ul>
SLAVE	<ul style="list-style-type: none"> <li>- Start in SspSM_Idle</li> <li>- DMA all bytes</li> <li>- Rx dummies to application</li> </ul>	<ul style="list-style-type: none"> <li>- Rx Interrupt driven</li> <li>- DMA byte-wise</li> <li>- Tx dummies from SSP_u8Dummies</li> </ul>
SLAVE FLOW CONTROL	<ul style="list-style-type: none"> <li>- Start in application (CS is asserted, app queues Tx message but does not complete flow control)</li> <li>- Peripheral byte-wise</li> <li>- Rx dummies not received</li> </ul>	<ul style="list-style-type: none"> <li>- Rx Interrupt Driven</li> <li>- Peripheral byte-wise</li> <li>- application defines dummy behaviour</li> </ul>

Figure 11-9 SPI Mode Behaviour

All three transmit cases follow the UART driver design closely. For receive, the Slave cases are also similar to the UART receive cases and are essentially all interrupt-driven. A big difference comes from the Master Receive that will still be handled initially by the SPI driver since the Master must initiate data transfer in a synchronous system.

Understanding data flow for each of the three supported SPI modes is fundamental to writing the driver. Remember that all communication is considered half-duplex by the EiE system. Note that the Master mode is further divided into a “manual CS” mode and “auto CS” mode in the code but this does not impact the behavior of transmit and receive functions.

The flowchart below describes how the SSP Driver will work. Numerous decisions had to be made in the design to ensure that abstraction was maintained, the API made sense, and that the whole thing worked. There is plenty of room for debate and improvement, particularly in the amount of onus placed on the application to successfully use the driver. In a simple system, usability issues are unlikely. In a busy system that requires multi-task access to the same peripheral, the client applications might get stuck with a lot of denials for requesting the peripheral and reading or writing data.

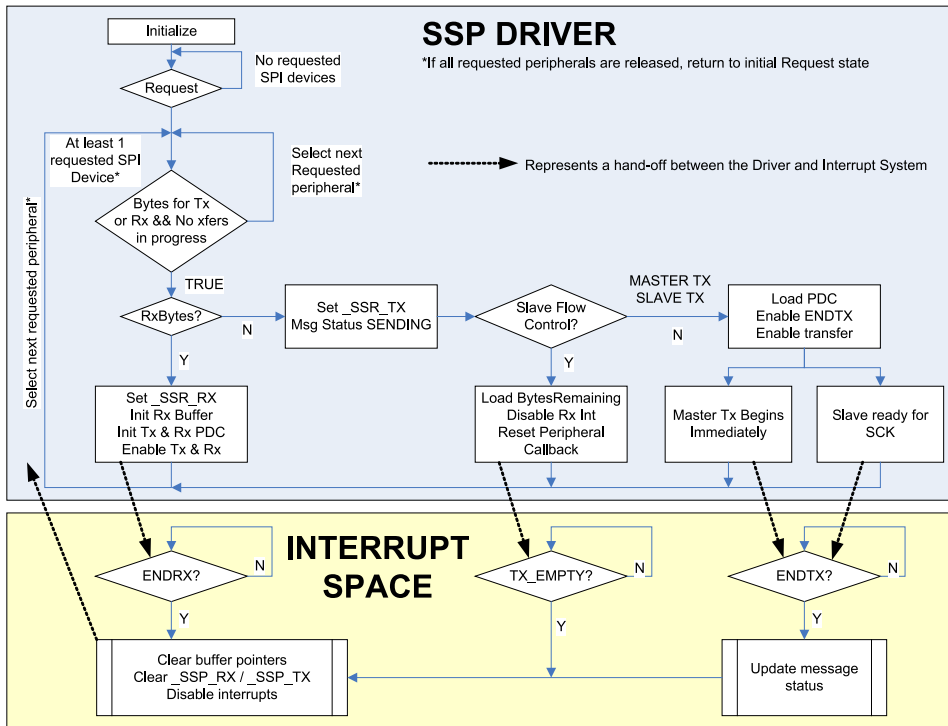


Figure 11-10 SSP Driver flowchart

Data reception on Master devices presents a problem. The driver code is written such that as soon as a task calls a Read API function, the RxBuffer pointer is set to the task's read buffer. The Idle state machine logic is conditioned on this pointer being NULL or pointing at a RxBuffer to determine if the Master is transmitting or receiving. Therefore, as soon as a request to read data is made, the driver will not properly send any queued transmit messages. The Rx message gets priority, which likely does not make sense in a sequence of write and read operations to a Slave.

Admittedly this is a significant restriction that resulted from the logic used by the Idle state, though it does demonstrate the benefits of using linked lists to queue up messages. How could this part of the design be improved? The first solution that came to mind was to have an additional queue with receive message information including the system time when the Rx function was requested. This could prevent setting the RxBuffer pointer while there are older messages to transmit since the Message task tags each queued message with the system time as well.

In real applications, the scenario is not likely to occur, but the system should still be protected from running into the condition. To solve this, we prevent the system from allowing both a transmit and receive message to be queued at the same time on the same peripheral. This control will be put in place in the Read and Write API functions for the driver. If a message is already present, the application will be denied access and will have to wait until the current message transfer is complete. From the state machine's perspective, the "in progress" flags are used to ensure a redundant operation is not started when a transfer is already occurring. Offloading this to the client task is lazy, but not the end of the world for the EiE application.



Beyond the data reception logic problem, data transfer in the SSP driver works well. Using the table and flowchart above, trace through the design and imagine the data flow that is taking place. Consider the application, the driver, and the interrupt system. Do you see any problems or opportunities for improvement? Make sure you consider all the consequences of any change you think could be made. In the following sections, each function is described – compare your understanding with these explanations.

### 11.5 • Master Transmit

Master devices are easy because they have total control of the system. Imagine that you have  $n$  bytes of data to send and assume that these are in sequential RAM locations on the Master device. All that is required is to set the DMA pointer to the data, tell the PDC how many bytes are waiting, and initiate the transfer. Dummy bytes being clocked in will still be supplied to the application which allows full-duplex communication if desired. In most cases, the application can just discard the dummies.

### 11.6 • Master Receive

A receiving Master SPI still has all the power in the system. Based on experience with dozens of different SPI Slave devices, the Master will typically know how many bytes the Slave has to send. Since the value is known, DMA can again be used easily to capture the incoming data as the Master clocks the system. Since a Slave cannot send unsolicited data, a receive operation by the Master is executed by a transmit-style operation.

This is a fundamental difference between the SSP vs. the UART drivers which both use the USART peripheral. The receive function on the UART driver worked exclusively on interrupts as bytes appeared in the Rx Buffer. For the SPI driver, reception is done very similarly to a transmit and will be initiated in the Idle driver loop. A transmit operation is set up with every receive operation.

We considered queuing a message to the Message task that was simply an array of dummy bytes. This would allow the built-in message status functions to be used. The downside is that this would require additional system resources and there is a slight problem with exact synchronization between transmit and receive. This extends from the tendency of SPI peripherals to signal that a transmit is complete slightly before the receive is complete.

Instead, we initialize the receive buffer to dummy bytes and will use this as the source for the Master's dummy transmissions. This is a "just in time" operation where the current

dummy is read from the receive buffer and placed in the transmit buffer which starts the SPI transaction. The received byte will be placed in the same location in the Rx buffer which is fine since the dummy byte that was there is no longer needed. An initialized receive buffer is nice as it is much more obvious when new data has arrived, and we don't have to use any extra resources for the transmit dummies or change how the sending algorithm works.

### 11.7 • Slave Transmit

A Slave transmit function without flow control will work almost identically to a Master transmit. The system must have some way to know how much data the Master will request and where that data is, so it can be ready when the Master clocks the system. This must be worked out at the application level, and the Slave would provide some timing specifications to ensure that it is always ready for the Master clock.

The application should be able to queue a transmit operation with the same API calls as if the system were a Master. The process for getting ready to send those bytes is identical to the Master transmit flow. The only difference is that once the transfer is ready, the Slave can do nothing but wait for the Master to start clocking.

Just like the Master, the Slave transmits using the PDC and terminates when the ENDTX interrupt fires. A question that should arise is what happens if the Master does not clock all the bytes that the Slave was set up to send? Though this shouldn't happen since it would indicate a faulty system, production code still should recover gracefully. A timeout could be set up with error reporting and clean-up. For the EiE system, we choose to let the operation remain active unless the CS line was de-asserted. In this case, the system will properly clean up by setting the message to ABANDONED and resetting the PDC registers and associated flags.

### 11.8 • Slave Receive

Receiving on a Slave device is done almost identically to the UART receive operation. The PDC will be set up to operate byte-wise using both the current and next PDC counter/pointer registers. These pointers will leap-frog each other as bytes arrive, and wrap-around the circular receive buffer of the application. The driver makes this all happen automatically so the application does not have to consider it. The application that has the SPI peripheral requested should inspect and manage the received data as it comes in. Presumably, the receive buffer provided by the application is large enough to accommodate the highest possible data rates while still giving enough time to process the data.

Keep in mind that 6 MHz SPI can theoretically dump 750 bytes of data in 1ms. If you are working with a system that needs this kind of speed, the EiE driver as we have written it is not going to work very well. Likely a dual-buffer implementation with strict rules on usage would be required. In this case, a separate, dedicated, high-speed SPI driver might be written with an optimized implementation and API.

### 11.9 • Slave Transmit with Flow Control

Flow control provides an interesting challenge to an interrupt-driven system where maintaining task abstraction is a priority. In cases like this, callback functions are the de facto standard to make the connection between two otherwise isolated pieces of code. Callback functions are registered as part of `SPI_Request()`.

With flow control, the Slave has enough control to ensure that it can properly set up for an expected transmit to the Master. As shown in the functional table above, Slave



transmit with flow control starts at the application level at the point where the Master has asserted CS. The Slave can take its time to call a data write API function and let the SSP driver get everything ready. The system is set up to execute the application's callback as a final setup step to initiate the transfer. The assumption is that CS from the Master is asserted while all that takes place, so the Master will likely start clocking immediately.

We chose to use the direct peripheral registers since transmit must be byte-by-byte to allow the flow control callback to run after each transfer. DMA would arguably be slightly more efficient depending on how the address registers were updated. This is also the only opportunity to change between LSB or MSB first data transfer for USART-based SPI since the USART peripheral cannot do this automatically. If DMA was used, the data would not be accessible before the PDC drops the byte into US\_THR.

The design has assumed that per-byte flow control is required which is what the ANT device needs. Some devices may only require initial preamble and perhaps some clean-up after full transmission is complete. The EiE system could easily be adapted accordingly based on the callback function, though data transfer would be far less effective due to continual interrupts.

#### 11.10 • Slave Receive with Flow Control

Receiving data on a Slave where the system uses flow control is easy with the EiE system if the peak 1ms latency from when the Master asserts CS to when the application gets ready and responds to the Master is ok. If necessary, this could be sped up with an interrupt but that adds difficulty to the abstraction.

As in the Slave Transmit flow control case, the application has time to get ready. From that point, reception is technically no different than the non-flow control case where it is up to the application to handle the incoming data to its receive buffer. Again, byte-wise transfers are necessary, so a callback function can be used to manage the flow control requirements. Ideally, the callback is fast and only manages the flow control lines. If there will be too much data for the buffer to handle, or if there are other special requirements, the callback function could take care of this. Ideally, the data is handled by the application at the regular 1ms loop interval.

#### 11.11 • Chip Select

An added complication to SPI is control of the Chip Select line. The first time you use SPI it may be assumed that CS is just the start and stop of a message frame. Technically that's correct, but it might also have more control function than just that. We have also seen timing requirements for CS vary where some devices expect CS to deassert during the last byte transmission instead of after it. If a Master does this, a Slave that doesn't follow the rule may interrupt the last byte transfer since some Slave SPI peripherals will shut down immediately if CS is not asserted. Though most SPI peripherals can manage the CS line directly, there are usually overrides to allow a driver to manage CS manually. If not, a standard GPIO line can be used.

Chip select can get more complicated on Slave devices especially when flow control is involved and if the application needs to be aware of it. This makes it more difficult to abstract the peripheral function from the application layer as an application should not have direct access to an SPI resource. There may be additional timing implications here, too, depending on how the hardware and firmware work. In the EiE 1ms system design, high-speed hardware CS changes can easily be missed. If that's a problem, a latching mechanism must be implemented that the application can acknowledge during the regular loop time. However, if the CS line has changed states, it probably does not make sense to react to a previous state.



The CS hardware signal for Slave devices triggers an interrupt and then sets an application flag for the SPI peripheral on which it occurred. This flag is not latched and will deassert in real-time with the actual CS signal. Using a software flag eliminates the need for the application to know anything about the pin location or peripheral operation.

Master devices are set up to default to manual control of the CS signal. This can be managed by the SPI task in a standard way or controlled by the application for special cases. The hardware configuration of the CS line for each SPI peripheral on the development board won't change with the exception of the Blade SPI which could connect to either a Master or Slave. The EiE code defaults to Master mode. Master devices leave access of CS to the PIO controller. Slave devices will have to use the CS peripheral function so the interrupt triggers properly. The driver would need some refactoring if PIO interrupts were used, instead.

For the dedicated SPI driver, we will leave CS control entirely up to the peripheral. Only basic Master and Slave modes will be offered.

### 11.12 • SPI Data Structures



Since configuring a USART for UART and SPI are very similar, the corresponding data structures to hold all the configuration information are also very similar. SPI is more complicated, so there happens to be double the number of members. Pay attention to the fields as you enter them in the code and make sure you relate them back to the design description so far.

```

/*!
@struct SspConfigurationType
@brief User-defined SSP configuration information
*/
typedef struct
{
    PeripheralType SspPeripheral;           /* Easy name of peripheral */
    AT91PS_PIO pCsGpioAddress;             /* Base address for GPIO port for CS */
    u32 u32CsPin;                          /* Pin location for SSEL line */
    SspBitOrderType eBitOrder;             /* SPI_Slave_FLOW_CONTROL only */
    SspModeType eSspMode;                 /* Type of SPI configured */
    fnCode_type fnSlaveTxFlowCallback;     /* For SPI_Slave_FLOW_CONTROL transmit */
    fnCode_type fnSlaveRxFlowCallback;     /* For SPI_Slave_FLOW_CONTROL receive */
    u8* pu8RxBufferAddress;               /* Address to circular receive buffer */
    u8** ppu8RxNextByte;                 /* Next byte for SPI_Slave_FLOW_CONTROL */
    u16 u16RxBufferSize;                  /* Size of receive buffer in bytes */
    u16 u16Pad;                           /* Preserve 4-byte alignment */
} SspConfigurationType;

```

The peripheral information that will be attached to any requested SPI peripheral also follows the same design as the UART. In the SPI case, four private flags are used.



```

/*!
@struct SspPeripheralType
@brief Full definition of SSP peripheral
*/
typedef struct
{
    AT91PS_USART pBaseAddress;             /* Base address of peripheral */
    AT91PS_PIO pCsGpioAddress;             /* Base GPIO port for chip select line */
    u32 u32CsPin;                          /* Pin location for SSEL line */
    SspBitOrderType eBitOrder;             /* SPI_Slave_FLOW_CONTROL mode */
    SspModeType eSspMode;                 /* Type of SPI configured */
    u32 u32PrivateFlags;                   /* Private peripheral flags */
}

```

```

fnCode_type fnSlaveTxFlowCallback; /* For SPI_Slave_FLOW_CONTROL transmit */
fnCode_type fnSlaveRxFlowCallback; /* For SPI_Slave_FLOW_CONTROL receive */
u8* pu8RxBuffer; /* Receive buffer in user application */
u8** ppu8RxNextByte; /* Next Rx byte (SPI_Slave_FLOW_CONTROL) */
u16 u16RxBufferSize; /* Size of receive buffer in bytes */
u16 u16RxBytes; /* Number of bytes to receive (DMA transfers) */
u8 u8PeripheralId; /* Simple peripheral ID number */
u8 u8Pad; /* Preserve 4-byte alignment */
u16 u16Pad; /* Preserve 4-byte alignment */
MessageType* psTransmitBuffer; /* Transmit message struct linked list */
u32 u32CurrentTxBytesRemaining; /* Bytes remaining in current transfer */
u8* pu8CurrentTxData; /* Current location in Tx buffer */
} SspPeripheralType;

```

```

/* u32PrivateFlags in SspPeripheralType */
#define _SSP_PERIPHERAL_ASSIGNED (u32)0x00100000
#define _SSP_PERIPHERAL_TX (u32)0x00200000
#define _SSP_PERIPHERAL_RX (u32)0x00400000
#define _SSP_PERIPHERAL_RX_COMPLETE (u32)0x00800000
/* end u32PrivateFlags */

```

The private flags for the peripheral are used exclusively by the SPI driver and not intended for any application to access. To communicate information about the operation of the SPI peripherals to a task, global variables are created for each SPI peripheral.

```

/*****
Global variable definitions with scope across entire project.
All Global variable names shall start with "G_<type>Ssp"
*****/
/* New variables */
u32 G_u32Ssp0ApplicationFlags; /* Status flags for application */
u32 G_u32Ssp1ApplicationFlags; /* Status flags for application */
u32 G_u32Ssp2ApplicationFlags; /* Status flags for application */

```



The corresponding flags for these registers are in the driver header file.

```

/* G_u32SspxApplicationFlags */
#define _SSP_CS_ASSERTED (u32)0x00000001
#define _SSP_TX_COMPLETE (u32)0x00000002
#define _SSP_RX_COMPLETE (u32)0x00000004
/* end G_u32SspxApplicationFlags */

```



The application can use these flags in any way and does not have to have the peripheral requested which might be useful although not currently used in any task that we have written. In fact, these flags have only been used by the ANT task that uses SPI Slave. Having the information available makes sense for the driver.



We also define several enums to give self-documenting names to the various configuration parameters for the SPI devices. These appear separately for both the USART SSP and simple SPI drivers. Much debate was held over making these common to both drivers or keeping them unique with the latter option being settled on for several reasons.

```

/*!
@enum SspBitOrderType
@brief Controlled list to specify data transfer bit order.
*/

```

```
typedef enum {SSP_MSB_FIRST, SSP_LSB_FIRST} SspBitOrderType;

/*!
@enum SspModeType
@brief Controlled list of SSP modes.
*/
typedef enum {SSP_MASTER_AUTO_CS, SSP_MASTER_MANUAL_CS, SSP_SLAVE,
SSP_SLAVE_FLOW_CONTROL} SspModeType;

/*!
@enum SspRxStatusType
@brief Controlled list of SSP peripheral Rx status.
*/
typedef enum {SSP_RX_EMPTY = 0, SSP_RX_WAITING, SSP_RX_RECEIVING, SSP_RX_COMPLETE,
SSP_RX_TIMEOUT, SSP_RX_INVALID} SspRxStatusType;
```

Lastly, we need some local global variables to track flags and objects in the driver. Each peripheral has its own object pointer and the driver state machine will cycle through all peripherals available.

```
static fnCode_type Ssp_pfnStateMachine;          /* SSP state machine */

static u32 SSP_u32Timer;                         /* Timeout used across states */
static u32 SSP_u32Flags;                        /* Application flags for SSP */

static SspPeripheralType SSP_Peripheral0;        /* SSP0 peripheral object */
static SspPeripheralType SSP_Peripheral1;        /* SSP1 peripheral object */
static SspPeripheralType SSP_Peripheral2;        /* SSP2 peripheral object */

static SspPeripheralType* SSP_psCurrentSsp;      /* Current SSP in task */
static SspPeripheralType* SSP_psCurrentISR;      /* Current SSP in ISR */
static u32* SSP_pu32SspApplicationFlagsISR;     /* Current SSP status ISR */
```

### 11.13 • SPI Driver Functions in Common with UART

Some of the functions that make up the SPI driver are very similar to the UART functions already explained. A summary is provided here for functions that are essentially the same. If you are writing your own code base, this is an excellent opportunity to write the whole driver yourself and then compare the solution in the end. Generally, your code should be quite like what is shown. The descriptions below can be used as hints.

**void SspInitialize(void):** Sets all the SSP\_Peripheralx members whose values need to be initialized. The BaseAddress and PeripheralID members must be set correctly for each peripheral. All address pointers should be NULL. All global and local global flags are zeroed, and the Idle state is selected. At the end of this function, the SSP driver is ready for tasks to start requesting peripherals.

**void SspRunActiveState(void):** Calls the current active state. This function is identical to all other RunActiveState functions.

**void SspManualMode(void):** Like the UART function, this forces execution of the SSP driver during initialization. The \_SSP\_MANUAL\_MODE flag needs to be active, and the starting SSP should be reset to SSP\_Peripheral0. The SSP and Messaging tasks should be run.

**void SSP0\_IRQHandler(void), void SSP1\_IRQHandler(void), void SSP2\_IRQHandler(void):** Interrupt handlers that will trigger for specific peripherals. The code loads two variables for the generic part of the ISR, increments a debug counter, and transitions to the generic handler.

```

SSP_psCurrentISR = &SSP_Peripheral0;
SSP_pu32SspApplicationFlagsISR = &G_u32Ssp0ApplicationFlags;
SSP_u32Int0Count++;
SspGenericHandler();

```

**u32 SspWriteByte(SspPeripheralType\* psSspPeripheral\_, u8 u8Byte\_):** Queues a single-byte message to the specified peripheral and returns the message token. Driver execution is forced to Manual Mode if this is called during initialization.

**u32 SspWriteData(SspPeripheralType\* psSspPeripheral\_, u32 u32Size\_, u8\* pu8Data\_):** Queues a multi-byte message to the specified peripheral and returns the message token. Driver execution is forced to Manual Mode if this is called during initialization.

#### 11.14 • New SPI Driver Functions

The rest of the SSP driver functions differ at least in some extent from the UART-based functions and so we will look more closely at them.

##### 11.14.1 • SspRequest()

Requesting an SSP resource begins by loading INIT values to the peripheral registers that require configuration. The configuration.h definitions are critical here as the logical names associated with the standard USART register INIT values are mapped in. If the user does not use the correct peripheral name (or it is not mapped correctly) the driver will not function properly at the physical layer.

```

switch(psSspConfig_>SspPeripheral)
{
    case USART0:
    {
        psRequestedSsp = &SSP_Peripheral0;

        u32TargetCR    = USART0_US_CR_INIT;
        u32TargetMR    = USART0_US_MR_INIT;
        u32TargetIER   = USART0_US_IER_INIT;
        u32TargetIDR   = USART0_US_IDR_INIT;
        u32TargetBRGR  = USART0_US_BRGR_INIT;
        break;
    }
    <...repeated for all USARTs...>

```

The user-defined configuration values are all loaded to the SSP data structure and the binary semaphore is taken so another task cannot request the peripheral until it is released (code not shown).

For SSP\_SLAVE configurations, the PDC must be configured so reception is ready and the corresponding transmission will be the defined dummy byte. Slaves use “leap-frogging” single byte DMA transfers that start at the beginning of the RxBuffer provided. The transmitter and receiver must be turned on, and the Chip Select and ENDRX/TX interrupts should be enabled.

```

/* Special considerations for SPI Slaves */
if(psRequestedSsp->eSspMode == SSP_SLAVE)
{
    /* Preset the PDC receive pointers and counters */
    psRequestedSsp->pBaseAddress->US_RPR = (u32)psSspConfig_>pu8RxBufferAddress;

```

```

psRequestedSsp->pBaseAddress->US_RNPR =
    (u32)(psSspConfig->pu8RxBufferAddress + 1);
psRequestedSsp->pBaseAddress->US_RCR = 1;
psRequestedSsp->pBaseAddress->US_RNCR = 1;
psRequestedSsp->ppu8RxNextByte = NULL; /* not used for SSP_Slave */

/* Preset the PDC transmit registers to return predictable SPI dummy bytes
if the Slave is receiving. These will be changed if the Slave transmit is queued
by the application. */
psRequestedSsp->pBaseAddress->US_TPR = (u32)&SSP_u8Dummies;
psRequestedSsp->pBaseAddress->US_TNPR = (u32)&SSP_u8Dummies;
psRequestedSsp->pBaseAddress->US_TCR = 1;
psRequestedSsp->pBaseAddress->US_TNCR = 1;

/* Enable the receiver and transmitter so they are ready to go if the Master
starts clocking */
psRequestedSsp->pBaseAddress->US_PTCR = (AT91C_PDC_RXTEN | AT91C_PDC_TXTEN);
psRequestedSsp->pBaseAddress->US_IER =
    (AT91C_US_CTSIC | AT91C_US_ENDRX | AT91C_US_ENDTX);
}

```

Flow control Slaves need only to start with their Chip Select interrupts enabled. For all configurations, the SSP interrupt is enabled in the NVIC.

```

/* Special considerations for SPI Slaves with Flow Control */
if(psRequestedSsp->eSspMode == SSP_Slave_FLOW_CONTROL)
{
    /* Enable the CS interrupt */
    psRequestedSsp->pBaseAddress->US_IER = AT91C_US_CTSIC;
}

/* Enable SSP interrupts */
NVIC_ClearPendingIRQ( (IRQn_Type)psRequestedSsp->u8PeripheralId );
NVIC_EnableIRQ( (IRQn_Type)psRequestedSsp->u8PeripheralId );

return(psRequestedSsp);
} /* end SspRequest() */

```

#### 11.14.2 • SspRelease()

Releasing the SPI resource should disable the associated NVIC interrupts, clear the private flags and set the peripheral pointers to NULL. Clearing the private flags also clears the binary semaphore that was marking the peripheral as busy.

```

void SspRelease(SspPeripheralType* psSspPeripheral_)
{
    /* Check to see if the peripheral is already released */
    if( !(psSspPeripheral_->u32PrivateFlags) & _SSP_PERIPHERAL_ASSIGNED )
    {
        return;
    }
    /* Disable interrupts */
    NVIC_DisableIRQ( (IRQn_Type)(psSspPeripheral_->u8PeripheralId) );
    NVIC_ClearPendingIRQ( (IRQn_Type)(psSspPeripheral_->u8PeripheralId) );

    /* Now it's safe to release all the resources in the target peripheral */
    psSspPeripheral_->pCsGpioAddress = NULL;
    psSspPeripheral_->pu8RxBuffer = NULL;
}

```

```

psSspPeripheral_>ppu8RxNextByte = NULL;
psSspPeripheral_>u32PrivateFlags = 0;

psSspPeripheral_>fnSlaveTxFlowCallback = NULL;
psSspPeripheral_>fnSlaveRxFlowCallback = NULL;

```

A receive operation should not be in progress when the peripheral is released. It was decided to leave this as a pre-condition to the function rather than enforce it since it is unlikely to occur if an application is written with at least some intelligence. In the EiE system, this is a safe assumption. In a generic, unknown use-case or production application, any assumptions are begging for disaster.

To address this, `SspRequest()` could fail or it could put the SSP state machine in a different state while the receive operation completes. This is a potentially big problem with synchronous peripherals – what happens if the Master stops clocking? Keeping a timeout is one option to mitigate the issue. From a Master application's perspective, the timeout is not useful since the Master should always finish clocking the bytes. This will be different next chapter when we look at I<sup>2</sup>C synchronous communications that get an Ack from the Slave every byte.

A timeout makes more sense in SPI Slave applications where a Slave should perhaps do something if it is expecting more bytes but has not received them. If CS is asserted, the Slave is still dependent on the Master, but at least an error message or system warning could be triggered if the Master seems to have disappeared.

More can be done if a transmit is in progress when `SspRelease()` is requested. Once interrupts are disabled, it is safe to clear the transmit buffer pointer. This can work for a message currently being sent, or messages that are queued for transmission but have not been sent yet. Abandoning messages is still strange and suggests the system is not working as expected, but at least the code can clean itself up.

```

/* Empty the transmit buffer if there were leftover messages */
while(psSspPeripheral_>psTransmitBuffer != NULL)
{
    UpdateMessageStatus(psSspPeripheral_>psTransmitBuffer->u32Token, ABANDONED);
    DeQueueMessage(&psSspPeripheral_>psTransmitBuffer);
}

/* Ensure the SM is in the Idle state */
Ssp_pfnStateMachine = SspSM_Idle;
} /* end SspRelease() */

```

#### 11.14.3 • `SspAssertCS()` / `SspDeassertCS()`



These two complementary functions give API access so tasks can manually control the CS line of their Master peripheral. They only apply when `eSspMode` is set as `SPI_MASTER_MANUAL_CS`. Write these two simple functions. Remember that CS is active-low. Use the stored `CsGpioAddress` member to then access the appropriate `PIO_SODR` and `PIO_CODR` registers. The DeAssert solution is shown.

```

void SspDeAssertCS(SspPeripheralType* psSspPeripheral_)
{
    if( psSspPeripheral_>eSspMode == SSP_MASTER_MANUAL_CS )
    {
        psSspPeripheral_>pCsGpioAddress->PIO_SODR = psSspPeripheral_>u32CsPin;
    }
}

```

```
} /* end SspDessertCS() *
```

#### 11.14.4 • SspReadByte() / SspReadData()

Read functions are slightly odd for SPI peripherals. From the design description table and flowchart, we know that the Master case is different than the Slave cases. The Slave cases are very similar to the UART reception where interrupts monitor incoming bytes and dump data into the configured receive buffers. The application who owns those buffers is responsible for parsing data as it comes in and ensuring the data is read quickly enough so that it doesn't get overwritten. With proper buffer management, this is trivial. Devices with flow control are even easier as incoming data can be throttled back if too much is coming in.

Master devices are a special case. First, they are the only mode in which "Read" functions must exist to allow a task to initiate a read operation. Second, receiving must involve transmission by sending transmit dummies for the number of bytes to be read. As described in the design, we have a conflict with regular transmit operations because the state machine looks at the Rx buffer pointer to decide if the Master is trying to send or receive. There is only one Rx buffer pointer, but there can be many queued transmit messages and currently, there is no mechanism to distinguish when the Rx Buffer was loaded and thus what message should go next.

As it stands, the receive message would always get priority. The use-case for half-duplex SPI is unlikely to run in to this scenario, so instead of adding additional complexity to the driver to handle it, we simply don't allow it. The API will deny the task if it calls a read function when there is already a read or write in progress.

SspReadData is a superset of SspReadByte, so we'll just look at that. The function begins by confirming all the pre-conditions necessary before allowing the request to be queued. The function returns FALSE if any of the conditions are not met.

```
bool SspReadData(SspPeripheralType* psSspPeripheral_, u16 u16Size_)
{
    /* Confirm Master Mode */
    if( (psSspPeripheral_>eSspMode == SSP_Slave) ||
        (psSspPeripheral_>eSspMode == SSP_Slave_FLOW_CONTROL) )
    {
        return FALSE;
    }

    /* Make sure no Tx or Rx function is already in progress */
    if( (psSspPeripheral_>u16RxBytes != 0) ||
        (psSspPeripheral_>psTransmitBuffer != NULL) )
    {
        return FALSE;
    }

    /* Do not allow if requested size is too large */
    if(u16Size_ > U16_MAX_TX_MESSAGE_LENGTH)
    {
        DebugPrintf("\r\nSSP message too large\n\r");
        return FALSE;
    }
}
```

An important side note: though you could combine all the conditions into a single compound "if" statement, the resulting assembly language can end up being very bloated. C-compilers reserve only a few of the core registers for regular processing

operations. Any variable must be fetched from memory into the available core registers. To test a conditional statement, usually, all the variables must be in core registers which means if there are more than three or four, the compiler then must add a lot of extra code to put together the combinations and get the correct logical output with the resources it has available. Two or three are usually safe, but four or more can lead to significant extra assembler code.



*Some high-level programmers take pride in packing as much as possible in a “single line of code” without realizing that they may be having the completely opposite effect. These kinds of statements can also be very difficult to look at and understand what is supposed to be happening.*

Once the conditions have been verified, the target peripheral's RxByte counter is loaded with the function parameter passed that indicates how many bytes should be read. The actual operation will be initiated in the Idle state when the peripheral is available.

```
/* Load the counter and return success */
psSspPeripheral->u16RxBytes = u16Size;
return TRUE;

} /* end SspReadData() */
```

#### 11.14.5 • SspQueryReceiveStatus()

Since data reception is not managed through the Messaging task, there is no existing API access for a task to know the state of reception. The SPI drivers add a query function just for reception. The query returns a value of SspRxStatusType which the calling task would have to handle appropriately. This only applies to Master mode devices since they are the only ones allowed to use the Read functions.

```
SspRxStatusType SspQueryReceiveStatus(SspPeripheralType* psSspPeripheral_)
{
    /* Confirm Master Mode */
    if( (psSspPeripheral_->eSspMode == SSP_Slave) ||
        (psSspPeripheral_->eSspMode == SSP_Slave_FLOW_CONTROL) )
    {
        return SSP_RX_INVALID;
    }
}
```

The function works primarily from the RxBytes member of the peripheral being queried. If there are no bytes waiting in RxBytes, then the `_SSP_PERIPHERAL_RX_COMPLETE` private flag determines if a message was just completed or there has been no receive activity since last time the query function was called. This is the only place where this flag gets cleared. If a task is simplistic and does not require Rx status, it does not have to call `SspQueryReceiveStatus()` and `_SSP_PERIPHERAL_RX_COMPLETE` can remain set indefinitely.

If a task relies on the Rx status, then it must be diligent to call Query to keep the flag state accurate. Otherwise, the status of the reception will be inaccurate. This was a design choice – we could have forced the `RX_COMPLETE` flag clear whenever a new reception was started but chose to let it latch until specifically cleared by the application.

```
/* Check for no current bytes queued */
if(psSspPeripheral_->u16RxBytes == 0)
{
    /* If a transfer just finished and has not be queried... */
    if(psSspPeripheral_->u32PrivateFlags & _SSP_PERIPHERAL_RX_COMPLETE)
```



```

    {
        psSspPeripheral->u32PrivateFlags &= ~_SSP_PERIPHERAL_RX_COMPLETE;
        return SSP_RX_COMPLETE;
    }
    /* Otherwise it's just empty */
    else
    {
        return SSP_RX_EMPTY;
    }
}

```

In the other case where RxBytes is not 0, the peripheral is either waiting or in the process of receiving which the driver knows based on `_SSP_PERIPHERAL_RX` which gets set by the task when the reception is queued in `SspSM_Idle`.

```

/* If there are bytes waiting, check if waiting or in progress */
else
{
    if(psSspPeripheral->u32PrivateFlags & _SSP_PERIPHERAL_RX)
    {
        return SSP_RX_RECEIVING;
    }
    else
    {
        return SSP_RX_WAITING;
    }
}
} /* end SspQueryReceiveStatus() */

```

#### 11.14.6 • SspGenericHandler()

With an understanding of the data flow in the SSP driver, the main interrupt handler can be written. There are more cases to handle than the UART driver, but each one is relatively simple. A good strategy when implementing a more complicated design is to frame out the functions and add comments that reflect what the code should do. Simple functions that return preset values can be used so that different pieces of code can be tested independently of others within the system using known return values from functions. Writing this driver code is complicated when it comes to putting everything together with the state machine and the interrupt system. Though we encourage designing and thinking through a problem as completely as possible, there is a lot involved here with many interdependencies.

On a side note, this is a good place to briefly mention Test Driven Development (TDD). TDD is a design approach that can be very helpful in many situations despite the additional overhead required. In the end, TDD potentially provides a very robust firmware environment with close to 100% continual test coverage designed into the firmware system. As with any design strategies, there are tradeoffs. The EiE firmware system was not developed using TDD for several reasons, one being that low-level driver code is arguably “not applicable” to TDD. The reader is encouraged to find out more about TDD as it has a great deal of merit in many applications.

Since we have the advantage of hindsight, we’ll explain the finished code for the `SspGenericHandler()` and tie everything together in the next section about the Idle state. If you understood the UART chapter, then the SSP handler will feel quite familiar. Of course, it would be best if you attempted to write it yourself and compared your code to this solution.



A few local variables are required. The US\_CSR register changes when read, so it must be captured since the bits will be referenced several times to check current flags. The first interrupt to deal with comes from the CS line and is configured to interrupt on either assertion or de-assertion. This only applies to Slave peripherals.

```
static void SspGenericHandler(void)
{
    u32 u32Byte;
    u32 u32Timeout;
    u32 u32Current_CSR;

    /* Get a copy of CSR because reading it changes it */
    u32Current_CSR = SSP_psCurrentISR->pBaseAddress->US_CSR;

    /** CS change state interrupt - only enabled on Slave SSP peripherals ***/
    if( (SSP_psCurrentISR->pBaseAddress->US_IMR & AT91C_US_CTSIC) &&
        (u32Current_CSR & AT91C_US_CTSIC) )
```

The state of CS will be communicated to applications via the `_SSP_CS_ASSERTED` flag which is part of the Global application flags variable for each peripheral. Though the CS line is assigned to the peripheral, the PDSR register can still be used to check the current state and determine the edge that resulted in the interrupt. The asserted state is checked first. The “COMPLETE” flag states are cleared since they must be in this state at the start of a transaction. The application should have some control over what is happening so that this data isn't lost.

```
{
    /* Is the CS pin asserted now? */
    if((SSP_psCurrentISR->pCsGpioAddress->PIO_PDSR & SSP_psCurrentISR->u32CsPin) == 0)
    {
        /* Flag that CS is asserted and make sure TX and RX COMPLETE flags are clear */
        *SSP_pu32SspApplicationFlagsISR |= _SSP_CS_ASSERTED;
        *SSP_pu32SspApplicationFlagsISR &= ~(_SSP_TX_COMPLETE | _SSP_RX_COMPLETE);
```

Mode-specific initializations are also made. Standard SSP\_Slave configurations load US\_THR with the defined dummy byte if they are going to be receiving. Flow control devices need their RXRDY interrupt active.

```
/* No flow control Slave receiving: be ready to respond with dummy */
if(SSP_psCurrentISR->eSspMode == SSP_Slave)
{
    if(SSP_psCurrentISR->pTransmitBuffer == NULL)
    {
        SSP_psCurrentISR->pBaseAddress->US_THR = SSP_DUMMY_BYTE;
    }
}

/* Flow control Slaves should have their RXRDY interrupt enabled */
if(SSP_psCurrentISR->eSspMode == SSP_Slave_FLOW_CONTROL)
{
    SSP_psCurrentISR->pBaseAddress->US_IER |= AT91C_US_RXRDY;
}
}
```

When CS is deasserted, the global flag is updated and then the peripheral is checked to see if a transmission was in progress. For a Slave that was busy transmitting, it is important to clear the message from the Message system. For lack of a better option, in

this case, the private flag `_SSP_TX_COMPLETE` is set.

```
else
{
    /* Flag that CS is deasserted */
    *SSP_pu32SspApplicationFlagsISR &= ~_SSP_CS_ASSERTED;

    /* If a transmit was interrupted, then the message needs to be aborted. */
    if(SSP_psCurrentISR->u32PrivateFlags & _SSP_PERIPHERAL_TX)
    {
        /* Clean up the message status and flags */
        SSP_psCurrentISR->pBaseAddress->US_IDR = AT91C_US_ENDTX;

        SSP_psCurrentISR->u32PrivateFlags &= ~_SSP_PERIPHERAL_TX;
        UpdateMessageStatus(SSP_psCurrentISR->psTransmitBuffer->u32Token, ABANDONED);
        DeQueueMessage(&SSP_psCurrentISR->psTransmitBuffer);

        *SSP_pu32SspApplicationFlagsISR |= _SSP_TX_COMPLETE;
    }
}
```

Other clean-up is mode-specific. Standard Slaves need the PDC counters reset as it is unknown when the interrupt occurred. The PDC pointers can be left in their current state.

```
/* PDC counters reset for next transmission */
if(SSP_psCurrentISR->eSspMode == SSP_Slave)
{
    SSP_psCurrentISR->pBaseAddress->US_RCR = 1;
    SSP_psCurrentISR->pBaseAddress->US_RNCR = 1;
}
```

Slaves with flow control should ensure their RXRDY interrupt is enabled since it gets disabled for transmit. If it was receiving, it is safe to set an already set bit so no code has to be wasted making the distinction.

```
/* Special case for an interrupted flow control mode */
if(SSP_psCurrentISR->eSspMode == SSP_Slave_FLOW_CONTROL)
{
    /* Re-enable Rx interrupt, clean-up the operation */
    SSP_psCurrentISR->pBaseAddress->US_IER = AT91C_US_RXRDY;
}

} /* end of CS de-asserted */

} /* end CS change state interrupt */
```

In the order of our solution, TXEMPTY is the next interrupt processed. As a quick exercise, make sure you know which configuration and data direction this interrupt should correspond to. You should see that this is only applicable to Slave devices with flow-control. No other type of peripheral should have this interrupt enabled.

```
/** SSP ISR transmit handling for flow-control devices that do not use DMA */
if( (SSP_psCurrentISR->pBaseAddress->US_IMR & AT91C_US_TXEMPTY) &&
    (u32Current_CSR & AT91C_US_TXEMPTY) )
{

```

The TXEMPTY interrupt means that the byte previously loaded to `US_THR` has been placed in the shift register and finished clocking out. The counter `u32CurrentTxBytesRemaining`

of the SSP peripheral that triggered the interrupt holds the number of bytes in the current data transfer and must be decremented. Since the EiE system is half-duplex, the dummy byte that would have arrived in US\_RHR can be read and discarded to avoid overrun flags.

```
/* Decrement counter and read dummy byte */
SSP_psCurrentISR->u32CurrentTxBytesRemaining--;
u32Byte = SSP_psCurrentISR->pBaseAddress->US_RHR;
```

If there are more bytes to send, the non-circular data buffer at pu8CurrentTxData needs to be advanced to the next byte. The next byte to transmit is read, processed, optionally flipped from MSB to LSB, and loaded to US\_THR. Cortex-M processors have a bit flipping instruction, so other than the check to see if it's required, the operation is very fast. If a line of C code was used for the operation without specifying the use of the RBIT instruction, the operation could translate into dozens of lines of assembly code.

```
if(SSP_psCurrentISR->u32CurrentTxBytesRemaining != 0)
{
    /* Advance pointer (non-circular), load next byte, and callback */
    SSP_psCurrentISR->pu8CurrentTxData++;
    u32Byte = 0x000000FF & *SSP_psCurrentISR->pu8CurrentTxData;

    /* LSB first? Use intrinsic function to flip bits (single instruction) */
    if(SSP_psCurrentISR->eBitOrder == SSP_LSB_FIRST)
    {
        u32Byte = __RBIT(u32Byte) >> 24;
    }

    /* Clears interrupt flag and invoke callback */
    SSP_psCurrentISR->pBaseAddress->US_THR = (u8)u32Byte;
}
```

If u32CurrentTxBytesRemaining has reached zero, then the transfer is over so the system needs to clean up and be ready for the next transfer. The TXEMPTY interrupt will continue to fire unless THR is loaded but there is no more data to send. Therefore, we disable it. Status flags and the message status should be updated in the Messaging task. Note that the order in which this happens is purposely chosen to avoid reloading certain variable addresses assuming the compiler recognizes that. It might not make a difference unless optimizations are turned on. Organizing lines of code like this must be done within the bounds of what the flow of the program needs to be but is a subtle technique to possibly reduce flash resources. The assumption is that SSP\_psCurrentISR is referenced several times, though if the assembly code is examined it might only be the specific member addresses that are used in the registers. It was a good spot to point this out, though!

```
else
{
    /* Done! Disable TX interrupt */
    SSP_psCurrentISR->pBaseAddress->US_IDR = AT91C_US_TXEMPTY;

    /* Clean up the message status and flags */
    *SSP_pu32SspApplicationFlagsISR |= _SSP_TX_COMPLETE;
    UpdateMessageStatus(SSP_psCurrentISR->psTransmitBuffer->u32Token, COMPLETE);
    DeQueueMessage(&SSP_psCurrentISR->psTransmitBuffer);
    SSP_psCurrentISR->u32PrivateFlags &= ~_SSP_PERIPHERAL_TX;

    /* Re-enable Rx interrupt and make final call to callback */
    SSP_psCurrentISR->pBaseAddress->US_IER = AT91C_US_RXRDY;
}
```

The final step in both cases is to run the Tx callback function.

```
/* Both cases use the callback */
SSP_psCurrentISR->fnSlaveTxFlowCallback();

} /* end AT91C_US_TXEMPTY */
```

Next, it makes sense to write the code to handle the flow control receive RXRDY interrupt. This also applies to the Slave flow control cases. Note that the dedicated SPI peripheral driver only uses the peripheral-level interrupts, so the design here is the best reference for the other driver.

```
/** SSP ISR handling for Slave Rx with flow control (no DMA) */
if( (SSP_psCurrentISR->pBaseAddress->US_IMR & AT91C_US_RXRDY) &&
    (SSP_psCurrentISR->pBaseAddress->US_CSR & AT91C_US_RXRDY) )
{
```

RXRDY occurs when US\_RHR has a new byte. Read the byte, flip it MSB to LSB if necessary, and drop it in the client's Rx buffer. Set `_SSP_RX_COMPLETE` since the system runs based on single-byte transfers and invoke the Rx callback. Watch the pointer notation and variable size here. Order of operations for pointers is sometimes non-intuitive. Confirming that the pointer does what you think it is doing is easily done in the debugger if you're good at understanding the low-level addressing. Always check the edge cases!

```
/* Pull the byte out of the receive register into the Rx buffer */
u32Byte = 0x000000FF & SSP_psCurrentISR->pBaseAddress->US_RHR;

/* LSB first? Use intrinsic function to flip bits (single instruction) */
if(SSP_psCurrentISR->eBitOrder == SSP_LSB_FIRST)
{
    u32Byte = __RBIT(u32Byte)>>24;
}

/* DEBUG */
if((u8)u32Byte == 0xff)
{
    SSP_u32AntCounter++;
}

/* Send the byte to the Rx buffer and set _SSP_RX_COMPLETE */
**(SSP_psCurrentISR->ppu8RxNextByte) = (u8)u32Byte;
*SSP_pu32SspApplicationFlagsISR |= _SSP_RX_COMPLETE;

/* Invoke callback */
SSP_psCurrentISR->fnSlaveRxFlowCallback();
} /* end of receive with flow control */
```

The remaining two SSP interrupt sources that can be active are for peripheral configurations that use the PDC. For reception (ENDRX) this includes both Master and regular Slave devices.

```
/* ENDRX Interrupt when all requested bytes have been received */
if( (SSP_psCurrentISR->pBaseAddress->US_IMR & AT91C_US_ENDRX) &&
    (u32Current_CSR & AT91C_US_ENDRX) )
{
```

Starting with Master, u16RxBytes needs to be zeroed and the Rx flags adjusted. We have a debug counter in place to count the total number of received bytes per this interrupt. If the peripheral has left CS control to the SSP system, CS needs to be de-asserted. No more activity should be occurring on the peripheral at this point, so the transceiver is disabled along with the ENDRX interrupt.

```
/* Master mode and Slave mode operate differently */
if( (SSP_psCurrentISR->eSspMode == SSP_Master_AUTO_CS) ||
    (SSP_psCurrentISR->eSspMode == SSP_Master_MANUAL_CS) )
{
    /* Reset the byte counter and clear the RX flag */
    SSP_psCurrentISR->u16RxBytes = 0;
    SSP_psCurrentISR->u32PrivateFlags &= ~_SSP_PERIPHERAL_RX;
    SSP_psCurrentISR->u32PrivateFlags |= _SSP_PERIPHERAL_RX_COMPLETE;
    SSP_u32RxCounter++;

    /* Deassert CS for SSP_Master_AUTO_CS transfers */
    if(SSP_psCurrentSsp->eSspMode == SSP_Master_AUTO_CS)
    {
        SSP_psCurrentISR->pCsGpioAddress->PIO_SODR = SSP_psCurrentISR->u32CsPin;
    }

    /* Disable the receiver and transmitter */
    SSP_psCurrentISR->pBaseAddress->US_PTCR = AT91C_PDC_RXTDIS | AT91C_PDC_TXTDIS;
    SSP_psCurrentISR->pBaseAddress->US_IDR = AT91C_US_ENDRX;
}
}
```

Slave reception is configured for single-byte operation. The PDC counters are kept at 1, and the Rx buffer pointers need to leap-frog each other. Slave Rx buffers are circular, so pointer wrap-around must be handled correctly. It is safe to move these pointers because the interrupt is already disabled since the ISR is active. From the user guide, we see that RNCR (or RCR) is written to clear the ENDRX flag. This needs to be done because the interrupt remains active and ready for the next byte.

```
else
{
    /* Flag that a byte has arrived */
    *SSP_pu32SspApplicationFlagsISR |= _SSP_RX_COMPLETE;

    /* Update pointer to the next valid Rx location */
    SSP_psCurrentISR->pBaseAddress->US_RNPR++;
    if(SSP_psCurrentISR->pBaseAddress->US_RPR ==
        (u32)(SSP_psCurrentISR->pu8RxBuffer + (u32)SSP_psCurrentISR->u16RxBufferSize) )
    {
        SSP_psCurrentISR->pBaseAddress->US_RPR = (u32)SSP_psCurrentISR->pu8RxBuffer;
    }

    /* Write RNCR to 1 to clear the ENDRX flag */
    SSP_psCurrentISR->pBaseAddress->US_RNCR = 1;
}
} /* end ENDRX handling */
```

The final interrupt is ENDTX which applies to all peripherals configured to use the PDC for transmit operations. The first distinction made is whether ENDTX is happening because of an actual transmit which would have an associated message token. If so, the message needs to be updated and dequeued and the TX flag cleared. If it was just a dummy transmit from a receive operation none of that applies.

```

/* ENDTX Interrupt when all requested transmit bytes have been sent (if enabled) */
if( (SSP_psCurrentISR->pBaseAddress->US_IMR & AT91C_US_ENDTX) &&
    (u32Current_CSR & AT91C_US_ENDTX) )
{
    /* If this was a non-dummy transmit... */
    if(SSP_psCurrentISR->u32PrivateFlags & _SSP_PERIPHERAL_TX)
    {
        /* Update this message token status and then DeQueue it */
        UpdateMessageStatus(SSP_psCurrentISR->psTransmitBuffer->u32Token, COMPLETE);
        DeQueueMessage(&SSP_psCurrentISR->psTransmitBuffer);
        SSP_psCurrentISR->u32PrivateFlags &= ~_SSP_PERIPHERAL_TX;
    }
}

```

ENDTX occurs as a function of the PDC when it loads the last byte it has to US\_THR in the peripheral. That means the actual transmit of that byte is likely not yet complete by the time ENDTX is being processed. If the timing is just right, this can cause big problems if it is not factored in. To mitigate, we concluded that it makes sense from both a functional (i.e. this will always work) and timing (i.e. this will be very fast) perspective to wait out the transmit inside this ISR. This is done by watching the TXEMPTY flag in the peripheral's US\_CSR. To be safe, a timeout is added. Unless the SSP peripheral is configured with a very slow clock, this only adds a few microseconds to the ISR.

```

/* Allow the peripheral to finish clocking out the Tx byte */
u32Timeout = 0;
while ( !(SSP_psCurrentISR->pBaseAddress->US_CSR & AT91C_US_TXEMPTY) &&
        u32Timeout < SSP_TXEMPTY_TIMEOUT)
{
    u32Timeout++;
}

```

As a final step in the ENDTX handler, SSP\_MASTER\_AUTO\_CS peripherals should deassert CS.

```

/* Deassert chip select when the buffer and shift register are totally empty */
if(SSP_psCurrentSsp->eSspMode == SSP_MASTER_AUTO_CS)
{
    SSP_psCurrentISR->pCsGpioAddress->PIO_SODR = SSP_psCurrentISR->u32CsPin;
}

} /* end ENDTX interrupt handling */
} /* end SspGenericHandler() */

```

### 11.15 • Ssp State Machine

With all the API functions written and the ISR code ready, the final step is to put it all together in the SSP application. This is where remembering everything that was put together in the API and ISR can be difficult. If the State Machine operation does not match the rest of the design, there can be a lot of confusion about how the system works, and/or the system might not work at all. This is where detailed drawings can help to ensure the implementations of the different systems match. It is easy to end up with redundant functionality like clearing a flag in two places. This is highly dangerous because certain assumptions can be made about how the code is running. Code that is working by accident is prone to fail at some point in the future.



Only an Idle state is needed which cycles continuously through the configured peripherals. Action taken is based on the buffer pointers and Tx/Rx flags. Set up this framework first, then we will add the code in to match the flowchart design that we are following.

```
static void SspSM_Idle(void)
{
    u32 u32Byte;

    /* Check all SPI/SSP peripherals for message activity */
    if( ( (SSP_psCurrentSsp->psTransmitBuffer != NULL) ||
        (SSP_psCurrentSsp->u16RxBytes !=0) ) &&
        !(SSP_psCurrentSsp->u32PrivateFlags &
          (_SSP_PERIPHERAL_TX | _SSP_PERIPHERAL_RX) ) )
    {
        /* CONFIGURATION-SPECIFIC CODE WILL GO HERE */
    }

    /* Adjust to check the next peripheral next time through */
    switch (SSP_psCurrentSsp->u8PeripheralId)
    {
        case AT91C_ID_US0:
            SSP_psCurrentSsp = &SSP_Peripheral1;
            break;

        case AT91C_ID_US1:
            SSP_psCurrentSsp = &SSP_Peripheral2;
            break;

        case AT91C_ID_US2:
            SSP_psCurrentSsp = &SSP_Peripheral0;
            SSP_u32Flags &= ~_SSP_MANUAL_MODE;
            break;

        default:
            DebugPrintf("Invalid SSP attempt\r\n");
            SSP_psCurrentSsp = &SSP_Peripheral0;
            break;
    } /* end switch */
} /* end SspSM_Idle() */
```

The code follows the flowchart though has a few more details that weren't shown in the high-level design. Once a peripheral is deemed active, a check is made for SSP\_MASTER\_AUTO\_CS mode to assert CS.



```
/* For an SSP_Master_AUTO_CS device, start by asserting chip select
(SSP_Master_MANUAL_CS devices should have already asserted CS) */
if(SSP_psCurrentSsp->eSspMode == SSP_MASTER_AUTO_CS)
{
    SSP_psCurrentSsp->pCsGpioAddress->PIO_CODR = SSP_psCurrentSsp->u32CsPin;
}
```

This is followed by the action to determine if the peripheral is starting transmission or reception. Reception is a simpler case in the state machine as only certain configurations allow it. This condition is already imposed by the API, so no additional checks are made. If RxBytes is not 0, then the reception is flagged by setting \_SSP\_PERIPHERAL\_RX. This is one of the main points at which the system could be improved to avoid limiting only one queued message on a given peripheral.



```

/* Check if the message is receiving based on expected byte count */
if(SSP_psCurrentSsp->u16RxBytes !=0)
{
    /* Receiving: flag that the peripheral is now busy */
    SSP_psCurrentSsp->u32PrivateFlags |= _SSP_PERIPHERAL_RX;
}

```

The receive buffer is initialized to all dummy bytes per the design decision to use the Rx buffer as the source for the transmit dummies. This also means that both transmit and receive PDC pointers are set to the RxBuffer, and both the Tx and Rx PDC counters are loaded with u16RxBytes.

```

/* Initialize the receive buffer */
memset(SSP_psCurrentSsp->pu8RxBuffer, SSP_DUMMY_BYTE,
        SSP_psCurrentSsp->u16RxBufferSize);

/* Load the PDC counter and pointer registers */
SSP_psCurrentSsp->pBaseAddress->US_RPR =
    (unsigned int)SSP_psCurrentSsp->pu8RxBuffer;
SSP_psCurrentSsp->pBaseAddress->US_TPR =
    (unsigned int)SSP_psCurrentSsp->pu8RxBuffer;
SSP_psCurrentSsp->pBaseAddress->US_RCR = SSP_psCurrentSsp->u16RxBytes;
SSP_psCurrentSsp->pBaseAddress->US_TCR = SSP_psCurrentSsp->u16RxBytes;

```

Since RCR is loaded, the ENDRX interrupt flag will be clear. To get everything moving, the peripheral's transmitter and receiver are enabled. At this point, everything else is handed off to the interrupt system.

```

SSP_psCurrentSsp->pBaseAddress->US_IER = AT91C_US_ENDRX;

/* Enable the receiver and transmitter to start the transfer */
SSP_psCurrentSsp->pBaseAddress->US_PTCR = AT91C_PDC_RXTEN | AT91C_PDC_TXTEN;

} /* End of receive function */

```

There are three different transmit cases but they all start off by setting `_SSP_PERIPHERAL_TX` and updating the source message status to "SENDING".

```

else
{
    /* Transmitting: update message status and flag peripheral */
    UpdateMessageStatus(SSP_psCurrentSsp->psTransmitBuffer->u32Token, SENDING);
    SSP_psCurrentSsp->u32PrivateFlags |= _SSP_PERIPHERAL_TX;
}

```

The flow control case is handled first. We know that the message to send and its size are sitting at `psTransmitBuffer`, so these are copied into the "Current" values for the SSP. LSB to MSB flipping is done too, if necessary.

```

/* TRANSMIT SPI_SSP_Slave_FLOW_CONTROL */
if(SSP_psCurrentSsp->eSspMode == SSP_Slave_FLOW_CONTROL)
{
    /* Slave device with flow control */

    /* Load in the message parameters. */
    SSP_psCurrentSsp->u32CurrentTxBytesRemaining =
        SSP_psCurrentSsp->psTransmitBuffer->u32Size;
    SSP_psCurrentSsp->pu8CurrentTxData =
        SSP_psCurrentSsp->psTransmitBuffer->pu8Message;
}

```

```

/* LSB first? */
u32Byte = 0x000000FF & *SSP_psCurrentSsp->pu8CurrentTxData;
if(SSP_psCurrentSsp->eBitOrder == SSP_LSB_FIRST)
{
    u32Byte = __RBIT(u32Byte)>>24;
}

```

For the EiE half-duplex design, the Rx function is disabled to avoid having to deal with that interrupt (recall from writing `GenericHandler()` that the byte is still read so overflow flags do not get set). For “safety” the peripheral is reset before the transfer begins. This is admittedly a little hack to account for anything unexpected that might have happened to the peripheral which may have caused it to be in an unexpected state. This is a great example of how early, crappy code can remain in a design where the design should not depend on it, but if the code is removed it might stop working. Removing this code would be essential for a production system, as would the corresponding verification and refactoring to ensure that all potential issues that the code might be mitigating are properly addressed in firmware. For now, we kept it in.

US\_THR is loaded, the TX\_EMPTY interrupt is then safely enabled, and the Tx callback function is run which should manage the hardware flow control to provide the flow control signals and get the Master to start clocking.

```

/* This driver assumes half-duplex comms, so disable RX interrupt for now */
SSP_psCurrentSsp->pBaseAddress->US_IDR = AT91C_US_RXRDY;

/* Reset the transmitter, load THR and enable TXEMPTY */
SSP_psCurrentSsp->pBaseAddress->US_CR = (AT91C_US_RSTTX);
SSP_psCurrentSsp->pBaseAddress->US_CR = (AT91C_US_TXEN);
SSP_psCurrentSsp->pBaseAddress->US_THR = (u8)u32Byte;
SSP_psCurrentSsp->pBaseAddress->US_IER = AT91C_US_TXEMPTY;

/* Trigger the callback to provide flow-control*/
SSP_psCurrentSsp->fnSlaveTxFlowCallback();
}

```

The final piece of code in the driver covers both the Master transmit cases and the regular Slave transmit. All three are set up identically for a single PDC transfer. This means only the current PDC registers are adjusted for the new message, the ENDTX interrupt is enabled, and the main transmit function is started.

```

/* A Master or Slave device without flow control uses the PDC */
else
{
    /* Load the PDC counter and pointer registers */
    SSP_psCurrentSsp->pBaseAddress->US_TPR =
        (unsigned int)SSP_psCurrentSsp->psTransmitBuffer->pu8Message;
    SSP_psCurrentSsp->pBaseAddress->US_TCR =
        SSP_psCurrentSsp->psTransmitBuffer->u32Size;

    SSP_psCurrentSsp->pBaseAddress->US_IER = AT91C_US_ENDTX;

    /* Enable the transmitter to start the transfer */
    SSP_psCurrentSsp->pBaseAddress->US_PTCR = AT91C_PDC_TXTEN;
}
} /* End of transmitting function */
}

```

The only difference between the Master and Slave at this point is shown in the flowchart design. The Master will start clocking immediately but the Slave will be waiting for the Master. In either case, no further code is required. This would be where a timeout for the Slave could be added since the peripheral will be stuck in this state if the Master does not start clocking.

With all of that in place, the SSP driver is ready to go. You can examine the simpler version of the dedicated SPI peripheral and should see the similarities between the two drivers. A great exercise would be to write that driver yourself. There is value both in trying to write it similarly to the USART SSP version, but also in designing it entirely from scratch to see first-hand what sort of problems arise and therefore what design decisions must be made. Our simple SPI driver will be used for the Blade example in the next section.

### 11.16 • Blade Daughter Board Interface

The EiE development boards come stock with more built-in accessories than a lot of the other popular main-stream systems. However, all good development boards rely on easy expansion through daughter boards to reduce cost and offer infinite updateability.

The EiE development boards are no exception and offer a robust interface to expansion boards. The staples of embedded systems are designed already offering common services like GPS, Wifi, accelerometer, data flash, etc. There is even an interface board that supports Arduino Shields so any Arduino part can be used with the EiE system if you write the driver.

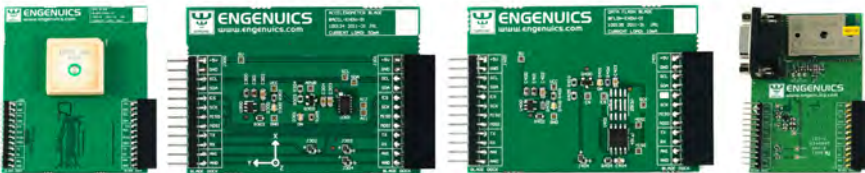


Figure 11-11 EiE development boards

The EiE system is meant for teaching and for embedded developers looking to prototype full systems before going to a form factor design. It was specifically designed in a daisy-chain physical layout so developers could easily access all the daughter board circuitry during development, rather than having circuits buried in a board stack. The tradeoff is that once a few Blades are chained together, the hardware footprint begins to take over the desk it is sitting on. For the exercise, we will use the ASCII dev board and a data flash Blade.

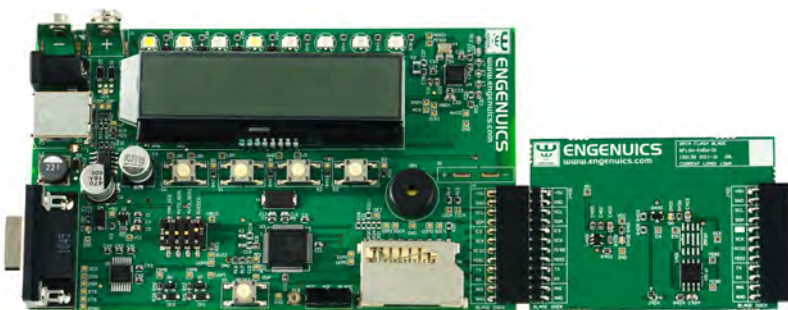


Figure 11-12 ASCII development board with data flash blade

Though there are stock Blades available, the intended use-case is for developers to create their own Blades with the exact hardware that they have designed in. In our own designs, we will often put two or three variations of a part on a Blade. This is a fast and low-cost way to test multiple devices and scrub out any footprint or hardware design errors that might slip in when a new circuit is first built. This significantly de-risks the first form-factor build. The Blade connection is totally open source and reference files are provided on the EiE website.

The EiE standard 12-pin Blade connector was designed to be as small as possible but still offer enough connectivity to build up a prototype for a real system. The connection offers an SPI, I<sup>2</sup>C, and UART port as well as two analog inputs. Any of the 10 lines can be repurposed for general purpose digital input or output. To date this is a manual process, though eventually a Blade API could be written that provides easier configuration and use of the Blade connection.

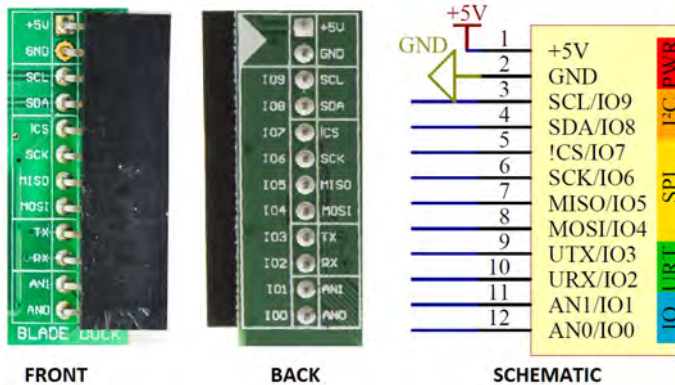


Figure 11-13 EiE standard 12-pin Blade connector

Every Blade is rated with a peak current draw and graphical indication of any pins it “consumes” when connected. Blades are designed to take power from the 5V rail which is expected to be able to source 500mA from the USB power supply. Since most circuitry runs at 3.3V, a Blade board typically has an onboard LDO to make the conversion. The expectation is that a Blade will have the hardware to run from the 5V supply whether that be through linear or buck regulation down or boost up. A Blade could also connect to an external supply.

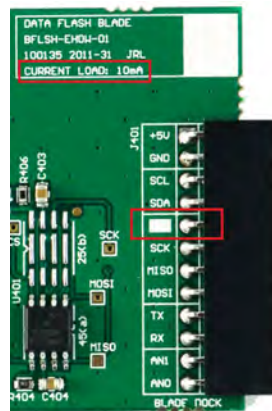


Figure 11-14 Blade board current rating and connection

Ideally, Blades are designed with I<sup>2</sup>C interfaces to their circuitry to allow maximum expansion. Realistically, that is hardly a real requirement as most product development happens (or could be done) one peripheral at a time. From designing in industry, by the time the individual low-level hardware and firmware drivers have been perfected, a form-factor design with all the hardware integrated is the next logical step. At that point, firmware integration can happen with all the pieces of the puzzle together. Prototyping piece-wise is often sufficient to prove out most of a design and de-risk the first full circuit.

### 11.17 • Blade Example Project

For this mini-example, we'll do a basic connection to a data flash Blade since data flash parts are very common on small embedded devices that need some bulk, non-volatile storage. The only communication we'll do is to read the flash's ID information to prove the driver and basic function is working. You can extend this into a full-blown application to use the data flash in your projects.

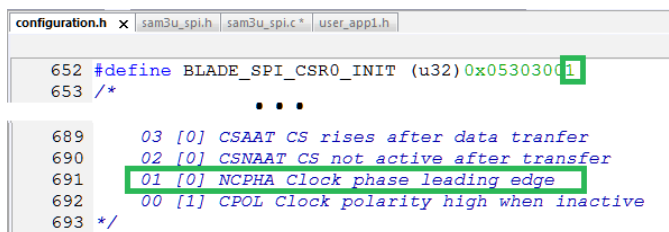
#### 11.17.1 • Blade Firmware Configuration Defaults and Interface

Remember that the SPI driver for the Blade connection uses the dedicated SPI peripheral which was not described thoroughly in this chapter. This is on purpose so that you can do some additional learning as you look at this example code and compare it to the USART version and possibly your own version if you took the challenge to write the SPI driver on your own.

The default PIO setup values for the Blade enable all the peripheral functions. A task using the Blade interface should re-purpose the correct IO lines for the specific application during the initialization function. In this case, the current defaults are almost sufficient.

If the communications peripherals are used, then configuration.h can be updated to reflect the connections being made. This is where a Blade task could be very helpful to offer an API to more easily facilitate the configuration. For now, it remains very low-level and requires a developer to understand PIO and peripheral hardware configuration. Again, the EiE system is not intended for rapid deployment of any possible hardware configuration. It is expected that it be used in a long development cycle towards real product development. Obfuscating details that will eventually need to be figured out is counter-productive to our goals! "Front-load effort" is our mantra.

For the data flash application, the memory's datasheet indicates that the IC samples data on the leading edge, which means the clock phase bit NCPHA must be 0. It is admittedly difficult to line up the differing terminology of various vendors and parts to decide exactly what the right settings should be (for MSB-first vs. LSB-first config as well). When testing our application, the data flash was not responding until the bit was flipped. To keep this example quick, the change was made in the CSR INIT value.



```
configuration.h x sam3u_spi.h sam3u_spi.c * user_app1.h
652 #define BLADE_SPI_CSR0_INIT (u32) 0x05303001
653 /*
    . . .
689 03 [0] CSAAT CS rises after data transfer
690 02 [0] CSNAAT CS not active after transfer
691 01 [0] NCPHA Clock phase leading edge
692 00 [1] CPOL Clock polarity high when inactive
693 */
```

Figure 11-15 Clock phase leading edge

The mini-application is written in `userapp1`. Local global variables are needed for the SPI configuration, receive buffer and messages.

```
static SpiPeripheralType* UserApp1_Spi; /* Pointer to peripheral object */
static SpiConfigurationType UserApp1_eBladeDataflashSPI;

static u8 UserApp1_au8RxBuffer[U16_UA1_RX_BUFFER_SIZE];
static u8 *UserApp1_pu8RxBufferNextChar;
static u8 *UserApp1_pu8RxBufferUnreadChar;

static u8 UserApp1_au8MessageManufacturerID[] =
    {0x9F, SPI_DUMMY, SPI_DUMMY, SPI_DUMMY, SPI_DUMMY};
static u32 UserApp1_CurrentMsgToken;
```

The message to request the device information from the data flash is described in the data flash datasheet. No one would “just know” this information. Working with external circuits requires careful examination of the interface requirements in the datasheet. This starts with the hardware requirements if you are in charge of hardware design, but even if someone else built the circuit you still need to understand how to use it properly.

There is a lot more complexity in the logical information as you would see if you read through the datasheet to learn how to use the data flash properly. Details about reading the ID bytes are highlighted here.

#### Manufacturer and Device ID Information

##### Byte 1 – Manufacturer ID

Hex Value	JEDEC Assigned Code							
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1FH	0	0	0	1	1	1	1	1

Manufacturer ID 1FH = Atmel

##### Byte 2 – Device ID (Part 1)

Hex Value	Family Code			Density Code				
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
27H	0	0	1	0	0	1	1	1

Family code 001 = DataFlash

Density code 00111 = 32Mb

##### Byte 3 – Device ID (Part 2)

Hex Value	MLC Code			Product Version Code				
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
00H	0	0	0	0	0	0	0	1

MLC code 000 = 1-bit/cell technology

Product version 00001 = Second version

##### Byte 4 – Extended Device Information String Length

Hex Value	Byte Count							
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
00H	0	0	0	0	0	0	0	0

Byte count 00H = 0 bytes of information

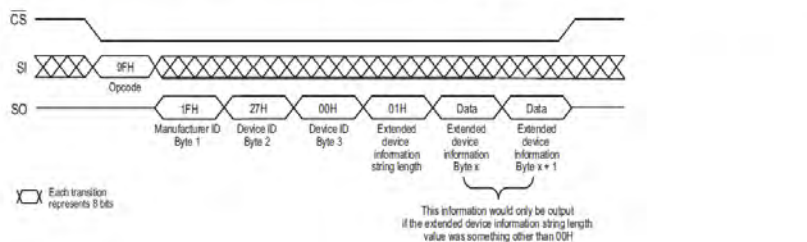


Figure 11-16 Manufacturer's information



There are often very particular requirements for how the MCU must interact with the part - hopefully, your driver function is able to do what you need. It is not uncommon to have to tweak the behavior of the driver to ensure the signaling ends up being compatible. At this point, you must decide at what level those changes need to occur.

For example, the SPI driver we first wrote expected distinct half-duplex communication. That means it was first coded to throw away receive bytes during a transmit operation. This is how the USART SSP code is written. However, the data flash combines writes and reads in a single transaction so throwing away Rx bytes will not work.

To read the ID data, 0x9F is sent (Master to Slave write) and then additional clocks are provided to read the information (Slave to Master read). CS is not allowed to be deasserted and reasserted between the two. Therefore the driver was adjusted to be more full-duplex so that receive bytes are always read with transmit bytes even if they are known to be dummies. This also means our user app must know what is expected in the Rx buffer. This, in turn, imposes limits on how communication between the MCU and data flash would have to work.

Any time a new low-level driver is written, it is suggested to thoroughly examine the signaling that results from the firmware. Understanding a peripheral's behavior just from reading the datasheet is difficult to get 100% correct. It should still be attempted - it is highly recommended to get as close as possible from carefully reading and implementing the details described. This ensures the depth and breadth of knowledge you need to have. Guessing and testing and not correlating what is physically happening to the firmware is begging for disaster. Even if a system appears to be working, making sure it is working for all the right reasons and in all the right ways is important.

All the SPI signals can be verified from a scope trace. Any logic analyzer can be used and this is a critical resource to have when working with low-level drivers. There are many low-cost, PC-based USB logic analyzers that have sufficient speed for looking at MCU peripheral communications. A note of caution: any digital analyzer will show "perfect" bit transitions based on the signals being interpreted. Checking signal integrity in the analog domain is also an important verification step. We'll see more about that in the next chapter.

If the communication didn't work as expected, the firmware registers on their own have very little useful information. Verifying the signaling answers so many questions:

- Is the behavior of CS working as expected and does it match the requirements of the data flash that the processor is trying to communicate to? Remember it is being controlled entirely by the peripheral.
- Is the clock signal (SCK) working and does the clock speed match what was configured?
- Is the MOSI data being sent correctly?
- Is there any response on the MISO line from the target device?
- Are there any unexplained gaps between clock bursts?

The figure below is the digital signaling for our Blade application. All the above details can be confirmed and the results should also be seen in firmware. The programmed delays between bytes and around CS transitions can also be verified.

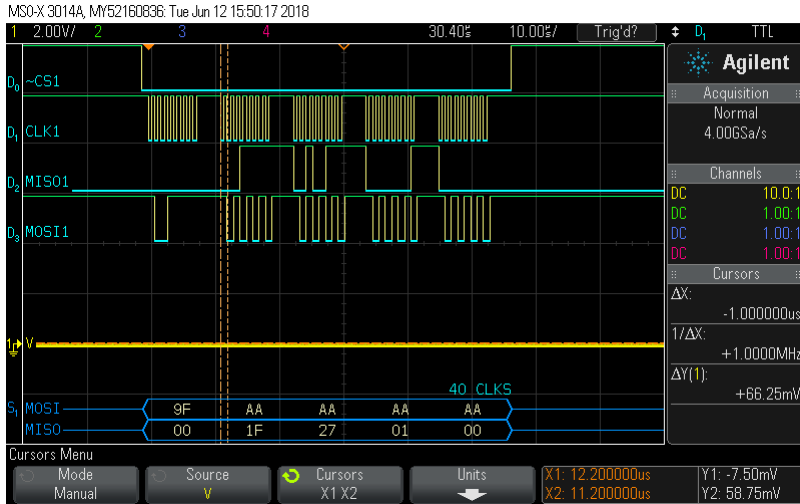


Figure 11-17 Blade application digital signals

Our first cut at the SPI driver didn't work because of the attempt to turn off receiving during transmit. This would not have worked, anyway, with the data flash. Once that was solved and updated, no response was seen on MISO. Troubleshooting this led to swapping the CPHA bit and then everything worked well. Be careful when testing with breakpoints, because peripherals may not respond to breakpoints or single stepping of the main core. Testing new hardware or new low-level drivers plus new application firmware at the same time can be extremely difficult. With the bugs sorted out, our solution can be described.

### 11.17.2 • UserApp1Initialize()

Initialization requires setting up pointers and requesting the SPI resource. The request can be made during initialization because in this simple system we know that our application is the only one interested in the Blade SPI resource. If there was a chance that other tasks needed the resource, then the request should be made during the main loop so it can fail and be retried.

```
void UserApp1Initialize(void)
{
    /* Initialize pointers */
    UserApp1_pu8RxBufferUnreadChar = UserApp1_au8RxBuffer;
    UserApp1_pu8RxBufferNextChar = UserApp1_au8RxBuffer;

    /* Setup the SPI configuration variable and request the SPI peripheral */
    UserApp1_eBladeDataflashSPI.u32CsPin = BLADE_CS_PIN;
    UserApp1_eBladeDataflashSPI.eBitOrder = SPI_MSB_FIRST;
    UserApp1_eBladeDataflashSPI.SpiPeripheral = BLADE_SPI;
    UserApp1_eBladeDataflashSPI.pCsGpioAddress = BLADE_BASE_PORT;
    UserApp1_eBladeDataflashSPI.ppu8RxNextByte = &UserApp1_pu8RxBufferNextChar;
    UserApp1_eBladeDataflashSPI.u16RxBufferSize = U16_UA1_RX_BUFFER_SIZE;
    UserApp1_eBladeDataflashSPI.pu8RxBufferAddress = UserApp1_au8RxBuffer;

    UserApp1_Spi = SpiRequest(&UserApp1_eBladeDataflashSPI);
    BLADE_SPI_FLAGS = 0;
}
```



```
/* If good initialization, set state to Idle */
if( UserApp1_Spi != NULL )
{
    DebugPrintf("Blade dataflash task ready\n\r");
    UserApp1_pfStateMachine = UserApp1SM_Idle;
}
else
{
    /* The task isn't properly initialized, so shut it down and don't run */
    UserApp1_pfStateMachine = UserApp1SM_Error;
}

} /* end UserApp1Initialize() */
```

The CS pin and GPIO information are not needed in this application because all CS operation is managed by the peripheral. The fields are still populated so they are available if the application decides they are needed.

### 11.17.3 • UserApp1SM

Two states will be used: one to wait until BUTTON0 is pressed to queue the necessary transmit message to the SPI peripheral. The second is to wait for the message to be complete before reading the received data.

In the Idle state, check the button. When pressed, ack it, queue the SPI message and change to the wait state. Make sure the parameters to SpiWriteData are correct and make sense to you.

```
static void UserApp1SM_Idle(void)
{
    if( WasButtonPressed(BUTTON0) )
    {
        ButtonAcknowledge(BUTTON0);
        UserApp1_CurrentMsgToken = SpiWriteData(UserApp1_Spi,
                                                sizeof(UserApp1_au8MessageManufacturerID),
                                                UserApp1_au8MessageManufacturerID);
        UserApp1_pfStateMachine = UserApp1SM_WaitResponse;
    }
}

} /* end UserApp1SM_Idle() */
```

The wait state is straightforward in operation, though parsing and printing data requires fighting with ASCII conversions. In a full application, a timeout would be added with appropriate action taken if the message doesn't send. For a Master SPI, only a major issue would prevent the transfer and likely a peripheral reset or even device reset would be necessary to fix it. The chances of this occurring are practically 0 once the firmware is proven during development, so a harsh action like reset isn't as bad as it might seem.

Our solution keeps it simple and expects the transmission to occur. The transmission is actually complete long before the state ever executes since 5 bytes at 1 MHz SCK take less than 60us to transfer. Since every Rx byte is sent to UserApp1\_au8RxBuffer, the application must know what they mean. The first is a dummy, the next 4 should be ID information from the device. The UnreadChar pointer is owned by the application for going through the Rx buffer. It should never be ahead of the incoming character buffer pointer. When the two pointers match, there is no new data in the buffer. This simple application knows exactly how many bytes are present, so it's valid to read exactly that many bytes. Only the first-byte processing is shown in the solution.

```

static void UserApp1SM_WaitResponse(void)
{
    u8 u8ManufacturerId = 0xff;
    u8 u8DeviceId1 = 0xff;
    u8 u8DeviceId2 = 0xff;
    u8 u8Extended = 0xff;
    u8 au8AsciiMsg[] = "FF\n\r";

    if(QueryMessageStatus(UserApp1_CurrentMsgToken) == COMPLETE)
    {
        /* The rx buffer should contain 1 dummy, the Manufacturer ID (0x1F),
        Device ID1 (0x24), Device ID2 (0x00), Extended Info (0x00) */
        UserApp1_pu8RxBufferUnreadChar++;
        u8ManufacturerId = *UserApp1_pu8RxBufferUnreadChar++;
        u8DeviceId1 = *UserApp1_pu8RxBufferUnreadChar++;
        u8DeviceId2 = *UserApp1_pu8RxBufferUnreadChar++;
        u8Extended = *UserApp1_pu8RxBufferUnreadChar++;

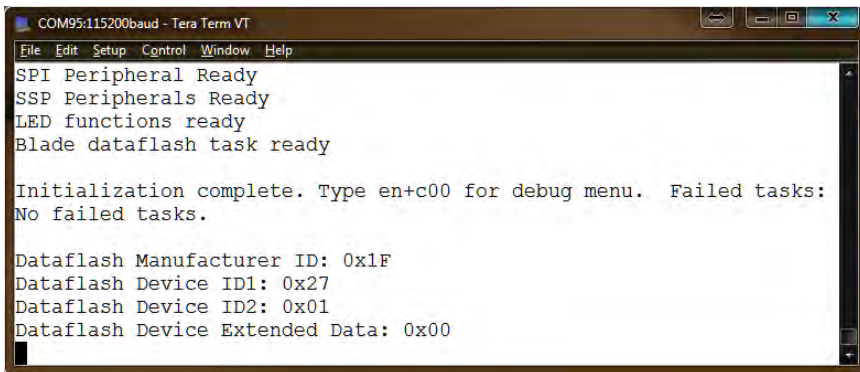
        DebugPrintf("Dataflash Manufacturer ID: 0x");
        au8AsciiMsg[0] = HexToASCIICharUpper( (u8ManufacturerId & 0xF0) >> 4);
        au8AsciiMsg[1] = HexToASCIICharUpper( (u8ManufacturerId & 0x0F) >> 0);
        DebugPrintf(au8AsciiMsg);

        <... process other 3 bytes ...>

        UserApp1_pfStateMachine = UserApp1SM_Idle;
    }
}
/* end UserApp1SM_WaitResponse() */

```

The resulting output on the terminal is shown. We can visually verify that the data received from the data flash is what is expected. Since the device information should not change, this verification could be added to the application. In a real product, this would be an obvious addition and likely part of the automated production and/or device POST (Power On Self-Test).



```

COM95:115200baud - Tera Term VT
File Edit Setup Control Window Help
SPI Peripheral Ready
SSP Peripherals Ready
LED functions ready
Blade dataflash task ready

Initialization complete. Type en+c00 for debug menu. Failed tasks:
No failed tasks.

Dataflash Manufacturer ID: 0x1F
Dataflash Device ID1: 0x27
Dataflash Device ID2: 0x01
Dataflash Device Extended Data: 0x00

```

Figure 11-18 Power On Self-Test (POST)

### 11.18 • Chapter Exercise

At this point in the book, the magnitude of the examples is potentially very large with many choices of what you could choose to work on. If you want to explore the data flash further, continue building the data flash application to add read, write, and erase capability. If you are looking for more practice with SPI, you could write a loop-back

program to practice sending and receiving data or use a second development board to plug in and try different combinations of Master and Slaves running on each board.

To practice more with the Blade connections, try different Blades or make your own using a device that you're interested in. If you have a UART to USB adapter, try making a Blade UART to a computer. Watch the EiE website for a growing list of Blade-related firmware projects.

## Chapter 12 • I<sup>2</sup>C & ASCII LCD

I<sup>2</sup>C (Inter-Integrated Circuit) is a two-wire communication system meant primarily for data transmission within an embedded system between the microcontroller and various other integrated circuits. It is a synchronous, Master-Slave, half-duplex system where the microcontroller is usually the Master and other devices like memories, LCDs, accelerometers, GPS modules, etc. are Slaves. Though multi-Master systems are permissible, they are not common and will not be discussed here. The EiE development board uses I<sup>2</sup>C to transfer data between the microcontroller and the LCD controller.

### 12.1 • Inter-Integrated Circuit (I<sup>2</sup>C) Communication

Up until October 2006, any device that used I<sup>2</sup>C was subject to royalties since the official standard invented by Phillips (now NXP) was licensed intellectual property. Designers wishing to build an I<sup>2</sup>C device and use the I<sup>2</sup>C logo paid a fee which was usually incorporated in the device cost.



Figure 12-1 I<sup>2</sup>C logo

The I<sup>2</sup>C standard also regulated device addresses. Multi-Slave configurations work by using unique addresses with all the Slaves connected on the bus. A Master initiates communication by writing a “Start condition” to the bus and then sends the address of the Slave on the bus it wants to talk to. All devices on the bus see this address, but if the address is not theirs then they ignore the remaining data until a “Stop condition” appears on the bus, at which time they start listening for the address of the next transaction. The protocol started with 7-bit addresses but allows 10-bit addresses now. Still, that is only 1023 unique addresses, and certainly, there are more than that many I<sup>2</sup>C devices. In fact, some vendors might have that many parts themselves!

Device addresses are supposed to be grouped depending on their function. As an arbitrary example, memory devices might be allowed addresses from 0x01 to 0x0F, and accelerometers are allowed 0x10 to 0x1F. The whole concept makes sense until you start trying to manage an exponentially increasing number of device types and devices within those types. Plus, every other manufacturer was coming up with their own implementation of I<sup>2</sup>C that dodged the patent but still allowed compatibility. Lawyers must have eventually got involved and decided to find a way around the licensing requirements because it did not really make sense anymore, and people did not want to pay royalties. In the end, the fee was relaxed in 2006 although a few rules remain.

Nowadays, instead of offering “I<sup>2</sup>C” communications, a simple “2-wire interface” is offered and the addressing specification is quite relaxed (though if you want, you can still pay NXP and get an officially registered address and then probably win in court if anyone’s non-registered device conflicts with yours). Most companies are careful to never say “I<sup>2</sup>C” in their documentation, but even that seems to be a non-issue these days. Atmel uses “TWI” for all references to its I<sup>2</sup>C peripheral.

## 12.2 • I<sup>2</sup>C Hardware

On the hardware side, the single data line (SDA) and single clock line (SCL) are connected to all devices on the bus. Up to 128 devices can be supported and there are two standard system speeds: 100kHz and 400kHz, though essentially any clock speed can be used if it is less than the maximum specified on the device. The one data line is used for both transmit and receive data (thus the half-duplex restriction) and device selection is entirely address-based. That means that regardless of the number of devices on the bus, I<sup>2</sup>C only requires two IO lines to hook everything together, making it one of the lowest hardware overhead protocols available. Both lines require a pull-up resistor to drive the high state, and all devices use open-drain GPIOs to drive the low state. The figure shows an example of a single Master / multi-Slave I<sup>2</sup>C configuration. The Slave addresses must be unique on the bus.

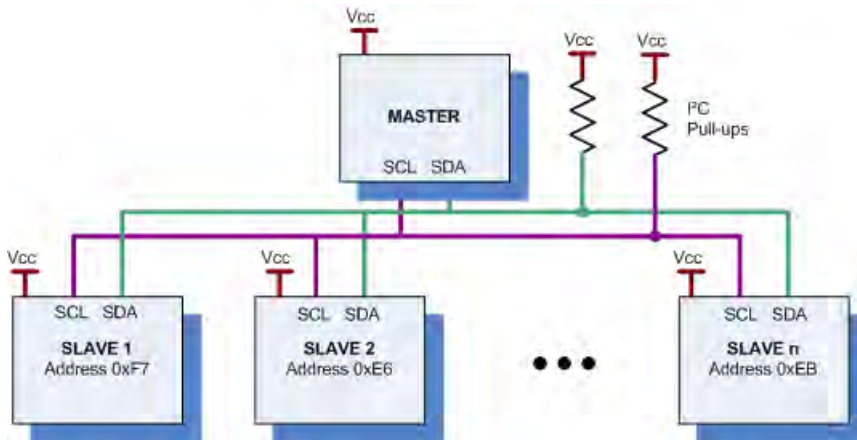


Figure 12-2 Example I<sup>2</sup>C bus configuration

When no communication is occurring we call this the idle state, during which the Master and all the Slaves keep their respective SCL and SDA GPIOs in a high-impedance state. Low signals are driven by the device that has control of the bus lines, but high states are achieved by resetting the driver line to a hi-z state and letting the pull-up resistor do the work.

Because of the open-drain drivers, the designer must pay special attention to the speed of the bus and the capacitance that each device and interconnect adds. There are tradeoffs in selecting the pull-up resistor value to get a good low signal level voltage and to minimize current draw during communication since in the low state, the resistor is between Vcc and Vss. It also impacts the signal response time due to the bus capacitance. If there is one Master and one Slave on the bus, then 10k pull-ups are usually selected and the signal performance is fine. As you add more devices and/or longer bus lengths, the resistor size is reduced perhaps to 4.7k but may be as low as 1k or less to provide a stronger pull-up.

At some point in the design phase, you should put an oscilloscope probe on SCL and observe the clock signal to determine if the edges are fast enough. In extreme cases of very long communication lines, I<sup>2</sup>C repeaters can be added. The Figure 12-3 on page 407 shows a rough example of what the SCL signal might look like with and without proper pull-ups. While you will never achieve a perfect square wave, you should have a reasonably good quality signal.

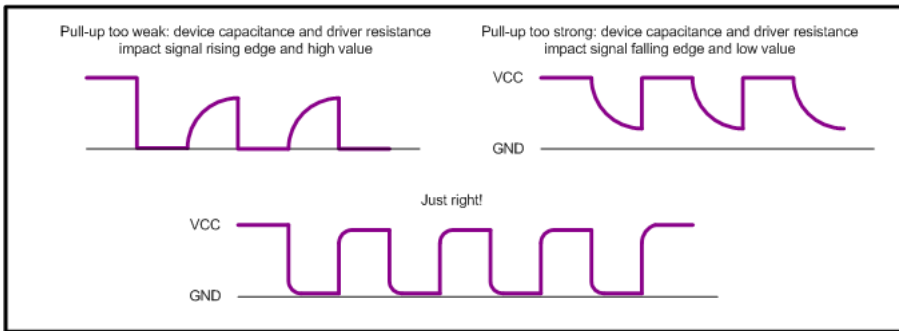


Figure 12-3 SCL signal without proper pull-ups

The figure below is taken from the EiE development board to show the actual waveforms on the board. You can see the clock slew as it rises slowly with the pull-up. The falling edge is sharp because it is being driven to ground. The figure also shows the clock speed we will be using. Just left of the first cursor line on the image is where the Master has released SDA so the Slave can Acknowledge. There is a short delay that causes a small spike, but the Slave quickly takes over holding it low for the ACK. The ACK is sampled on the rising edge of SCL, so there is plenty of margin in time on either side of the bit. Don't worry about the voltage level not being driven as low as the Master drives it. The level is still very safe in the logic 0 realm. The quick spike DOES exceed the logic threshold as you can see in the digital trace of SDA. This means if you were implementing your own I<sup>2</sup>C driver, it would be important to consider this transition if you planned to use edge-triggered interrupts.

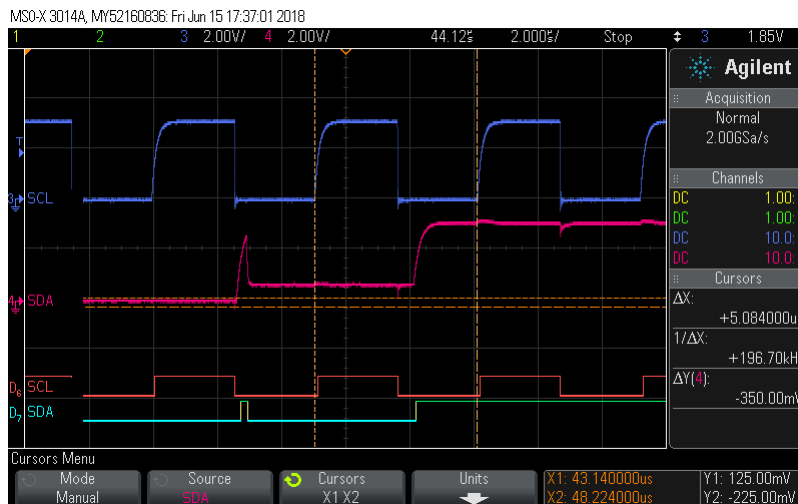


Figure 12-4 EiE development board SCL and SDA signals

Since the protocol is synchronous and the clock is always provided by the Master, the clock does not have to be perfect and can start and stop as needed. In fact, there is a special mode called “clock stretching” that allows communications to pause. This is always allowed by the Slave and Masters can do it but only in multi-Master systems. Allowing the Slave to hold the clock line low is a great feature of the protocol that lets the Slave do basic flow control without adding any additional hardware lines.

Because of the output low / high impedance signaling, there is no risk of hardware problems like shorting high signals to ground. Depending on the Master, there may be a certain maximum time that the Slave can hold the clock line low before the Master decides that the communication has failed and tries to reset the bus and perhaps reset the external device if it has control over power or a reset line to it.

Like with UART-connected devices, an I<sup>2</sup>C system assumes that the receiver is always ready for incoming data. Additional signaling lines could be added like a “data ready” line from a Slave device that could tell the Master to communicate, but most of the time the Master takes care of managing the data connection. The relatively slow clock speed of I<sup>2</sup>C allows ample time for most devices to respond fast enough with an ACK to keep the system going without any delays between bytes.

### 12.3 • I<sup>2</sup>C Signaling

Signaling for I<sup>2</sup>C is quite neat as there is a fair bit of control and handshaking information communicated between devices on the bus even though there are only two physical lines. For this discussion and probably for most of the systems you will work with, 7-bit addresses will be used. This makes it easier to look at since the complete address is contained in a single byte. If 10-bit addresses are required, be sure to set up the I<sup>2</sup>C peripheral accordingly on both the Master and Slave(s) on the bus. Some devices may not support 10-bit addressing, so be careful when specifying parts that will share the same bus.

When no communication is taking place, SDA and SCL are high and the bus is considered idle. To begin communication, the Master initiates a start condition on the bus. A start condition is the act of bringing SDA low while SCL remains high. This transition order is a special case per the protocol and never occurs during data transmission so it can be identified by all devices on the bus.

As a rule of the system, all Slaves on the bus are normally idle and looking for the start condition to occur. The Master then broadcasts the 7-bit address in the top 7 MSBs of the first byte and adds a read (1) or write (0) bit for the 8th bit (the LSB). The Master toggles the clock line for each bit in the address byte and adds an additional clock cycle for a ninth bit. The Master also stops driving SDA so it can float (remember both SCL and SDA are open-drain). This means the Master changes the SDA pin from an output to an input.

When the Slave sees its 7-bit address on the bus plus a read/write bit, it will acknowledge the address by holding the data line low on the 9th clock provided by the Master. As long as one of the Slaves on the bus sees its address, the Master will get an ACK. The Master then knows that the Slave it was trying to address is present and ready. This is illustrated in the figure.

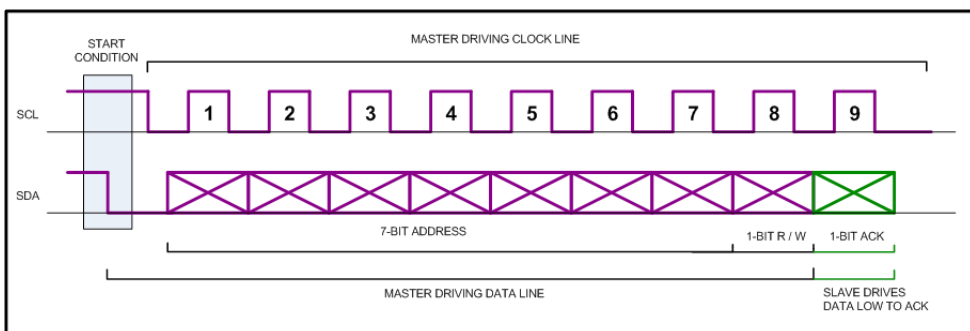


Figure 12-5 One I<sup>2</sup>C message frame

If the Master has sent an address that does not match any of the devices on the bus, then nothing will hold the data line low and therefore no ACK will be seen by the Master. At this time, the Master would set a Stop condition on the bus and then try to figure out why a device that should have responded did not.

There is no provision to handle two devices on the bus with the same address – that is a problem that is up to you as a designer to avoid or mitigate. It is also assumed that the Master knows what device it expects to talk to for every given address it uses to communicate. Though there are ways to “discover” devices on the bus, the most typical applications have known devices at known addresses that the Master uses.

Assuming a device ACKs the address byte, data transfer will begin on the next transmitted byte. All other devices that did not see their address will ignore subsequent communication for the current session. If the Master is transmitting data to the Slave, the Slave is expecting that the next byte on the bus is the first data byte in the transfer from the Master. The Slave ACKs every byte sent by holding SDA low at the end of each byte just like it did when it ACKed its address.

The Master can continue transmitting bytes for as long as it wants and as long as the Slave can handle all the data and keeps ACKing. Once the last byte is sent, the Master puts the stop condition on the bus (SDA goes high while SCL is high) which terminates that communication session.

To receive data from a Slave, the Master puts a start condition on the bus and sends the Slave’s address with a “1” bit as the LSB. Once the address byte is sent and acknowledged by the Slave, the Master assumes the Slave is ready to send data and activates SCL to clock in bytes which the Slave should be sending. The Master will ACK every byte from the Slave during transmission, which indicates to the Slave that it should keep sending data, though it is still up to the Master to provide the clock to enable the Slave to send. When the Master has had enough data, it NAKs the last byte and then puts a stop condition on the bus. The Master has full control over how many bytes the Slave sends. It is still important to follow the datasheet specification to properly interface to a Slave device.

The question you might be thinking about is how does the Slave know what data to send? This is the same problem we see in SPI Master/Slave configurations. The situation depends on what the Slave is, as it will have data available in different ways. One of the most common schemes is setting an address pointer in the Slave by writing a register address prior to reading data, and then clocking out a known number of bytes. Both the Master and Slave know how many bytes will be sent for any given transaction. The Master might also tell the Slave how many bytes it is looking for, which requires the Master to first address the Slave with a write to send it a command or other information to set up the data transfer. In some cases, the Master would terminate the write frame, and then immediately start another frame but this time in read mode. The Slave, having just been told what the Master is looking for, can then expect to provide the data to the Master. The stop and start condition can essentially occur at the same time, which is referred to as a “repeated start.”

To summarize, a complete I<sup>2</sup>C read transaction might proceed like this:

1. Master issues start condition and sends Slave’s address with write bit
2. Slave acknowledges the write address
3. Master transmits the start address that it wants to read data and the number of bytes it wants to read
4. Slave acknowledges each data byte and knows what the purpose of each byte is based on a predefined protocol



5. Master issues a repeated start condition (which is essentially a stop condition and then a start condition)
6. Master transmits the Slave's address with the read bit
7. Slave acknowledges the read address
8. Master clocks in the number of bytes it wants, ACKing each one as it is received
9. Master NAKs the final byte to confirm it is done and puts the stop condition on the bus.



As an exercise, capture this in a flowchart and include a simple drawing of the signals you would expect to see on the SDA and SCL lines.

Typically the Slave is set up to read the register address the Master sends and set a pointer to that location in memory. The Slave will then automatically increment its internal pointer after every byte it sends. In a simpler case, a Slave may only have a single register to output so the same data is sent each time (like an 8-bit temperature sensor). In this case, the Master would only ever have to read data and not worry about specifying an address or sending commands. If the Master tried to read multiple bytes, then it might just keep getting the same information repeatedly, it might get additional useful data, or it might get garbage.

There are lots of scenarios that you can come up with that may need to be handled depending on the system you are working with. Regardless of what you are trying to communicate with, the key is to read the device datasheet to see what it can do and how to do it. You may then write a specific driver set to run with your system to access the Slave, or it might fit in with a more generic I<sup>2</sup>C driver you can write.

If you can understand the above, then implementing I<sup>2</sup>C in firmware is quite straightforward even if you had to bit-bash it. I<sup>2</sup>C peripherals take care of most of the details to implement the protocol. The abstraction is usually quite neat and tidy and essentially any processor you use will have the same set of functions consisting of the following:

1. Peripheral initialization
2. Control functions including `I2CStart()`, `I2CStop()`, `I2CRepeatedStart()`
3. Data functions `ReadByte()` and `WriteByte()`

The peripheral initialization will be completely processor-specific though you will likely find many similarities between different vendor's I<sup>2</sup>C peripherals. The control functions are very simple to implement and usually just set a bit. The Read and Write functions will center around holding registers for the main interface to the peripheral. Even if peripherals in different microcontrollers are implemented very differently, the API can be the same so applications are portable beyond the driver level.

The last feature of this protocol to mention is the reserved addresses in the scheme, of which there are five or six. The one you see the most and will probably use is the "general call" or generic address, 0x00. All devices will listen to the general call address and ACK it and will then receive a single command byte. These special addresses and data bytes along with every other detail about I<sup>2</sup>C can be found on [www.i2c-bus.org](http://www.i2c-bus.org).

I<sup>2</sup>C tends to be a very popular choice for onboard communications in an embedded system due to the addressing capability and simple hardware requirements, though some high throughput devices requiring more speed may choose a different protocol. No matter how many devices you have on the bus, you only ever need two GPIOs on your microcontroller as long as you do not go over some physical resistive and capacitive limits or the physical address size. You can probably make a safe bet that you will implement

I<sup>2</sup>C at some point in your embedded career.

#### 12.4 • EiE TWI Hardware

The SAM3U2 processor has one TWI peripheral. On the ASCII EiE development board, the LCD communication is I<sup>2</sup>C and we must also support the Blade I<sup>2</sup>C connection.

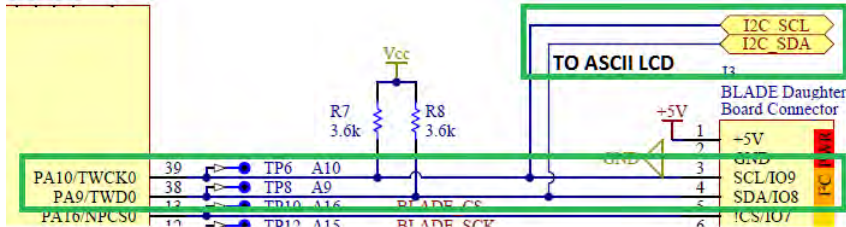


Figure 12-6 SAM3U2 Processor

The pull-ups are set at 3.6k and could always be changed to higher or lower values if required. Since the Blade and LCD share this bus, any non-I<sup>2</sup>C Blade connection using IO8 and IO9 would have to account for the same physical connection. For nearly all practical applications, this would mean the LCD would not be usable. In this case, the processor should hold the LCD in reset so it does not try to operate on the IO signals between the Blade and the SAM3U2.

#### 12.5 • I<sup>2</sup>C (TWI) on SAM3U2

Though there is certainly more involved than with RS-232, I<sup>2</sup>C is still relatively simple and about as difficult to implement as SPI but for different reasons. While most MCUs have I<sup>2</sup>C peripherals now, you still might end up bit-bashing a driver if you are working with some legacy product or if you run out of peripherals on your microcontroller. That becomes a bit challenging and you would need to do some further investigation to understand and implement the full I<sup>2</sup>C protocol.

We will stick to the peripheral for EiE, which is called “TWI0” on the SAM3U2. Due to the significant operational differences in Master and Slave I<sup>2</sup>C, this driver will only support Master mode. As with all new peripherals, reading the user guide is the essential first step to writing a driver. I<sup>2</sup>C peripherals can be drastically different in their implementation across different vendors and they all seem to have their quirks. One could argue that the SAM3U2 TWI documentation has some significant holes, so watch out for the questions that come up as you learn the peripheral.

It’s important to note the table of abbreviations because you will need to interpret the signaling diagrams in the user guide.

Abbreviation	Description
TWI	Two-wire Interface
A	Acknowledge
NA	Non Acknowledge
P	Stop
S	Start
Sr	Repeated Start
SADR	Slave Address
ADR	Any address except SADR
R	Read
W	Write

**Figure 12-7 SAM3U TWI abbreviations**

The TWI block diagram is basic and for some reason doesn't show the PDC which is available for Master transmit and receive. Pin configuration is typical. Remember that the I<sup>2</sup>C clock and data lines are open-drain. The user guide states that by assigning the lines to the peripheral, the open-drain configuration is not necessary, however, we have it set up to explicitly set open-drain mode to avoid confusion with anyone looking at the code. There does not seem to be any issue with that.

As with all the other peripherals we have studied, there is a PMC control line to activate TWI0, and the TWI0 peripheral has a line into the NVIC for the various TWI0 interrupts. The user guide also briefly describes the signaling for I<sup>2</sup>C which you should review to compare with our description to help reinforce the concepts.

The only other section of the user guide you need to read is the Master Mode description. A great exercise is to read the logical guides and create a flow chart based on what the user guide tells you. Do this for both Master Transmit and Master Receive. Then, compare your flow charts to the flow charts that are also provided in the user guide.

There is mention of the "internal address" feature of the peripheral that works with certain Slave memory devices to efficiently add a mechanism to write in a starting data address before transitioning to reading data from the Slave. Since this is a common requirement for many I<sup>2</sup>C devices, having the option as a built-in peripheral feature could be useful. You can accomplish the same result without this feature, so the choice is yours if you find yourself having to make it.

## 12.6 • TWI and PDC

The PDC is critically important and very well integrated on the SAM3U2, so using it is practically essential. Instructions for using the PDC with the TWI are limited to two small procedural lists, one for transmit and one for receive. When writing this driver, we found it very difficult to map the PDC operation to the direct-register implementation described in the rest of the TWI user guide description and flow charts. We had many questions about what was going to happen, and in the end, spent several hours checking operation with our code using an oscilloscope and the debugger to see what actually happens when the PDC is used with the TWI.

One of the main issues was where in the flowchart does the PDC take over the actions of a non-PDC system? The first two clock bursts that happen in a Master transmit are the Slave address and then the first data byte. There is no indication about how or when

these bytes get sent. The only help the user guide provided was that Master Transmit with PDC is initiated by setting the PDC's TXTEN bit which performs the first transmit holding register (THR) load. The first write to THR in the transaction automatically initiates the START condition (both in PDC mode and non-PDC mode). The flow charts fail to include an indication of where the address ACK/NACK from the Slave occurs and what the data flow should be in this case.

The second issue was how to use the PDC with repeated-start requirements or queueing and sending multiple messages where a STOP condition was undesired between them. The EiE system runs into this immediately with the LCD initialization that needs delays and different command and control messages when it's starting up.

It was not even clear if there was any logic in the peripheral that required the explicit setting of Master mode with every transaction. That would be strange, but that was our interpretation of the user guide flowchart. We ignored that and it worked fine. The long story short is that we updated the user guide's flowcharts to remove some clutter and highlight the PDC-specific details that are essential to code a transmit algorithm. This was done only for the multi-byte transmit.



As an exercise, sketch in the PDC involvement for the single byte transmit and the Slave read diagrams to check your understanding of the data flow. When you're finished, compare with the solution shown.

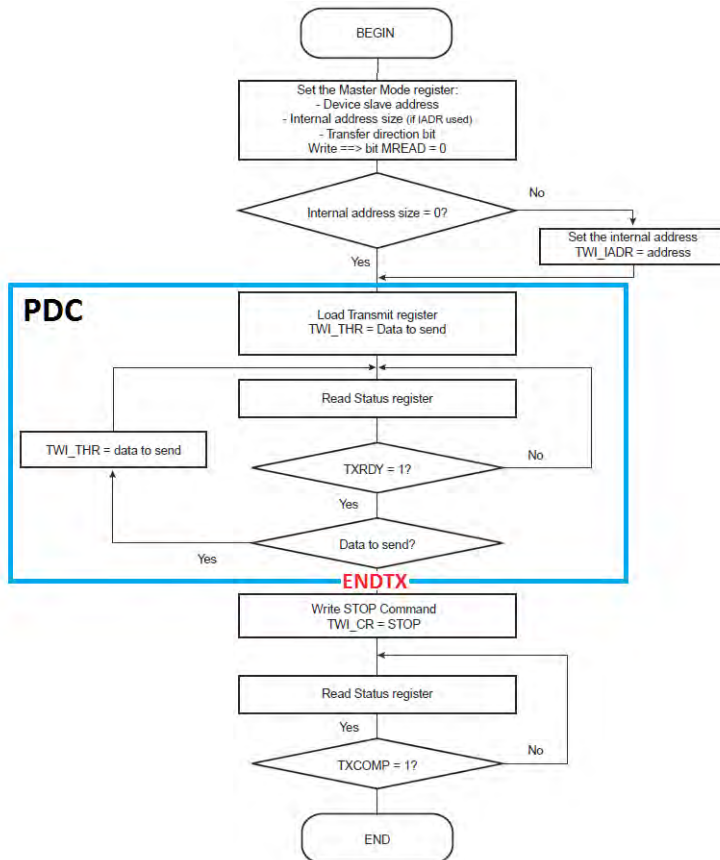


Figure 12-8 Updated multi-byte flowchart

## 12.7 • TWI Registers

The TWI registers are easier to understand than the peripheral operation itself. By now you should be able to read the User Interface register table and have a good idea about what you are going to find in each register. Only four registers require INIT values. If you're using the example exercise code, all of these values are defined in configuration.h already.

The register base address for TWI0 and optional TWI0\_PDC offset is as follows:

```
#define AT91C_BASE_PDC_TWI0 (AT91_CAST(AT91PS_PDC) 0x40084100) // PDC_TWI0 Base
#define AT91C_BASE_TWI0     (AT91_CAST(AT91PS_TWI) 0x40084000) // TWI0 Base
```

**Control Register (TWI\_CR):** If you are surprised by the existence of a Control Register in a peripheral you aren't paying attention. The difference of the TWI\_CR is that it behaves like a SET or ENABLE register and is write-only. As the driver code is written, make sure you change values only with "=" and not "|=" because the OR is a read operation with undefined outcome.

TWI\_CR is used continuously with the TWI driver code to send START and STOP conditions as required by writing the START and STOP bits, respectively. These bits should not be initialized. The register also holds bits to select Master or Slave mode – yes, these do need to be initialized. The EiE development board never repurposes the I<sup>2</sup>C bus, so the mode only needs to be set once. There is a peripheral software reset bit that could be useful if the peripheral gets stuck. This is a not-so-rare occurrence with I<sup>2</sup>C peripherals working with various Slaves. It is not clear if the reset is instantaneous or if other bits can be set at the same time. Therefore we suggest resets should be independent operations with a few milliseconds of delay after.

```
#define EiE_TWI_CR_INIT (u32)0x00000024
```

**Master Mode Register (TWI\_MMR):** MMR holds important information as the bus is configured to operate with a Slave. Most significantly is the Slave address in the DADR bits. This supports standard 7-bit addresses. If more bits are needed, the IADRSZ bits are used in conjunction with the TWI\_IADR register. We have yet to come across a non-7-bit address device... Since MMR depends on the device being addressed on the bus, the initialization value is 0 and left to the driver to program when a transmission begins. This will be taken care of in the API.

```
#define EiE_TWI_MMR_INIT (u32)0x00000000
```

**Slave Mode Register (TWI\_SMR):** The counterpart to MMR, the Slave mode register holds the SAM3U2 I<sup>2</sup>C address if it is a Slave. We will not use this mode, so ignore this register for now.

**Internal Address Register (TWI\_IADR):** This is a special register in the TWI peripheral used for specific applications where the Slave device uses up to 3 bytes following the main Slave address to set an internal address pointer. The expectation is that it will start reporting values from this location on subsequent clocks from the Master. This can be done manually, too, but it can be efficient to build it into the peripheral functionality. It does not require initialization and we will not make use of this in the driver.

**Clock Waveform Generator (TWI\_CWFR):** From the name, it is easy to guess that this sets the Master clock speed. Like with SPI/SSP, this does not apply to Slave devices. Any speed can be selected, but some Slave devices may only be rated to one of the two standard I<sup>2</sup>C speeds (100kHz or 400kHz). For the sake of example, we will calculate and

set the values to set SCK at about 200kHz.

```

/* Clock Wave Generator Register */
/*
  Calculation:
    T_low = ((CLDIV * (2^CKDIV))+4) * T_MCK
    T_high = ((CHDIV * (2^CKDIV))+4) * T_MCK

    T_MCK - period of Master clock = 1/(48 MHz)
    T_low/T_high - period of the low and high signals

    CKDIV = 2, CHDIV and CLDIV = 59
    T_low/T_high = 2.5 microseconds

    Data frequency -
    f = ((T_low + T_high)^-1)
    f = 200000 Hz 0r 200 kHz

  Additional Rates:
    50 kHz - 0x00027777
    100 kHz - 0x00023B3B
    200 kHz - 0x00021D1D
    400 kHz - 0x00030707 *Maximum rate*
*/
#define EiE_TWI_CWGR_INIT (u32)0x00021D1D

```

**Status Register (TWI\_SR):** We know status registers are important for the firmware to access various states and flags of a peripheral. Pay close attention to the conditions that must be present to set and clear these flags. All the flag bits in TWI\_SR are also the interrupt flag bits. TWI\_SR is read-only and does not require initialization.

- **TXCOMP** indicates if a transmission is totally complete (clocked out of THR). It also requires the STOP condition to be set on the bus. This means that multi-message transmissions where the bus does not go idle between messages should not expect this bit to be set until all messages in the sequence are done and a STOP condition is requested.
- **RXRDY** is the typical indicator to show the receive holding register has a new byte. The bit is cleared when the byte is read.
- **TXRDY** is the complementary bit to RXRDY and indicates that the transmit holding register is empty. An important note is that this bit reflects the state of MSEN (the Master enable bit in TWI\_CR). When debugging, make sure you see this bit go high during the TWI peripheral initialization to know that Master mode is enabled.
- **NACK** is the only other bit worth mentioning. This is the flag if a Slave does not acknowledge a byte and will trigger the NACK interrupt.

**Interrupt Enable / Disable / Mask Registers (TWI\_IER / TWI\_IDR / TWI\_IMR):** No surprises here. Since we are only writing a Master driver, and only using PDC transfers with interrupts, only ENDRX, ENDTX are needed for communication but not until a message is queued. NACK will also be enabled for interrupts and will be done so from the start so we don't have to worry about it. During development, this will be a good way to make sure any NACKs are detected as soon as they occur which can help in debugging.

```

#define EiE_TWI_IER_INIT (u32)0x00000100

```

**Receive Holding Register / Transmit Holding Register (TWI\_RHR / TWI\_THR):** the current bytes for receive and transmit, respectively. The usual conditions apply for overrun and underrun. The TWI\_PDC interacts directly with these registers when operating.

## 12.8 • TWI Driver

The API functions for the TWI driver are going to be very similar to both the UART and SPI driver. However, we decided on a fundamental difference to take out the Request and Release functions that limit the use of the peripheral to one task at a time.

Instead, we will introduce a local message queue that works together with the Message task. The circular local queue will hold TWI-specific information about transmit and receive messages. This will work very well since sharing an I<sup>2</sup>C bus with a single local Master and any number of Slaves requires no operational difference beyond transmitting the desired Slave address after the START condition. The I<sup>2</sup>C protocol takes care of everything else. I<sup>2</sup>C is inherently a multi-user system.

Any task will be able to add a read or write request to the TWI without having any knowledge of other tasks using the bus or any other Slaves on the bus. There still exists the problem that two tasks may be interested in using the same Slave which could result in some strange behavior. For example, what if two tasks wanted to use an I<sup>2</sup>C data flash? One task might queue up a write address, but then another task might change that address. The LCD is obvious if one task is trying to write a message when another task is trying to use the same space. However, trying to deal with this at the TWI driver level wouldn't solve a lot of the problems, anyway. Sharing resources like this would need to be handled at the application level.

The driver will work with a local message queue to save all relevant information about any task message including the Slave address, STOP condition behavior and Tx or Rx buffers. Transmit messages will be synchronized with the Message task token. The main difference between the TWI system and the UART/SSP is the connection to a receive buffer. Instead of a single receive buffer assigned to the peripheral object when it is requested, each Rx message in the local TWI queue keeps the Rx buffer information. The high-level view looks something like this:

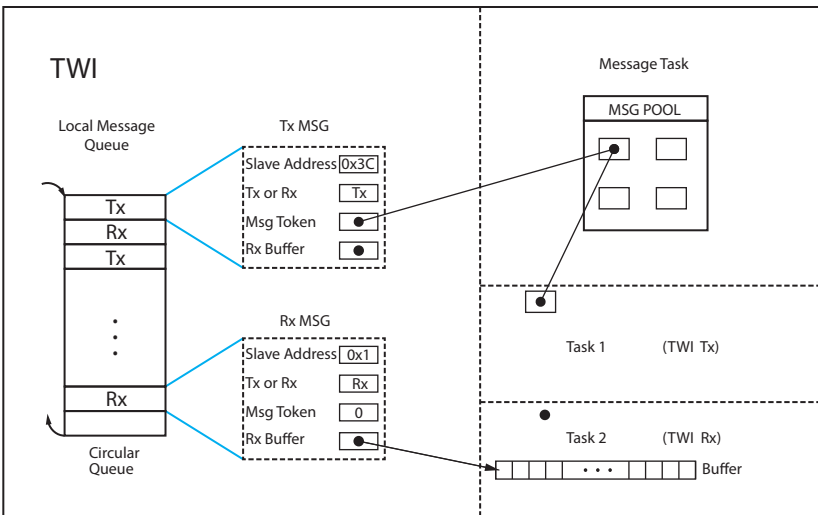


Figure 12-9 TWI local message queue and other task interactions

### 12.8.1 • TWI Data Structures

Before getting any deeper into coding, let's look at the data structures for the TWI driver starting with the simple TWI-specific types that will help in self-documenting the code.

Master transmit functions must specify whether a STOP condition should be placed on the bus after a message is sent. There are many factors in this and it falls on the task using the TWI driver to correctly specify it. This could easily be abstracted in a higher-level API to offload the burden from a typical programmer trying to use an I2C peripheral device. For this driver, making the choices clear while hiding the implementation details is a good outcome.

```
/*!
@enum TwiStopType
@brief Type of behavior for STOP condition after message.
*/
typedef enum {TWI_STOP, TWI_NO_STOP, TWI_NA} TwiStopType;
```

Messages in the local message queue must be flagged as read or write. Since the LSB of the address byte that is placed on the bus is the read/write flag bit, having this information separate is arguably redundant than if it was merged into the address. The EiE driver uses it for clarity both in using the driver and debugging.

```
/*!
@enum TwiDirectionType
@brief Controlled list to specify data transfer bit order.
*/
typedef enum {TWI_EMPTY, TWI_WRITE, TWI_READ} TwiDirectionType;
```

The peripheral configuration type for TWI contains much less information than for the UART and SPI. There are only three members that are general to the TWI object. There are still private flags used for control functions as the driver executes.

```
/*!
@struct TwiPeripheralType
@brief User-defined TWI configuration information
*/
typedef struct
{
    AT91PS_TWI pBaseAddress;      /* Base address of peripheral */
    MessageType* pTransmitBuffer; /* Transmit message struct linked list */
    u32 u32PrivateFlags;          /* Private peripheral flags */
} TwiPeripheralType;

/* u32PrivateFlags definitions in TwiPeripheralType */
#define _TWI_TRANSMITTING (u32)0x00000001 /* Peripheral is Transmitting */
#define _TWI_RECEIVING (u32)0x00000002 /* Peripheral is Receiving */
#define _TWI_TRANS_NOT_COMP (u32)0x00000004 /* Tx hasn't been completed */

#define _TWI_ERROR_TX_MSG_SYNC (u32)0x01000000 /* Local token != queued token */
/* end u32PrivateFlags */
```

Everything else needed to transmit and receive messages will be in the local message queue. A single object type is created even though some members apply only to Tx or Rx. Both transmit and receive messages need to track the target Slave address and indicate if the queued node is a transmit or receive operation so the TWI task executes the right function.



```

/*!
@struct TwiMessageQueueType
@brief Message-specific information
*/
typedef struct
{
    u8 u8Address;                /* Slave address */
    u8 u8Pad;
    TwiDirectionType eDirection; /* Tx/Rx Message Type */
    TwiStopType eStopType;        /* TX ONLY: STOP condition behavior */
    u32 u32MessageTaskToken;      /* TX ONLY: Token in msg task */
    u32 u32Size;                  /* RX ONLY: Size of the transfer */
    u8* pu8RxBuffer;              /* RX ONLY: Target receive buffer */
} TwiMessageQueueType;

```

From the user guide and knowledge of how I<sup>2</sup>C works, we know that Master Transmit messages need to define the STOP condition behavior after data is sent. For the EIE system, the local queue must also link to the Message Task. System tasks can be queuing transmit messages which will add Message Task objects to the peripheral's transmit buffer but will also add a node in the local TWI message queue. Both are FIFOs. When a Transmit message is up for processing in the local queue, the Message Task message object will be at the front of the peripheral's transmit buffer. It is obviously prudent to confirm that the message tokens match, hence the redundancy of the token member in both nodes.

Master receive operations do not require Message Task objects. The Rx-only members of `TwiMessageQueueType` include the size of the transfer since I<sup>2</sup>C is still a synchronous, Master-Slave protocol and the Master must know how many bytes it's clocking. The most important member is the pointer to the Rx buffer of the task that requested the receive operation.

For local globals in `sam3u_i2c.c`, define the peripheral object and the local message buffer. The driver will add messages to the buffer with a "Next" pointer, and it will process messages with a "Current" pointer. A count of how many messages are queued is kept as an easy way to manage the size and message queuing.

```

static TwiMessageQueueType TWI_asMessageBuffer[U8_TWI_MSG_BUFFER_SIZE];
static TwiMessageQueueType* TWI_psMsgBufferNext;
static TwiMessageQueueType* TWI_psMsgBufferCurrent;
static u8 TWI_u8MsgQueueCount;

```

### 12.8.2 • TWI Driver Functions

There are only a few protected functions for the TWI task and they mirror the other communications peripheral drivers closely.

#### **TwiInitialize()**



TWI initialization closely resembles the dedicated-SPI driver code since there is only one TWI peripheral on the processor. It is also simplified since only the Master driver is being written. As always, enable the peripheral in the PMC and set up the local globals.

```

void TwiInitialize(void)
{
    /* Enable the peripheral */
    AT91C_BASE_PMC->PMC_PCER |= (1 << AT91C_ID_TWI0);

    /* Init flags, pointers and globals */
}

```

```

TWI_u32Flags = 0;
TWI_psMsgBufferNext = TWI_asMessageBuffer;
TWI_psMsgBufferCurrent = TWI_asMessageBuffer;
TWI_u8MsgQueueCount = 0;

```

Writing known values to the local message buffer isn't completely necessary since a counter is being used, however, it is always good practice to write known, safe values. This can also be extremely helpful for debugging since you will have a mini-history of all the most recent messages that have been queued and sent. If tasks are overwriting initialized values, it is much easier to pick out errors either from the driver or from the task using the driver if debugging those tasks reaches this level of inspection.

```

/* Clear the local message buffer */
for(u8 i = 0; i < U8_TWI_MSG_BUFFER_SIZE; i++)
{
    TWI_asMessageBuffer[i].eDirection = TWI_EMPTY;
    TWI_asMessageBuffer[i].eStopType = TWI_NA;
    TWI_asMessageBuffer[i].pu8RxBuffer = NULL ;
    TWI_asMessageBuffer[i].u32MessageTaskToken = 0;
    TWI_asMessageBuffer[i].u32Size = 0;
    TWI_asMessageBuffer[i].u8Address = 0;
}

```

The peripheral object values are then initialized and a manual reset is performed. We have used some strange I<sup>2</sup>C peripherals in the past, and a common theme that emerged is that making use of the peripheral reset feature is a good practice when starting it up. Since this runs during system initialization, it is fine to block while waiting for the reset.

```

/* Initialize the TWI peripheral structures */
TWI_Peripheral0.pBaseAddress = AT91C_BASE_TWI0;
TWI_Peripheral0.pTransmitBuffer = NULL;
TWI_Peripheral0.u32PrivateFlags = 0;

```

The function concludes by loading the peripheral registers with the INIT values that were established. It is safe to enable the NVIC TWI input at this point.

```

/* Configure Peripheral for Master mode */
TWI_Peripheral0.pBaseAddress->TWI_CWGR = TWI0_CWGR_INIT;
TWI_Peripheral0.pBaseAddress->TWI_CR = TWI0_CR_INIT;
TWI_Peripheral0.pBaseAddress->TWI_MMR = TWI0_MMR_INIT;
TWI_Peripheral0.pBaseAddress->TWI_IER = TWI0_IER_INIT;
TWI_Peripheral0.pBaseAddress->TWI_IDR = ~TWI0_IER_INIT;

/* Enable TWI interrupts */
NVIC_ClearPendingIRQ( (IRQn_Type)AT91C_ID_TWI0 );
NVIC_EnableIRQ( (IRQn_Type)AT91C_ID_TWI0 );

/* Set application pointer */
TWI_pfnStateMachine = TwiSM_Idle;
} /* end TwiInitialize() */

```

TwiRunActiveState() and TwiManualMode() are already in the example code since there is nothing unique about them. Manual mode should include the TWI task, Message task, and the Debug task.

The TWI interrupt handler is also in the protected section but it is better to discuss this in the same context as the overall state machine. Let's look at the public API functions

first as that will help drive home some I<sup>2</sup>C concepts and gives a nice segue into the state machine.

### **TwiWriteData()**

In designing the API and looking at what must happen for I<sup>2</sup>C messaging, a design decision was to only offer WriteData() and ReadData() functions instead of including a WriteByte() and ReadByte() simpler version for single-byte data. The byte versions were included in the UART and SSP/SPI drivers because single-byte messages are common and the code required to implement the functions is simple. The motivation behind the single byte functions is that they save passing an extra parameter since “size” is not required. This can potentially add up to substantial flash and processing savings considering all the tasks that might use the API.

The TWI case is different from the start because every message is inherently more complicated because at least two bytes are always involved (the address and the data byte). Though neither the task nor the driver code explicitly writes the address byte, the TWI flowchart incorporates it and the driver must implement the expected behavior. That means both the Write and Read API functions are much longer, so there is little chance of ultimately saving processor resources by offering a function with one less parameter for single byte transfers.



It is highly recommended to write your own version of TwiWriteData. This will force you to study the peripheral description carefully and is a good test of what you have learned from the previous two communications peripherals. The header information is shown here to get you started.

```

/*!-----
@fn u32 TwiWriteData(u8 u8SlaveAddress_, u32 u32Size_,
                    u8* u8Data_, TwiStopType eSend_)

@brief Queues a WRITE message for transfer on the TWI0 peripheral.

Requires:
- TWI peripheral initialized

@param u8SlaveAddress_ holds the target's I2C address
@param u32Size_ is the number of bytes of data to send
@param u8Data_ points to u32Size bytes of data
@param eSend_ is the type of STOP condition to be placed on the bus after transmis-
sion

Promises:
- if the queue is not full:
  - adds a WRITE message to TWI_asMessageBuffer and increments TWI_u8MsgQueueCount
  - a Message task node is added to TWI_Peripheral0.pTransmitBuffer buffer
  - TWI_psMsgBufferNext is advanced and wrapped if necessary
  - returns the message token assigned to the message;
- otherwise 0 is returned if the message cannot be queued in which case
  G_u32MessagingFlags can be checked for the reason
*/

```

Our implementation starts by checking if the local message queue is full, then tries to queue the data to the Message task. If either does not work, the function returns 0. There is no apparent value in distinguishing between which operation did not work. Debug messages or other flags could be added if this was a concern.

```

u32 TwiWriteData(u8 u8SlaveAddress_, u32 u32Size_, u8* u8Data_, TwiStopType eStop_)
{
    u32 u32Token;

    if(TWI_u8MsgQueueCount == U8_TWI_MSG_BUFFER_SIZE)
    {
        /* TWI Message Task Queue Full or the Tx transmit isn't complete */
        return 0;
    }

    /* Queue Message in message system */
    u32Token = QueueMessage(&TWI_Peripheral0.pTransmitBuffer, u32Size_, u8Data_);
    if(u32Token == 0)
    {
        /* TWI Message Task Queue Full or the Tx transmit isn't complete */
        return 0;
    }
}

```

Next, the local message buffer members at `TWI_psMsgBufferNext` are updated with the relevant information for the write operation. The buffer and Next pointer will be accessed in the TWI interrupt, so this section is critical and needs interrupts disabled.

```

/* Critical section: TWI buffer management must be done with interrupts off since
an ISR can also manage the buffer values and pointers */
__disable_irq();

/* Queue Relevant data for TWI register setup */
TWI_psMsgBufferNext->u32MessageTaskToken = u32Token;
TWI_psMsgBufferNext->eDirection = TWI_WRITE;
TWI_psMsgBufferNext->u32Size = u32Size_;
TWI_psMsgBufferNext->u8Address = u8SlaveAddress_;
TWI_psMsgBufferNext->eStopType = eStop_;

/* Not used by Transmit */
TWI_psMsgBufferNext->pu8RxBuffer = NULL;

```

Once the information is ready, the buffer count and the Next pointer can be safely advanced and the critical section is over.

```

/* Update array pointers and size */
TWI_u8MsgQueueCount++;
TWI_psMsgBufferNext++;
if( TWI_psMsgBufferNext == &TWI_asMessageBuffer[U8_TWI_MSG_BUFFER_SIZE] )
{
    TWI_psMsgBufferNext = &TWI_asMessageBuffer[0];
}

/* End of critical section */
__enable_irq();

```

The function ends by checking if it has been called during initialization in which case Manual Mode is executed which will fully send the message. In regular operation, the message will sit in the local queue until it is picked up in the Idle state.

```

/* During init, manually cycle the TWI task to send the message */
if(G_u32SystemFlags & _SYSTEM_INITIALIZING)
{
    TwiManualMode();
}

```

```

    return(u32Token);
} /* end TwiWriteData() */

```

### TwiReadData()

Queuing a read operation is nearly identical to the Write operation. The client task must provide a receive buffer long enough to fit the number of bytes requested. The only check is that the maximum number of queued messages does not exceed the limit.

```

bool TwiReadData(u8 u8SlaveAddress_, u8* pu8RxBuffer_, u32 u32Size_)
{
    if(TWI_u8MsgQueueCount == U8_TWI_MSG_BUFFER_SIZE)
    {
        /* TWI Message Task Queue Full or the Tx transmit isn't complete */
        return FALSE;
    }
}

```

Like the Write case, interrupts must be disabled while the Next pointer is updated with the read information. There is no STOP type (TWI\_NA) and the message token is 0 since there is no associated Message task message.

```

/* Critical section: TWI buffer management must be done with interrupts off since
an ISR can also manage the buffer values and pointers */
__disable_irq();

/* Queue Relevant data for TWI register setup */
TWI_psMsgBufferNext->eDirection = TWI_READ;
TWI_psMsgBufferNext->u32Size = u32Size_;
TWI_psMsgBufferNext->u8Address = u8SlaveAddress_;
TWI_psMsgBufferNext->pu8RxBuffer = pu8RxBuffer_;

/* Stop condition type and message token do not apply for Rx */
TWI_psMsgBufferNext->eStopType = TWI_NA;
TWI_psMsgBufferNext->u32MessageTaskToken = 0;

/* Update array indexers and size */
TWI_u8MsgQueueCount++;
TWI_psMsgBufferNext++;
if( TWI_psMsgBufferNext == &TWI_asMessageBuffer[U8_TWI_MSG_BUFFER_SIZE] )
{
    TWI_psMsgBufferNext = &TWI_asMessageBuffer[0];
}

/* End of critical section */
__enable_irq();

```

The function exits through the Manual Mode check and returns TRUE. The local message buffer dictates all messaging in the TWI task, so any transmit or received message queued before will transfer first.

You are correct if you point out that there is no mechanism to check the status of Read messages. The Messaging task still provides transmit message status, but unlike the SPI driver, a receive check has been excluded. This functionality could easily be added, but due to the nature of TWI applications, it didn't seem necessary. Feel free to code it in as an exercise!

## 12.9 • TWI State Machine and ISR

The most significant part of the TWI driver is the main operation of sending and receiving messages. Since only the TWI Master will be written, the amount of work and overall code complexity is reduced. This is also easier than the other communications drivers because the bus is strictly half-duplex with no chance of having to process bytes coming in while bytes are going out or vice-versa. However, due to the nature of I<sup>2</sup>C operation, the state machine to run transmit and receive functions require a few more steps.

The requirement to manage ACKs/NACKs with every byte along with the integration of the START and STOP conditions is what makes I<sup>2</sup>C more difficult although arguably more reliable. Complex NACK handling routines could be coded at the driver level or pushed up to the application. The reality of an I<sup>2</sup>C-based embedded system is that in many cases the interface is between the MCU and an IC local to the same PCB. This means signal integrity is high and any issues should be sorted out during hardware design. Since the Slave devices are dedicated parts like sensors or memories, they will operate essentially flawlessly if what's coming out of the MCU properly meets the datasheet requirements of the IC. Implementing and testing this takes place during product design and only operating conditions will vary once the product is in the field. All of this reduces the probability of errors to a very low level and suggests that complicated error handling algorithms in firmware may not be necessary.

What we're absolutely not saying is that you can assume there will be no errors and you don't have to worry about it. What we are saying is that for the EiE development system where multiple different ICs could be connected via the Blade, we are not going to try to write any significant error handling system. The way an application is designed and coded must depend on the usual considerations of severity and likelihood of occurrence of any possible fault. If an LCD message on your home-built music player doesn't update now and then who cares. If you're putting a data logger on a satellite with I<sup>2</sup>C sensors and memories, you should probably have a pretty good system for handling NACKs.

### 12.9.1 • TWI Transmit

Starting with Master transmit is the most intuitive although slightly more difficult case. The design follows the transmit flowchart very closely. The most challenging part is filling in the missing information about what exactly happens or needs to happen with respect to the PDC. The user guide makes a point of distinguishing between single-byte and multi-byte transfers, so the Transmit state machine must capture this, also.



Start by sketching a flow chart/state diagram of how the system will work with the details about START and STOP conditions, PDC operations, and peripheral flag actions.

Remember that when a transmit operation starts (by writing the THR register either directly or via PDC), the START condition is done automatically and then the address byte is sent. The data byte that initiated the transfer will sit in THR until the address byte goes out and is ACKed. After that, the data gets moved to the shift register and clocked out.

Keep this in mind if you are debugging the system because seeing two bytes go out (the address and the data byte that was loaded) can catch you off guard. This should be the only instance where loading one-byte results in two bytes being transmitted! The address byte transmission and subsequent data byte are all handled in hardware and there does not seem to be any mechanism on this processor to control what or how it happens. For single-byte transfers, the STOP condition needs to be set right away so that the peripheral is prepared when the byte finishes clocking out.

Pay attention when working with the TWI peripheral registers. CR and IER are both set-style registers and should be written using the assignment operator with bits to turn on (0s are ignored).

```
TWI_Peripheral0.pBaseAddress->TWI_CR |= AT91C_TWI_STOP; // WRONG!
TWI_Peripheral0.pBaseAddress->TWI_CR = AT91C_TWI_STOP; // CORRECT!
```

MMR is a read/write register, so loading the Slave address should be done with the bit-wise OR operator AFTER the bits of interest are cleared. Clearing the bits can be done by masking them out in the destination register first and then ORing in the new bits, or by making a copy of the register, clearing the bits and loading the new ones, then writing the updated value back. Either way, this ensures that bits that are supposed to be set stay set.

```
TWI_MMR = (u8Address << TWI_MMR_ADDRESS_SHIFT); // WRONG!
TWI_MMR &= ~(0x7F << TWI_MMR_ADDRESS_SHIFT); // CORRECT part 1
TWI_MMR |= (u8Address << TWI_MMR_ADDRESS_SHIFT); // CORRECT part 2
```

This was all covered very early in this book, but it's a good time for a reminder. The above lines of code are fundamentally different in behavior and outcome. Using the wrong operator is an especially dangerous bug because it can result in correct behavior many times but is ultimately wrong. Many devices running for a long time will lead to periodic, non-repeatable, nearly impossible to find problems. These types of bugs are notoriously difficult to solve.

When you have finished your flowchart design, compare it to the design below. This is a combination flow chart and state machine. See Figure 12-10 on page 426

Our code solution matches the design above exactly, but that's because we updated the final design when everything was tested and running properly! Our initial design was a basic sketch that included the major details. We certainly don't encourage wasting time doing professional-looking drawings for work-in-progress designs. You should have enough that you can use as a foundation and guide for writing code. Some changes are inevitable, and the code will have more details that you might want to add back into the design drawing. If it turns out the code doesn't match the design, be sure to update or destroy the original design so you don't have conflicting information.

We can write the transmit state machine by following through our design. Once the token is checked and the message status has changed, the TWI peripheral and PDC are set up. The last step in Idle is to set TXTEN which is what triggers the start of the transmission including the START condition. Once transmit has started, the state machine just needs to wait until all the data has been sent. Single-byte and multi-byte transfers work the same at this point.

```
static void TwiSM_Idle(void)
{
    u32 u32Byte;

    /* Do nothing unless new Tx or Rx messages have been queued */
    if(TWI_u8MsgQueueCount != 0)
    {
        if(TWI_psMsgBufferCurrent->eDirection == TWI_WRITE)
        {
            /* Check that the local buffer Message token matches the message queued
            and the transmit buffer */
            if(TWI_psMsgBufferCurrent->u32MessageTaskToken !=
                TWI_Peripheral0.pTransmitBuffer->u32Token)
            {
                TWI_Peripheral0.u32PrivateFlags |= _TWI_FLAG_TX_MSG_SYNC;
                TWI_u32Flags |= _TWI_ERROR_TX_MSG_SYNC;
                TWI_pfnStateMachine = TwiSM_Error;
            }
        }
    }
}
```

```

else
{
    /* Update the message's status */
    UpdateMessageStatus(TWI_Peripheral0.pTransmitBuffer->u32Token, SENDING);

    /* Set up to transmit the message */
    TWI_Peripheral0.u32PrivateFlags |=
        (_TWI_TRANSMITTING | _TWI_TRANS_NOT_COMP);
    u32Byte = (TWI_psMsgBufferCurrent->u8Address) << TWI_MMR_ADDRESS_SHIFT;
    TWI_Peripheral0.pBaseAddress->TWI_MMR |= u32Byte;

    /* Setup PDC and interrupts */
    TWI_Peripheral0.pBaseAddress->TWI_TPR =
        (u32)TWI_Peripheral0.pTransmitBuffer->pu8Message;
    TWI_Peripheral0.pBaseAddress->TWI_TCR =
        TWI_Peripheral0.pTransmitBuffer->u32Size;

    /* Enable Tx interrupt and the transmitter (triggers THR load) */
    TWI_Peripheral0.pBaseAddress->TWI_IER = AT91C_TWI_ENDTX;
    TWI_Peripheral0.pBaseAddress->TWI_PTCR = AT91C_PDC_TXTEN;

    /* Single byte transfers need STOP immediately (if applicable) */
    if(TWI_Peripheral0.pTransmitBuffer->u32Size == 1)
    {
        /* Set up the stop condition immediately if applicable */
        if(TWI_psMsgBufferCurrent->eStopType == TWI_STOP)
        {
            TWI_Peripheral0.pBaseAddress->TWI_CR = AT91C_TWI_STOP;
        }
    }

    TWI_pfnStateMachine = TwiSM_Transmit;
}

```

When the PDC has loaded that last byte, ENDTX will interrupt and the ISR should shut down the ENDTX interrupt and disable the PDC transmitter (but not the TWI peripheral). Remember, ENDTX means the PDC has loaded THR with the last byte that it is responsible for. The data must still be shifted out so there is a bit of time. Even at a slow 100kHz I2C clock, this will occur in under 100us so the ISR must be used to prepare the STOP condition since waiting for the next call of the TWI state machine would often be too late. The ENDTX interrupt code is shown here.

```

/** ENDTX (transmit has finished) */
if(u32InterruptStatus & AT91C_TWI_ENDTX )
{
    /* Disable interrupt and PDC transfer */
    TWI_Peripheral0.pBaseAddress->TWI_IDR = AT91C_TWI_ENDTX;
    TWI_Peripheral0.pBaseAddress->TWI_PTCR = AT91C_PDC_TXTDIS;

    /* Set stop condition if multi-byte transfer */
    if( (TWI_Peripheral0.pTransmitBuffer->u32Size != 1) &&
        (TWI_psMsgBufferCurrent->eStopType == TWI_STOP) )
    {
        TWI_Peripheral0.pBaseAddress->TWI_CR = AT91C_TWI_STOP;
    }

    TWI_Peripheral0.u32PrivateFlags &= ~_TWI_TRANSMITTING;
} /* end ENDTX handler */

```



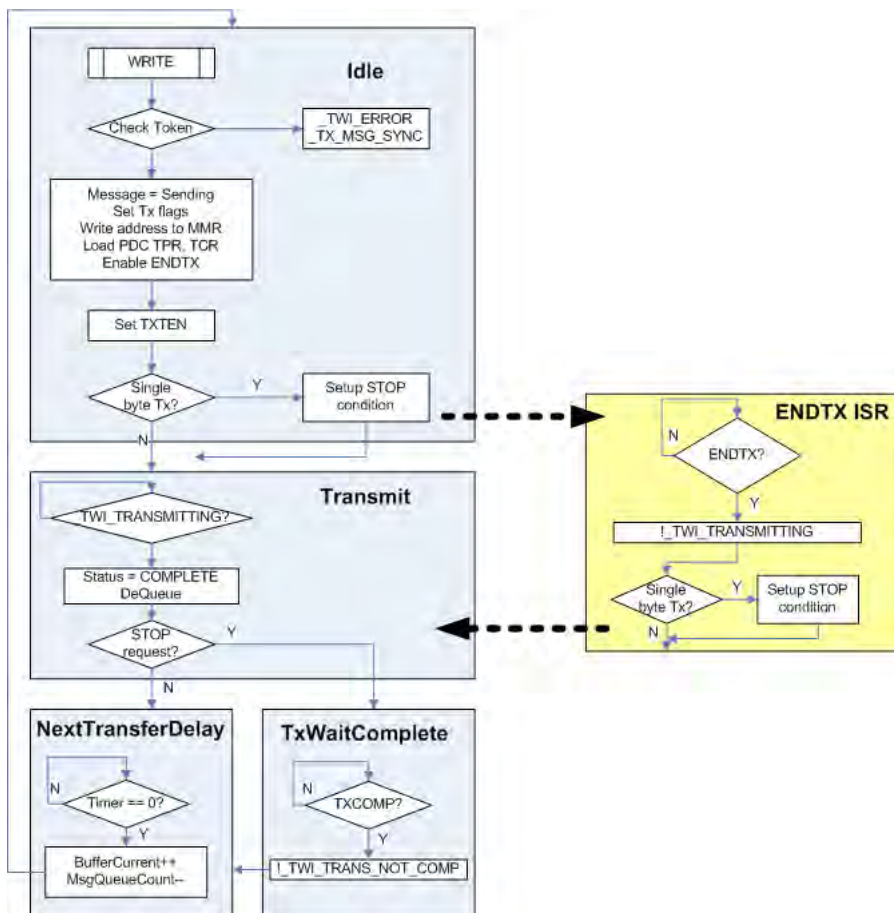


Figure 12-10 Flowchart and state diagram

The EiE design uses a flag bit `_TWI_TRANSMITTING` to control the state machine progression while it is waiting for transmit to occur. The ISR clears this flag. Since the STOP condition is already set up in the ISR (or when the transaction first started for single-bytes), the next operation of the TWI peripheral is not time-dependent and can be handled by the regular state machine.

```

static void TwiSM_Transmit(void)
{
    /* Watch _TWI_TRANSMITTING to indicate transmit is complete */
    if( !(TWI_Peripheral0.u32PrivateFlags & _TWI_TRANSMITTING) )
    {
        /* Clean up the Message task message */
        UpdateMessageStatus(TWI_Peripheral0.pTransmitBuffer->u32Token, COMPLETE);
        DeQueueMessage(&TWI_Peripheral0.pTransmitBuffer);

        /* Advance states depending on whether TXCOMP is expected */
        if(TWI_psMsgBufferCurrent->eStopType == TWI_STOP)
        {
            /* If a STOP condition is requested, need to wait for TXCOMP */
            TWI_pfnStateMachine = TwiSM_TxWaitComplete;
        }
    }
}
    
```

```

    else
    {
        /* Otherwise leave the bus active */
        TWI_u32Timer = U8_NEXT_TRANSFER_DELAY_MS;
        TWI_pfnStateMachine = TwiSM_NextTransferDelay;
    }
}
} /* end TwiSM_Transmit() */

```

The system should confirm when the last byte has been shifted out which is flagged when TXCOMP gets set. Though this should happen within microseconds, a Slave could be clock-stretching or broken so this still needs to be in a state that confirms it has completed. Robust systems should have a timeout and appropriate mitigation.

```

static void TwiSM_TxWaitComplete(void)
{
    /* Wait for TX to complete */
    if(TWI_Peripheral0.pBaseAddress->TWI_SR & AT91C_TWI_TXCOMP_Master)
    {
        /* Clear flags and advance states */
        TWI_Peripheral0.u32PrivateFlags &= ~_TWI_TRANS_NOT_COMP;

        TWI_u32Timer = U8_NEXT_TRANSFER_DELAY_MS;
        TWI_pfnStateMachine = TwiSM_NextTransferDelay;
    }
}

} /* end TwiSM_TxWaitComplete() */

```

The last state is common to transmit and receive. The message status and transmit flags are updated, and the final clean-up of the message buffer is performed after a short delay. NextTransferDelay updates the buffer pointers but is a non-critical section because the state machine ensures that no TWI-interrupts can be occurring at this time. No new transfers are done and no other tasks can run at this point since the executing code is part of the TWI task.

However, there is a potential danger if other systems or tasks queue TWI messages during interrupt service routines. For example, what if a timer was running that would interrupt and check a system condition which would then result in queuing an LCD update? This would violate our “keep ISRs simple” idea and is arguably poor design practice, but nothing prevents it from happening. Never leave a potential failure like this open in a production device, but here we won’t worry about it.

```

static void TwiSM_NextTransferDelay(void)
{
    TWI_u32Timer--;

    if(TWI_u32Timer == 0)
    {
        /* Clean up the local message queue (interrupts off, so not critical) */
        TWI_u8MsgQueueCount--;
        TWI_psMsgBufferCurrent++;
        if(TWI_psMsgBufferCurrent == &TWI_asMessageBuffer[U8_TWI_MSG_BUFFER_SIZE])
        {
            TWI_psMsgBufferCurrent = &TWI_asMessageBuffer[0];
        }

        /* Make sure _TWI_INIT_MODE flag is clear */
        if(TWI_u8MsgQueueCount == 0)
        {

```

```

    TWI_u32Flags &= ~TWI_INIT_MODE;
}

TWI_pfnStateMachine = TwiSM_Idle;
}

} /* TwiSM_NextTransferDelay */

```

Testing and verifying the operation of the driver is essential. It is close to impossible to write a low-level driver correctly the first time. Effectively using the debugger is very important to solving problems efficiently and confirming the operation. In the figure shown, you can see that our debugging environment is carefully set up to provide all of the code, variable, and register information relevant to the TWI driver. If you do nothing else in this book, at least try setting up the debugger and looking at everything that is going on in the system!

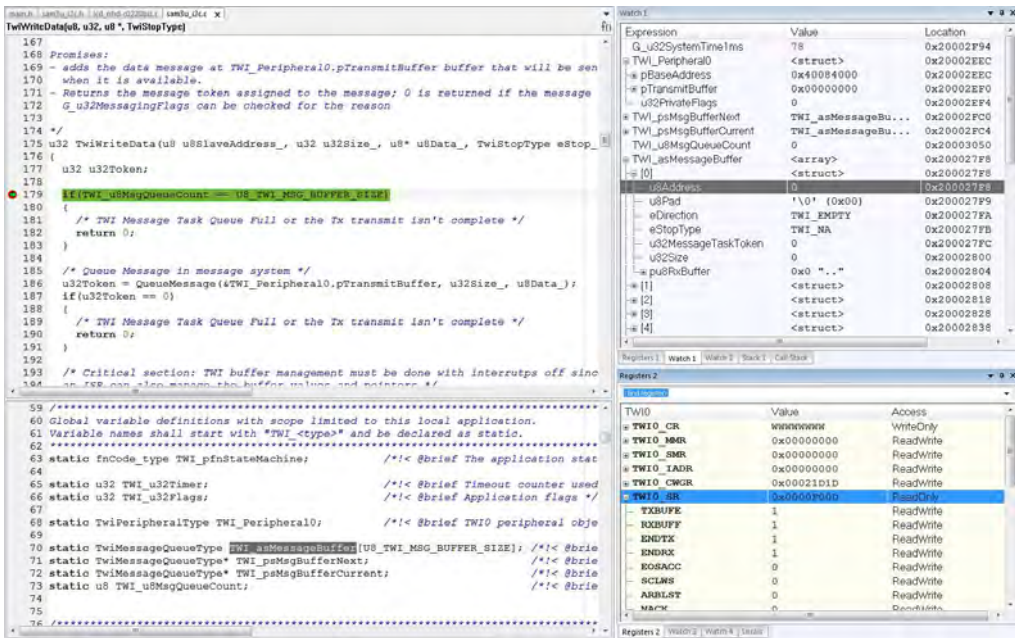


Figure 12-11 Debugger environment

## 12.10 • TWI Receive



The receive case is similar to transmit from a high-level perspective but the details are quite different. Again, take the time to design and code this to get the full experience of implementing the peripheral and solving the myriad of problems that will come along. As far as hints go, perhaps the most important is that to use PDC transfers, the PDC receive counter must be one less than the total number of bytes expected. Setting up for the last byte needs to tell the TWI peripheral to generate the STOP condition and the PDC hardware cannot do that automatically. This also means that single byte receives are special cases that will not even use the PDC receive function.

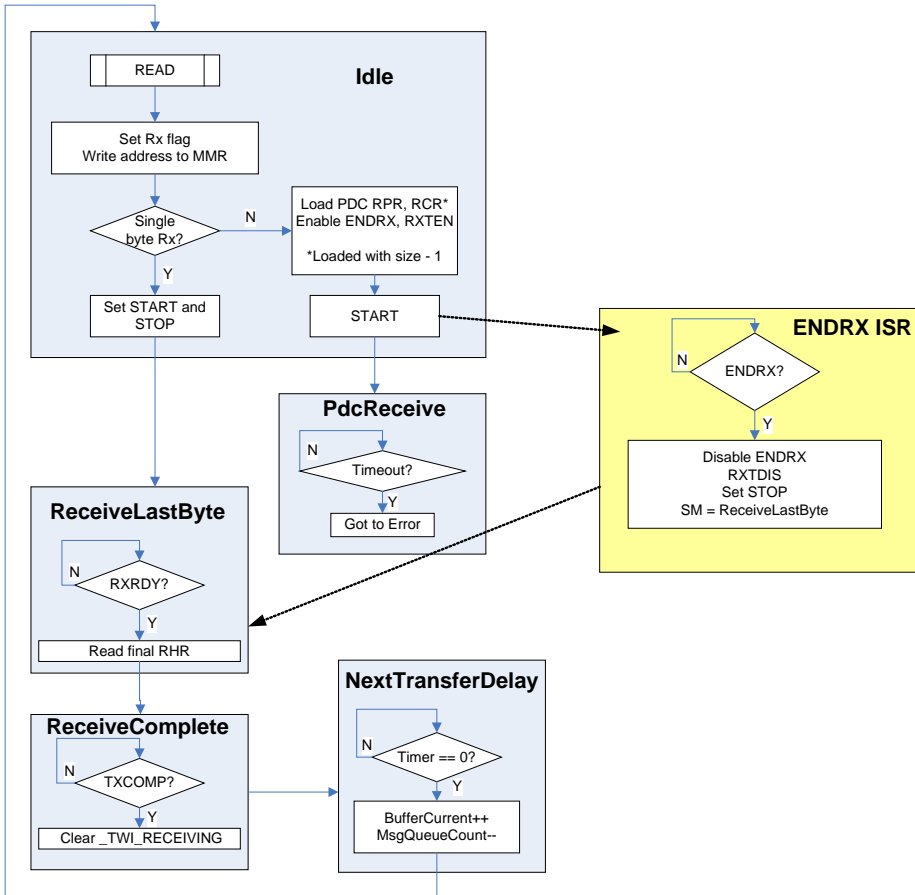


Figure 12-12 I2C Rx state machine

Our TWI\_READ case starts by updating flags and preparing the read address. The check and response for a single byte receive is done here because in this case the code only has to set START and STOP and proceed to the ReceiveLastByte state. Remember that for receive operations, the transaction is initiated by setting the START bit.

```

else if(TWI_psMsgBufferCurrent->eDirection == TWI_READ)
{
    /* Set up for READ transaction */
    u32Byte = AT91C_TWI_MREAD |
        (TWI_psMsgBufferCurrent->u8Address << TWI_MMR_ADDRESS_SHIFT);
    TWI_Peripheral0.pBaseAddress->TWI_MMR |= u32Byte;
    TWI_Peripheral0.u32PrivateFlags |= _TWI_RECEIVING;

    /* Set up to receive the message based on number of bytes */
    if(TWI_psMsgBufferCurrent->u32Size == 1)
    {
        /* Single byte direct receive (no PDC required) */
        TWI_Peripheral0.pBaseAddress->TWI_CR = (AT91C_TWI_START | AT91C_TWI_STOP);
        TWI_pfnStateMachine = TwiSM_ReceiveLastByte;
    }
}

```

Multi-byte receives use the PDC so RPR and RCR need to be updated. RCR is set for one less than the number of bytes to receive so the PDC will be finished early and can hand the last byte reception back to the state machine. While receives are in progress, the TWI task is in PdcReceive.

```

else
{
    /* Multi-byte PDC-based receive */
    TWI_Peripheral0.pBaseAddress->TWI_RPR =
        (u32)TWI_psMsgBufferCurrent->pu8RxBuffer;
    TWI_Peripheral0.pBaseAddress->TWI_RCR = TWI_psMsgBufferCurrent->u32Size - 1;
    TWI_Peripheral0.pBaseAddress->TWI_IER = AT91C_TWI_ENDRX;
    TWI_Peripheral0.pBaseAddress->TWI_PTCR = AT91C_PDC_RXTEN;

    /* Trigger the peripheral to start */
    TWI_Peripheral0.pBaseAddress->TWI_CR = AT91C_TWI_START;

    /* Proceed to receiving state*/
    TWI_u32Timer = G_u32SystemTime1ms;
    TWI_pfnStateMachine = TwiSM_PdcReceive;
}
} /* end TWI_READ */
} /* if(TWI_u8MsgQueueCount != 0) */

} /* end TwiSM_Idle() */

```

PdcReceive doesn't have to do anything except watch for a timeout. The state will be advanced in the ENDRX interrupt handler.

```

static void TwiSM_PdcReceive(void)
{
    if( IsTimeUp(&TWI_u32Timer, U32_RX_TIMEOUT_MS) )
    {
        TWI_u32Flags |= _TWI_ERROR_RX_TIMEOUT;
        TWI_pfnStateMachine = TwiSM_Error;
    }
}

} /* end TwiSM_PdcReceive() */

```

It is expected that the ENDRX ISR will update the TWISM state and disable the PDC so the last byte can be received manually. The STOP condition is set inside the ISR before it exits as this is an urgent operation that cannot wait for a potential 1ms state delay.

```

/**/ ENDRX (receive has finished ALL BUT ONE bytes) ***/
if(u32InterruptStatus & AT91C_TWI_ENDRX )
{
    /* Disable interrupt and PDC transfer */
    TWI_Peripheral0.pBaseAddress->TWI_IDR = AT91C_TWI_ENDRX;
    TWI_Peripheral0.pBaseAddress->TWI_PTCR = AT91C_PDC_RXTDIS;

    /* Set stop condition and change states */
    TWI_Peripheral0.pBaseAddress->TWI_CR = AT91C_TWI_STOP;
    TWI_pfnStateMachine = TwiSM_ReceiveLastByte;
}
} /* end ENDRX handler */

```

Once the last receive is in progress, the system can take its time to wait and confirm. Confirmation is done by checking the RXRDY bit which indicates that the byte has been

clocked into RHR. The biggest trick is reading this byte into the correct location of the receive buffer. The `TWI_psMsgBufferCurrent` pointer has all the information needed to find the correct address for the last byte. If you did not try writing this solution, at least look at the code and convince yourself that this address is correct.

```
static void TwiSM_ReceiveLastByte(void)
{
    if( TWI_Peripheral0.pBaseAddress->TWI_SR & AT91C_TWI_RXRDY )
    {
        /* Read the final byte */
        *(TWI_psMsgBufferCurrent->pu8RxBuffer + TWI_psMsgBufferCurrent->u32Size - 1) =
            TWI_Peripheral0.pBaseAddress->TWI_RHR;

        TWI_pfnStateMachine = TwiSM_ReceiveComplete;
    }
}

/* end TwiSM_ReceiveLastByte() */
```

We used a final state to verify that the whole receive operation was complete which includes the completion bit that gets set once the STOP condition has been placed on the bus. Again, this should occur very quickly after the last byte has arrived, but there are no guarantees about that so we must ensure the 1ms system timing is not disrupted.

```
static void TwiSM_ReceiveComplete(void)
{
    if(TWI_Peripheral0.pBaseAddress->TWI_SR & AT91C_TWI_TXCOMP_Master)
    {
        /* Clear RX flag and advance states */
        TWI_Peripheral0.u32PrivateFlags &= ~_TWI_RECEIVING;

        TWI_u32Timer = U8_NEXT_TRANSFER_DELAY_MS;
        TWI_pfnStateMachine = TwiSM_NextTransferDelay;
    }
}

/* end TwiSM_ReceiveComplete() */
```

When this state is finished, the Rx function exits through the same transfer delay state that was already covered in the transmit case.

### 12.10.1 • NACK and Errors

The only remaining code to write in the TWI driver is the NACK ISR and the TWI Error state. NACKs only occur when Masters are transmitting. As alluded to earlier, it is unlikely that NACKs will occur in a mature system. If they do, handling them will be application-specific.

At the code level that the EiE firmware system is written at, any NACK suggests a catastrophic failure that will be dealt with as the system is developed or as new tasks are added. If the base firmware was used in a production device, an appropriate NACK handling strategy should be developed and implemented. For now, the NACK ISR shuts down the PDC transmit, disables the `ENDTX` interrupt and changes the TWI driver state machine to the Error state.

```
/** NACK Received (Master only) */
if(u32InterruptStatus & AT91C_TWI_NACK_Master )
{
    /* Error has occurred, abort the message */
```

```

    TWI_u32Flags |= _TWI_ERROR_NACK;
    TWI_Peripheral0.pBaseAddress->TWI_IDR = AT91C_TWI_ENDTX;
    TWI_Peripheral0.pBaseAddress->TWI_PTCR = AT91C_PDC_TXTDIS;
    TWI_pfnStateMachine = TwiSM_Error;
}

```

The most likely use case will be a breakpoint inside this ISR that will be used during driver development. If a task is being written that is not working properly, the same breakpoint may be helpful to see immediately when a NACK is occurring.

The TWI error state in the EiE firmware is meant to report errors, not attempt to fix them. If an application has been coded and is thought to be working correctly, watching for the error message on the terminal would be sufficient. Without any recovery code added, it is very likely that the system will have to be reset and run again. This is very “dev-minded” thinking. If “just reset it” is a published solution for a production device, think again.

The error state parses error flags and deals with each one or groups of them if they are related. Multiple errors will be processed in the call and by the end, all error flags will be clear. The state still exits through NextTransferDelay to make use of that code to clean up other variables and buffers.

```

static void TwiSM_Error(void)
{
    /* Transmit errors */
    if( (TWI_u32Flags & _TWI_ERROR_NACK) ||
        (TWI_u32Flags & _TWI_ERROR_TX_MSG_SYNC) )
    {
        /* Announce the error */
        DebugPrintNumber(TWI_Peripheral0.pTransmitBuffer->u32Token);

        if(TWI_u32Flags & _TWI_ERROR_NACK)
        {
            DebugPrintf(" TWI NACK.");
        }

        if(TWI_u32Flags & _TWI_ERROR_TX_MSG_SYNC)
        {
            DebugPrintf("TWI transmit message out of sync!");
        }

        DebugPrintf(" Message deleted.\n\r");

        /* Clear flags and clean up the Message task message */
        UpdateMessageStatus(TWI_Peripheral0.pTransmitBuffer->u32Token, FAILED);
        DeQueueMessage(&TWI_Peripheral0.pTransmitBuffer);
        TWI_Peripheral0.u32PrivateFlags &= ~(_TWI_TRANSMITTING | _TWI_TRANS_NOT_COMP);
    }

    /* Receive errors */
    if(TWI_u32Flags & _TWI_ERROR_RX_TIMEOUT)
    {
        DebugPrintf("TWI Rx Timeout.\n\r");
    }

    /* Clear error flags and advance states */
    TWI_u32Flags &= TWI_ERROR_FLAG_MASK;

    TWI_u32Timer = U8_NEXT_TRANSFER_DELAY_MS;
    TWI_pfnStateMachine = TwiSM_NextTransferDelay;
} /* end TwiSM_Error() */

```

Any errors should be tracked down and solved for the root cause. Definitely finding error sources is critical for production devices. “Trying something” and then rebuilding to see if it works is dangerous unless you have identified and resolved a specific issue. In the ideal case, the conditions for the error should be understood to the point where the problem can be repeated consistently. Then the fix can be applied and tested to try to show that the problem is solved. This can take hours, or days, or weeks.

Even if the system is coded to recover, the debug output is still very useful for long-term tests where the output can be logged and reviewed later. Our solution prints the message Token for transmit errors. Adding the system time-stamp (or configuring the terminal to stamp it with real-time) would be a logical next step. The more information you have, the better.

What we hope is ridiculously clear by this point is that not only is there a very large amount of thought, design, code and test involved in building a firmware system, there are also infinite ways to make the system better. Deciding what to do is all part of the extensive tradeoffs that must be considered in the design.

### 12.11 • ASCII LCD

The EiE development board has become more and more useful as each chapter and driver is completed. Hopefully, you can start to imagine some applications for the board or some embedded applications in general that you could build with different hardware and a similar driver set. Now that the TWI functionality is ready, we can finally get the LCD up and running. This presents a few new challenges since we must interface to a system external to our own microcontroller.

Even though a UART debugging port can provide all the access an engineer requires to check on the device status or get information in or out of the system, it is not very helpful to the average user. Very simple user interfaces can get by with a few LEDs since many people can understand that a green LED is probably good and a red LED is probably bad. You might be tempted to add a few more colors of LEDs for status, and then start blinking those LEDs or turning on different combinations of them to mean various things. For engineers and maybe even product support people whom you can train, that approach is okay and understandable. If you think that a typical consumer will have any hope of understanding what a 1Hz blinking LED means vs. a 4Hz blinking LED means or even looks like, you are in for a nasty surprise!

A liquid crystal display (LCD) offers a HUGE advantage for a user interface since it can communicate graphically or in plain text. Grayscale text or dot matrix displays are relatively inexpensive and quite easy to implement. If you want high resolution and color, TFTs (thin film transistor) can be used but at a cost of complexity, power consumption and dollars. On the downside, LCDs are fragile since they are pieces of glass with ultra-tiny wires, and they do not perform well in cold temperatures since the liquid freezes or at least gets very thick and slow.

LCDs come in many shapes and forms, but all rely on the same basic technology to make them work. This section starts by looking at the physical characteristics of an LCD display, discusses the various configurations you will come across, and then looks specifically at the part used on the development board on which the firmware driver will be built to use it.

#### 12.11.1 • LCD Hardware

There is nothing magical about the way an LCD works. Every pixel has a positive and return connection through which that pixel can be electrically told to be on or off. If you look at an LCD, you might not think that can be physically possible, but if you look under



high magnification, you will see tiny wires embedded in the glass and tiny vias connecting different layers. For LCDs with very small and dense pixels, there will be multiple signal layers and multiple ground layers/backplanes.

As resolution increases further, the displays use a different technology called “TFT” (thin film transistor) where every pixel has a transistor switch. This allows row and column addressing of each pixel. You might think that is a lot of transistors, especially on a big 2500 x 1600-pixel display – 4 million! But considering how physically big a display like that is, a few million transistors is nothing compared to the billions of transistors that are in microprocessors these days. It is still impressive technology, though, and there are only a few factories in the world that produce these displays that various vendors use in the LCD monitors and TVs.

### **LCD Pixels Characteristics**

In the context of the EiE dev board, the LCD is an “FSTN” type or, filtered super-twisted nematic display. A pixel is lit by biasing it with a positive voltage, which aligns the particles in the liquid crystal solution. The pixel does not actually light up. In fact, a pixel that is “on” and visible is really blocking light from reflecting off the back surface of the glass. The front side of the glass is polarized as well, so when the crystals are aligned 90° different than the glass, all the reflected light is blocked. The exact same effect can be observed with a pair of polarization filters for a camera or two pairs of polarized sunglasses. Stack the lenses on top of each other and look through them in different orientations.

LCDs draw very little current though the circuit does need to remain energized and refreshes continuously to keep the crystal solution polarized. New displays are emerging that do not even need to be refreshed, making static images with nearly zero current draw possible (check out “E ink” and zero-power (bi-stable) LCDs). An analogy is dynamic RAM where each memory cell is like a pixel and has a capacitance that must be charged and refreshed periodically to stay energized. Eventually, the charge will drain if it is not refreshed. To improve the look of LCDs, a backlight is often added which make the overall power draw considerably higher. The EiE ASCII LCD has an RGB backlight so the screen can be set to any color desired.

Dot-matrix LCDs are common and allow the programmer to draw any character or image they want if they can make it out of the available dots. Though this provides some freedom, it also comes with more overhead as more care must be taken to draw shapes and you can only get detail in resolution of the smallest size of you square pixels.

Pixels do not have to be small squares in a grid. LCDs can be created with “pixels” that are whole images or symbols. In this case, the resolution and the detail of the pixel shape can be extremely fine and precise. For example, an alarm icon showing a small ringing clock can be done as a single pixel, even if the icon is not entirely continuous. A 7-segment-like display can be made with 7 “pixels,” and thus numerical characters can be shown with 7 pixels per character just like a calculator that uses 7-segment LED displays. Hint: use 11 segments to allow a much wider range of letters and numbers!



**Figure 12-13** Example of a custom, hybrid dot-matrix LCD display

Custom LCDs are relatively inexpensive to produce though they include some tooling costs at the beginning where the vendor creates the design you want. You can choose how the pixels are hooked up, what kind of controller (if any) is on the LCD device, and how it will connect to your product. Once the design is complete, the custom LCD can be produced in mass quantity for only a few dollars each. The instrument shown above is a design we did that has a custom LCD with predefined symbols as well as dot-matrix areas for writing numbers and letters. The Key symbol is a single segment, as are the tiny gas names under each reading.

#### 12.11.2 • LCD Controllers

Making pixels light up is easy in theory and it is barely different than turning on a discrete LED. The hard part comes more from the physical quantity of pixels since every pixel needs to be addressed. This can result in a lot of hardware to manage. Stand-alone LCDs have a controller chip that takes care of most of the hard parts about setting and clearing LCD segments. All your host micro must do is tell the LCD controller which pixel(s) are to be on or off, and the controller will take care of any addressing, backplane management or voltage waveform generation to make it happen. It is not uncommon for these controllers to have hundreds of pins.

There are a few microcontrollers that have built-in LCD drivers. These are peripherals built to work with any LCD (either custom or standard) that can connect directly to the LCD pixel and backplane wires. The programmer is responsible for all the mapping, though the peripheral provides some help with backplane management and generates a somewhat complex waveform to make sure all the pixels work together with their backplanes. The downside of built-in controllers is that they are usually limited in the total number of pixels that can be supported due to the physical size of the MCU. While this is often sufficient for symbol-based or 7-segment style displays, it is unlikely able to support a dot-matrix display.

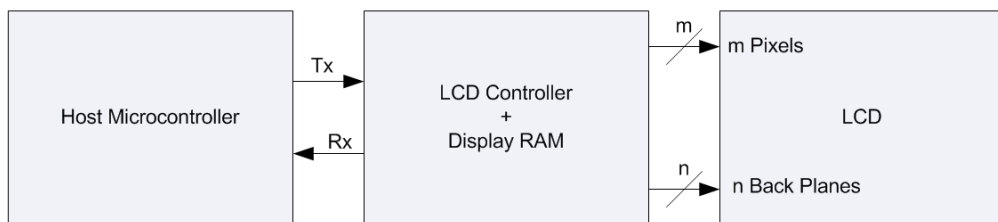
An LCD controller requires physical connections for all the pixel lines, so there will usually be a specification for the maximum number of pixels that the controller can support. An integrated MCU LCD controller, for example, may have 16 lines. Since 16 pixels is barely enough to make two characters, the controller will probably support two or more backplanes. Each backplane control requires an additional output signal line that can produce a “backplane waveform.” This brings the total number of pixels that can be supported to [pixel lines] x [backplanes]. So the 16-pixel device that can run 4

backplanes can support 64 pixels, and that starts to look a lot better. With 64 pixels, you could run six 7-segment characters and still have 22 other symbols available. However, you still are using 20 pins of the microcontroller which will occupy a lot of the processor GPIO just for the LCD.

If you need more pixels, then you have no choice but to go to an external LCD controller. There is an added cost to this, but also many advantages. Many LCDs are “chip on glass” (COG) which means you buy the whole LCD and controller as one piece. If it is not a custom design, these can be quite reasonably priced even in low quantity. This is exactly what the EiE LCD is. Other LCDs might have the glass on a small PCB and have an LCD controller chip on that PCB. Communication to tell the controller what pixels to light up now takes place with a digital interface like SPI, I<sup>2</sup>C, parallel, or others.

### 12.11.3 • LCD Interface

Ultimately, wiring up an LCD system requires three parts.



**Figure 12-14 LCD interface diagram**

The Host Microcontroller is what triggers the LCD controller to drive pixels on or off on the LCD. The LCD controller takes commands and data from the Host and configures its display RAM that maps to the LCD pixels. Then the controller updates its output lines that cause the pixels to update on the LCD itself.

The work required to generate the data and organize it into display RAM is pretty much the same in any LCD system, but the share of the work between the host and the LCD controller can vary significantly. For most of the basic character display LCDs that you work with, the LCD controller takes care of most of the work.

The EiE LCD is a very common type of alpha-numeric display. The interface commands are known as the HD44780 standard and originally developed by Hitachi. Though this display is a two-line by 20-character display, the controller chip will support LCDs down to one-line, 8-characters all the way up to four-line, 20-characters. If you are looking at any ASCII LCD, it is probably being driven by an HD44780 controller.

The controller chip itself runs a basic application that allows other devices to send commands and ASCII characters that the controller will interpret. The front-end of the EiE ASCII controller uses an I<sup>2</sup>C interface which was very important for the development board due to limited GPIO lines available on the processor. The standard HD44780 character LCDs use a parallel data interface (8 lines) along with 3 control lines and thus require 11 processor lines to communicate with. Though that makes them slightly easier to use since the communication is very intuitive, the hardware overhead can be impractical.

Correctly attaching the LCD into the development board PCB involves carefully looking at the LCD datasheet to put the right parts on the board. With I<sup>2</sup>C, there are only two communication lines, but the LCD requires additional hardware for the controller to work

properly as well as an external interface to the three LEDs that make up the backlight.

The hardware connections as described by the LCD datasheet and the corresponding hardware connections implemented for the development board are shown. Note that the 5V rail is used for the LEDs with relatively small resistors so the backlights are quite bright. Since we need to switch 5V lines and because the LEDs are high current, transistor switches are used.

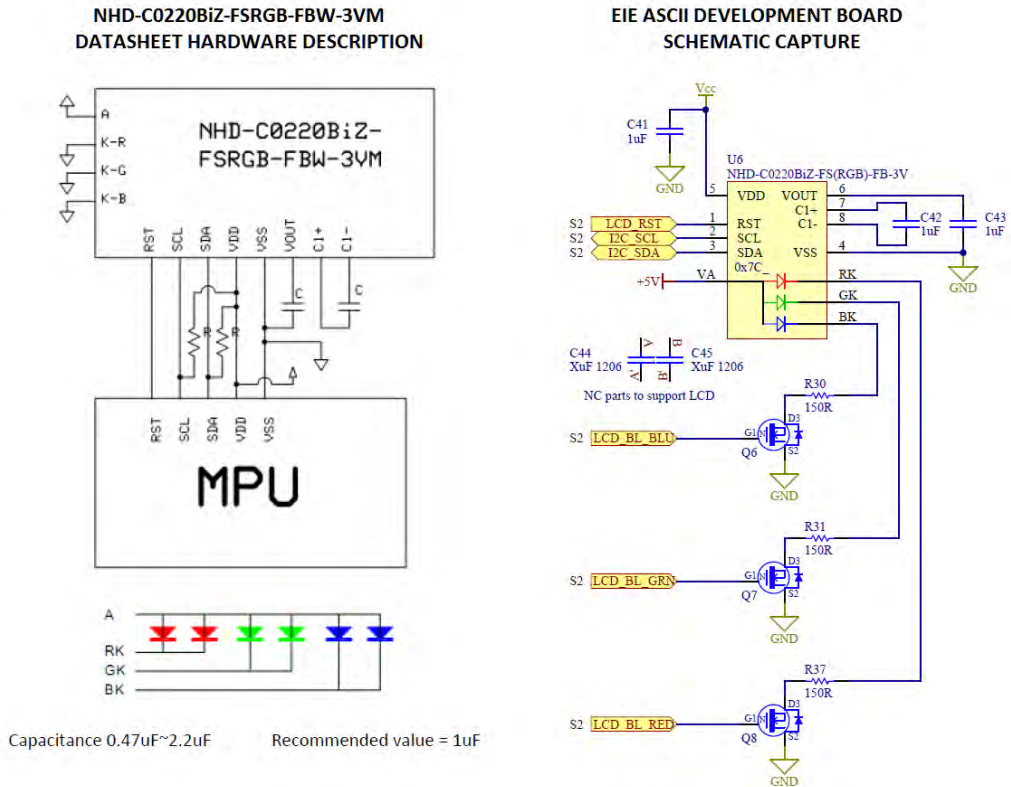


Figure 12-15 LCD hardware connections

### 12.12 • Character and Control Data

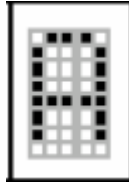
When communicating to this LCD, two modes are used: character and control. This determines how the LCD controller will interpret the binary data it receives from the host MCU. It is likely that every LCD that you work with will use this strategy or something very similar. For the EiE LCD, the mode is selected in the first few bytes of the communication sequence.



If you have not done so already, open the LCD datasheet and take a quick read through to set up the rest of this discussion. A copy of the datasheet is stored on the EiE website (LCD\_Character\_Newhaven\_NHD-C0220BiZ-FSRGB-FBW-3VM.pdf). Note that this datasheet is prepared by the LCD vendor and describes the complete LCD and not just the controller. Much of the information is a subset of the full LCD controller datasheet that you can find via a link in the vendor's datasheet. The controller's datasheet is 70 pages long and contains a lot of information that you do not really need to use the LCD, though

you may want to browse to learn more. It is a good idea to keep a copy of LCD datasheets because they tend to disappear from the web on occasion.

In character mode, an ASCII LCD will take the bytes you send it with the assumption the data are ASCII characters and light up the correct pixels corresponding to the character sent. In other words, you send the character 'A' (0x60) and magically the pixels required to make an 'A' will light up at the current cursor location. But what really is happening here? The controller sees that you want to write the letter A because you have sent the character using the defined protocol. To know what an 'A' looks like, the LCD controller must look up the bitmap for the character that it has stored in its memory that tells it what pixels need to be on to make an 'A' appear. The block of pixels for each character bitmap in this LCD is 5 columns by 8 rows as shown.



**Figure 12-16** Bitmap for a 5 x 8 dot letter 'A'

Each character can exist at any of the 20x2 character positions on the display screen. Though each pixel on the screen has an address accessible to the LCD controller, only full character addresses are used by the host MCU to indicate where on the screen the character should be placed. Once the bitmap is determined, the controller can activate the corresponding pixels.

While most character LCDs have built-in bitmaps for at least one font, custom LCDs and many graphical / dot matrix LCDs do not have any bitmap data, so it is up to you to provide all the bitmaps. The EiE Dot Matrix development board is an example and you can find a tutorial about writing that driver on the EiE website if you want to see an example of a fully customized dot matrix LCD.

In addition to having a font stored in ROM for you to use, there are other features available. For example, the controller allows you to set the cursor position or blank the display, or choose between a solid, blinking, or no cursor. When characters are being sent to the display, the controller can be set to automatically change addresses after each character so that it is ready in the next sequential position when the next character requested comes in on the interface. Instructions to move the cursor to a different location, erase characters, change the cursor, erase the whole screen, etc. are also available. These commands are accessed when the controller is in command mode.

All the functionality is included in the price of the LCD and makes integrating an alphanumeric LCD relatively easy. The datasheet lists the commands available, the font set available, and the memory organization of the LCD. The main information is looked at further on in the chapter to write the driver. We need some of the datasheet information now to ensure our I<sup>2</sup>C driver is going to provide the services needed.

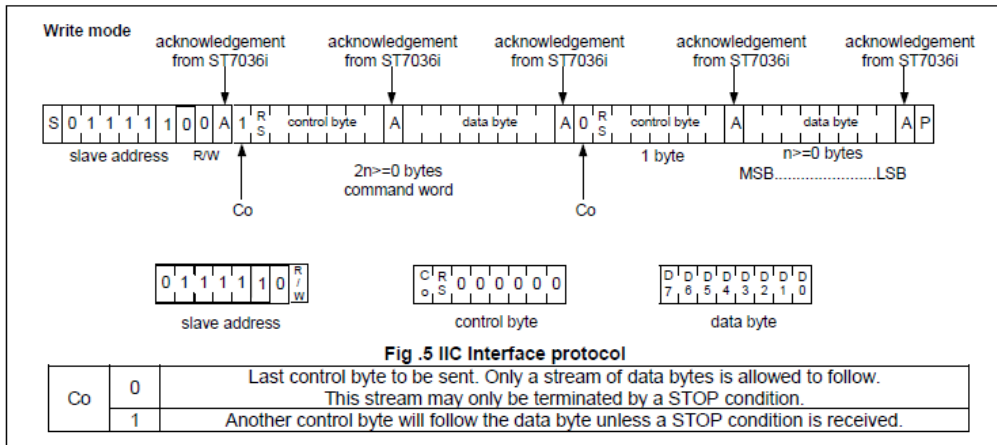
### 12.13 • Using the LCD Controller

Alpha-numeric displays with an ASCII interface are pretty easy to use since the built-in controller will handle most of the work. All we need is to program the interface to meet the protocol and write some functions to send character updates. If you have not read the datasheet for the LCD yet, you pretty much cannot go forward without doing so. What you need to note now are five things:



1. LCD I2C address
2. Control Byte
3. Character RAM addresses
4. LCD command set
5. LCD initialization sequence

The vendor datasheet captures the most useful information from the controller information along with the relevant hardware references, electrical specifications and example code for initialization. There are a few details that did not make it over that might be nice to know, such as how the I2C version of the controller only supports Write mode (see page 14 of the controller datasheet). Both documents have the drawing shown that we will reference several times in the following sections.



**Figure 12-17 I2C command example from controller specification**

The address byte is the first byte sent to an I2C device after the START condition. Remember that the I2C standard dictates that device addresses are 7 bits in length plus the Read (1) or Write (0) bit in the LSB. Specifying the device address does not seem to follow a standard form. For example, if the address of a device is given as 0x64, that might mean the full 8-bit address byte you send to Read the device is 0x65 since the LSB is high to indicate Read. It might also mean that the first 7 bits are 0x64 (b'1100100x') and you need to add an 8th bit for x=R/W. We attempt to illustrate this, but admittedly it is a little hard to explain in text.

0x64	Specified device address							
MSB	6	5	4	3	2	1	LSB	Comment
0	1	1	0	0	1	0	0	Address includes the R/W bit - "Write" address is 0x64
0	1	1	0	0	1	0	1	"Read" address is therefore 0x65
1	1	0	0	1	0	0	0	Address was meant as only upper 7-bits - "Write" address is 0xC8
1	1	0	0	1	0	0	1	"Read" address is 0xC9

**Figure 12-18 Address bytes**

Fortunately, in the case of the LCD controller, the diagram shows the actual bit detail of the address byte so you can deduce that the write address is  $b'01111100' = 0x7C$ . Unfortunately, that is NOT what the device address is. If you dig through the longer controller datasheet, you will find some explanation on page 16 that describes four different possible addresses that the controller can be set to. The address the vendor picked happens to be different than the one the LCD controller vendor picked for their example.

Regardless, the vendor datasheet says in large bold letters that the address is 0x78 but they fail to mention which of the above formats that value is in. With a quick bit of testing, the complete LCD Write address byte was confirmed as 0x78, so this particular specification format matches the first case shown above and thus means the 7-bit address is 0x3C. This address is captured as a constant in the header file, `LCD_ADDRESS_WRITE`. This test was carried out during driver testing to confirm that it was acknowledged by the LCD. Fortunately, that part of the example code was correct. Imagine if you were not sure if you had the correct LCD address and spent hours (or days) debugging your I<sup>2</sup>C driver and/or board hardware because the LCD refused to respond to its address, only to (finally) try changing the address and discover that everything was working just fine! It is problems like that which are extremely frustrating, especially with the tight deadlines that you will always have in industry.

#### 12.14 • Control byte with Co and Rs

Communicating to the LCD requires following a strict input and output protocol that is very typical of any MCU-IC interface. After the LCD controller acknowledges its address, the next byte sent must be a control byte that has two important bits called Co and Rs. Rs is the bit used to tell the LCD if the bytes that follow are meant as control codes or simply data that should be written to the screen. If you read the controller datasheet, you will see a truth table that shows that Rs works in conjunction with the R/W bit to get four different modes from the LCD controller. However, as we have mentioned already, the I<sup>2</sup>C version of the LCD controller does not support Read operations, so there is only, in fact, two modes that can be accessed:

- Rs = 0: Instruction mode
- Rs = 1: Data mode

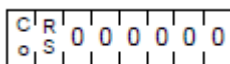


Figure 12-19 Control byte

The Rs bit works in conjunction with the Co bit.

- If the Co bit is set, then the next byte sent will be interpreted as a data mode byte or instruction mode byte depending on what Rs was. The byte after will be interpreted as another control byte allowing you to change the Rs bit and thus toggle between data or instruction mode.
- If the Co bit is clear in the control byte, then the controller assumes that all subsequent bytes in the transmission session are of the same Instruction mode or Data mode that is indicated by Rs so you can send a series of bytes in the same mode.

In other words, the Co bit allows you to change modes during a transmission session. There might be a scenario where you wanted to send a command byte, send a data byte, send another command byte, send another data byte, etc. There would be some extra overhead if you had to set a STOP condition and restart a new frame each time you



wanted to change modes. However, this scenario is unlikely to occur so the Co bit tends to be unused. For all of the communications in the driver we build here, Co is always 0 so each transaction will only ever be a bunch of commands or a bunch of data. If we need to switch modes, the session should be stopped and a new one started.

#### 12.14.1 • Character RAM Addresses

This LCD controller supports screens up to 20 columns x 4 rows. The ASCII character data that ends up being displayed on the screen is stored in the controller's "Display Data RAM" (DDRAM) in which there are 80 bytes of space. Even if you have an LCD screen that is smaller than that, you can still write to all the character addresses and the data will be stored but not displayed beyond the addresses that are visible on screen. You can use the non-displayed area as general-purpose RAM if you really wanted to or make use of the shifting features to do some message scrolling. For the 20 x 2 implementation of the EiE LCD, the RAM is organized as shown in the LCD controller datasheet.

Display Position		1	2	3	4	5	6		38	39	40
DDRAM Address (hexadecimal)		00	01	02	03	04	05	.....	25	26	27
		40	41	42	43	44	45	.....	65	66	67

Display Position	1	2	3	4	5	6	7	8		17	18	19	20
DDRAM Address	00	01	02	03	04	05	06	07	.....	10	11	12	13
	40	41	42	43	44	45	46	47	.....	50	51	52	53
For Shift Left	01	02	03	04	05	06	07	08	.....	11	12	13	14
	41	42	43	44	45	46	47	48	.....	51	52	53	54
For Shift Right	27	00	01	02	03	04	05	06	.....	0F	10	11	12
	67	40	41	42	43	44	45	46	.....	4F	50	51	52

Figure 12-20 LCD Character RAM

The displayed characters for Line 1 will be at addresses 0x00 thru 0x13 and hidden characters will be at 0x14 thru 0x27. Line 2 visible addresses are 0x40 thru 0x53 with hidden characters 0x54 thru 0x67.

The second part of the figure shows what happens when you request a shift command (either left or right). For Line 1, a left shift moves the displayed character range to 0x01 – 0x14; a right shift displays 0x27, 0x00 – 0x12. Line 2 has similar behavior with the important concept being that each line rotates through itself. What all this means is that you should be able to do a bunch of neat scrolling effects on a per-line basis just using



control commands to shift the display. The only limitation is that there is just a single shift command so both lines would have to scroll. However, the scrolling behavior does not work like you might expect. Try it during the chapter exercise, but you will find that it will not really work for what the exercise requests you to do.

Scrolling displays aside, you need to be comfortable with the character addressing since you must use it to position the cursor where characters will be displayed. You can write to any location on the screen first by setting the cursor address, then by loading a message.

### 12.14.2 • LCD Command Set

Randomly sending bytes to the LCD will not get you very far – you must use the defined command set to talk to the device correctly. The complete set of commands that you need is shown on pages 7 and 8 of the vendor LCD datasheet. Any command where the R/W bit is 1 (Read) is not available in I<sup>2</sup>C mode.


Most of the commands are obvious for what they do. Many of the commands have configurable bits that need to be set or cleared depending on what you want to accomplish. For example, the Display ON/OFF command is shown as b'00001DCB' so the actual command is b'00001xxx' with three configurable bits. The command description tells you what the individual bits do.


Display ON/OFF	0	0	0	0	0	0	1	D	C	B	D=1:entire display on C=1:cursor on B=1:cursor position on
-------------------	---	---	---	---	---	---	---	---	---	---	--

Figure 12-21 Display command example

```
u8 u8Command;  
u8Command = (LCD_DISPLAY_CMD | LCD_DISPLAY_ON |  
              LCD_DISPLAY_CURSOR | LCD_DISPLAY_BLINK);
```

The LCD controller datasheet offers further description of each command and its parameters if you require it. There are some commands like for setting hardware behavior that, to use correctly, you would have to do some reading in the controller datasheet and/or test out some settings. The vendor has done this already and included the information in their suggested initialization sequence that is discussed in the next section. The only one you might want to play with is the Contrast Set command if your display contrast needs adjustment.

 All the command literals have already been set up in `lcd_nhd_c0220biz.h` – open up this file and the source code file if you have not done so yet.

 *When working with a new device, be prepared to spend the time to build a good header file with all the protocol constants written out. Depending on the type of device you are working with, this might take a few minutes or it might take hours. However, it is worth the time and will save you much frustration continuously referencing the datasheet and/or hard-coding values without meaningful names. Check with the vendor to see if they have a header file that you can download.*

### 12.14.3 • LCD Initialization

Before regular commands and data can be sent to the LCD, it must be initialized in a specific way. If you do not follow the initialization sequence, then the LCD will not function correctly. This is true for virtually every LCD you work with. The steps should be defined in the vendor or controller documentation.

The EiE LCD module datasheet from the vendor shows an example initialization code snippet but no other documentation.

```

/*****
*           Initialization For ST70361           *
*****/
void init_LCD()
{
    I2C_Start();
    I2C_out(Slave); //Slave=0x78
    I2C_out(Comsend); //Comsend = 0x00
    I2C_out(0x38);
    delay(10);
    I2C_out(0x39);
    delay(10);
    I2C_out(0x14);
    I2C_out(0x78);
    I2C_out(0x5E);
    I2C_out(0x6D);
    I2C_out(0x0C);
    I2C_out(0x01);
    I2C_out(0x06);
    delay(10);
    I2C_Stop();
}
/*****/

```

**Figure 12-22 Initialization code snippet**

Working on the assumption that “delay(10)” meant a 10ms delay, we found that the LCD will often get stuck on the last command 0x06 (LCD\_DISPLAY\_ON). The command does not get ACKed so the screen does not turn on. Upon investigation, the timing delays shown in the example code do not match the specification from the LCD controller datasheet. When the LCD controller initialization process is used, the LCD starts up consistently.

The sequence below is taken directly from the LCD controller datasheet. The minimum delays are clearly specified and the command names are indicated so you know what is happening during the sequence.

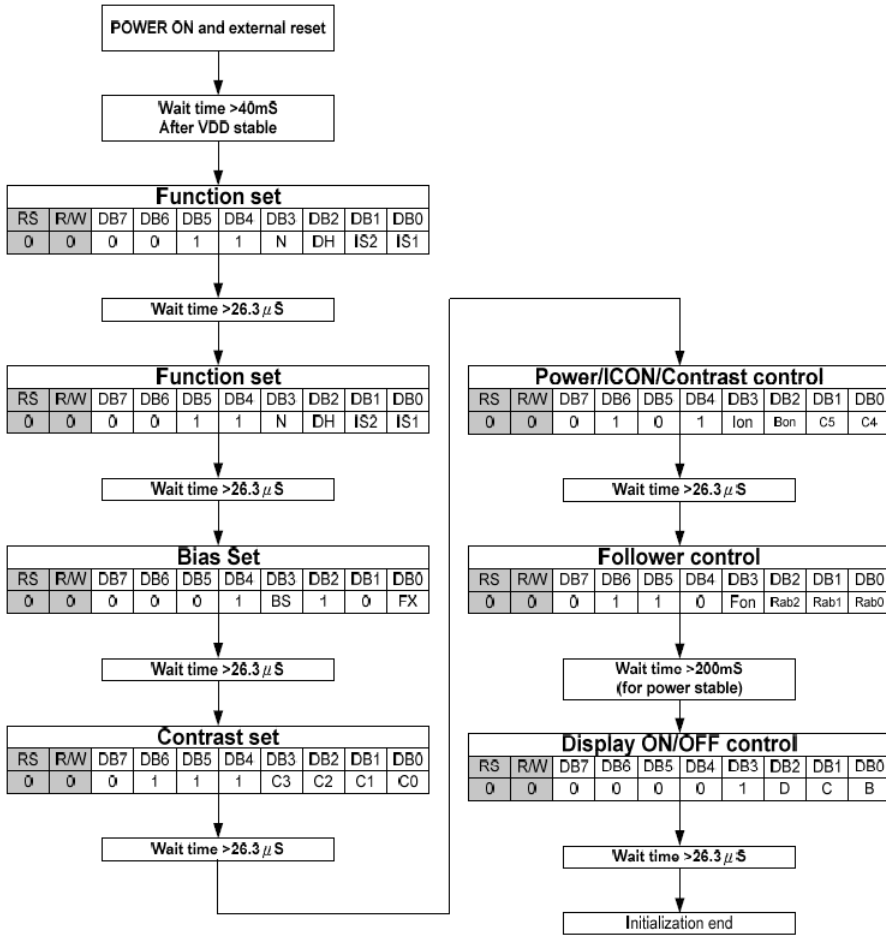


Figure 12-23 LCD Initialization sequence



Write the `LcdInitialize()` function based on the sequence shown. At this layer of code, call the `TwiWriteData()` function directly to issue the commands. `IsTimeUp()` can be used for the long delays. With the 200kHz TWI clock, the time between bytes is 40us. It is not clear if the 26.3us spec is for byte periods or a required delay before the end of one byte and the start of the next. The screen seems to function fine if continuous data is sent, so we did not include additional time. If a delay was required, `IsTimeUp()` could be used since the values are just minimums and there are no issues in waiting longer. Alternatively, you can use `kill_x_cycles`.

Our solution does exactly what the start-up sequence requests. The function begins by setting up a command array and welcome message. Notice how the welcome message has a character spacing guide comment above it so you can see what characters will go where. A subtle, easy piece of information that is very helpful.

```

void LcdInitialize(void)
{
    u8 u8Byte;
    u8 au8Commands[] =
    {
        LCD_FUNCTION_CMD, LCD_FUNCTION2_CMD, LCD_BIAS_CMD,
        LCD_CONTRAST_CMD, LCD_DISPLAY_SET_CMD, LCD_FOLLOWER_CMD
    };
    /* "01234567890123456789" */
    u8 au8Welcome[] = "RAZOR SAM3U2 ASCII* ";

    /* State to Idle */
    Lcd_pfnStateMachine = LcdSM_Idle;
}

```

The init sequence is coded with the long delays (40ms and 200ms) but the code relies on the TWI timing to meet the short delays between the commands. The most useful piece of information from the vendor's example init sequence is that the TWI transaction is one long sequence with just a single START condition at the beginning and a single STOP condition at the end. That means that nearly all the `TwiWriteData()` calls should use `TWI_NO_STOP`.

```

/* Turn on LCD wait 40 ms for it to setup */
AT91C_BASE_PIOB->PIO_SODR = PB_09_LCD_RST;
Lcd_u32Timer = G_u32SystemTime1ms;
while( !IsTimeUp(&Lcd_u32Timer, U8_LCD_STARTUP_DELAY_MS) );

/* Send Control Command */
u8Byte = LCD_CONTROL_COMMAND;
TwiWriteData(U8_LCD_ADDRESS, 1, &u8Byte, TWI_NO_STOP);

/* Send Control Commands */
TwiWriteData(U8_LCD_ADDRESS, sizeof(au8Commands), &au8Commands[0], TWI_NO_STOP);

/* Wait for 200 ms */
Lcd_u32Timer = G_u32SystemTime1ms;
while( !IsTimeUp(&Lcd_u32Timer, U8_LCD_CONTROL_COMMAND_DELAY_MS) );

/* Send Final Command to turn it on */
u8Byte = (LCD_DISPLAY_CMD | LCD_DISPLAY_ON);
TwiWriteData(U8_LCD_ADDRESS, 1, &u8Byte, TWI_STOP);

```



As a final step to the initialization, program a custom message and set the backlight to the color of your choice. Adjust one of the commands in `LcdInitialize()` so that the cursor is hidden and not blinking after initialization. The code below is for a one-line message. The pictures show an example of a two-line message with and without the cursor blinking.

```

u8Byte = LCD_CONTROL_DATA;
TwiWriteData(U8_LCD_ADDRESS, 1, &u8Byte, TWI_NO_STOP);
TwiWriteData(U8_LCD_ADDRESS, 20, &au8Welcome[0], TWI_STOP);

Lcd_u32Timer = G_u32SystemTime1ms;
G_u32ApplicationFlags |= _APPLICATION_FLAGS_LCD;

} /* end LcdInitialize */

```

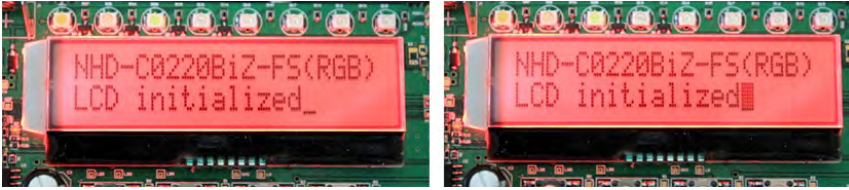


Figure 12-24 LCD initialized (static and blinking cursor)

The first time you get something to show up on an LCD is very exciting! If it shows what you expect and does so consistently, that is even more exciting. By getting to this stage, we have proven that the I<sup>2</sup>C driver is functioning, the LCD hardware is hooked up correctly, the LCD command set is working and the initialization sequence is correct. Given all the things that could go wrong, it is a fair achievement to get to this point in a new design.

**i** *If you think you are doing everything correctly to initialize and display something on an LCD but it just won't show anything, make sure you double check the display contrast settings. It might be that the LCD is working properly, but the message is not visible because of low contrast. We may have lost an afternoon due to that reason once.*

A big part of this driver development was testing and verifying the signaling. The figure below is a scope screen grab of the initialization sequence being sent on the bus. We have both analog and digital probes attached so the signal integrity can be verified along with the simple digital display. The I<sup>2</sup>C logic analyzer in our scope is able to pick out the bytes and recognize the ACKs (denoted 'a').

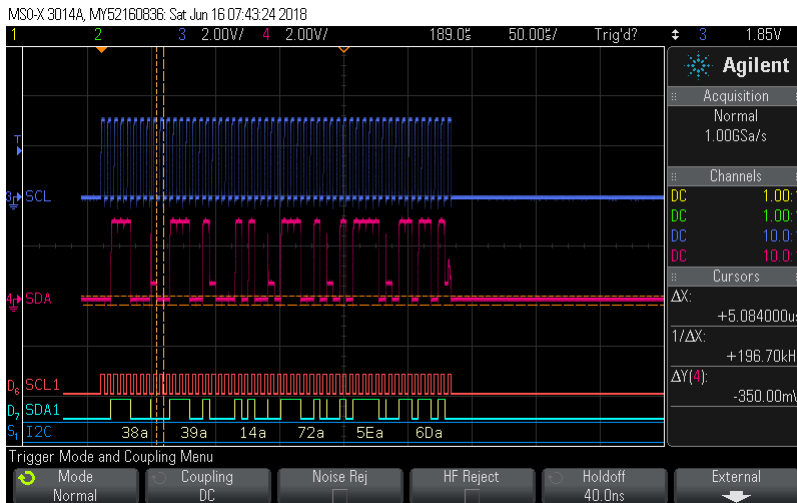


Figure 12-25 Logic Analyzer displays signals

### 12.15 • LCD Application

Even though the lead-up to this point has required a great deal of thought and effort, the final code to write for the LCD application ends up being very simple. The LCD functionality that will be provided is with a set of three API functions that queue appropriate messages to make the LCD perform as required. There is currently no need to run any sort of LCD task, though the hooks are in place should the driver ever evolve to

requiring a state machine.

### 12.15.1 • LcdCommand()

Sending commands is an essential part of using the LCD, especially since the calling application is completely responsible for managing what is on the screen. Moving the cursor and clearing the screen will likely be used most often, but any of the commands might come in handy. This could be further abstracted to a series of API functions that provided specific command functions but figuring out command strings is not difficult to keep the code simple.



The single function parameter takes a literal from the list of LCD Commands in the LCD header file, adds the command into the last location in the command array message, and then queues the array to the I<sup>2</sup>C application.

```
void LcdCommand(u8 u8Command_)
{
    static u8 au8LCDWriteCommand[] = {LCD_CONTROL_COMMAND, 0x00};

    /* Update the command parameter into the command array */
    au8LCDWriteCommand[1] = u8Command_;

    /* Queue the command to the I2C application */
    TwiWriteData(U8_LCD_ADDRESS, sizeof(au8LCDWriteCommand),
                &au8LCDWriteCommand[0], TWI_STOP);
} /* end LcdCommand() */
```

### 12.15.2 • LcdMessage()

Loading a character message is the next obvious function to write. The function takes the LCD address for the first character in the message, and a pointer to a NULL-terminated message string. A command is queued to set the cursor to the desired address, then a message array is populated with the message characters while a character count is kept. Once all the characters have been loaded into the array, the message and its determined size are queued to the I<sup>2</sup>C app.



```
void LcdMessage(u8 u8Address_, u8 *u8Message_)
{
    u8 u8Index;
    static u8 au8LCDMessage[U8_LCD_MESSAGE_OVERHEAD_SIZE + U8_LCD_MAX_MESSAGE_SIZE] =
                                                {LCD_CONTROL_DATA};

    /* Set the cursor to the correct address */
    LcdCommand(LCD_ADDRESS_CMD | u8Address_);

    /* Fill the message */
    u8Index = 1;
    while(*u8Message_ != '\0')
    {
        au8LCDMessage[u8Index++] = *u8Message_++;
    }

    /* Queue the message */
    TwiWriteData(U8_LCD_ADDRESS, u8Index, au8LCDMessage, TWI_STOP);
} /* end LcdMessage() */
```

To make this function a little more robust, you could add checks to see if the characters available based on the addresses provided would fit the intended message. At the very least, you could truncate the message to the available space. Perhaps the function could return a value based on the success of the message or not. You could also add arguments to automatically clear the screen or the line on which the message will be displayed or change the cursor behavior. For now, compound functions will be left to the application to handle discretely.

### 12.15.3 • LcdClearChars()



Clearing characters is a common enough operation and a strange enough implementation that having an API function makes sense. This will clear a smaller group of characters from the screen instead of using the clear screen command that wipes out all display RAM. This is useful for user interfaces or for displays where values are changing but do not require refreshing the whole screen. The operation is brute force as there is no direct command for clearing a subset of characters. The function just sets the starting address, then writes ASCII spaces for the number of characters to be cleared.

```
void LcdClearChars(u8 u8Address_, u8 u8CharactersToClear_)
{
    u8 u8Index;
    static u8 au8LCDMessage[U8_LCD_MESSAGE_OVERHEAD_SIZE + U8_LCD_MAX_MESSAGE_SIZE] =
                                                {LCD_CONTROL_DATA};

    /* Set the cursor to the correct address */
    LcdCommand(LCD_ADDRESS_CMD | u8Address_);

    /* Fill the message characters with ' ' */
    for(u8Index = 0; u8Index < u8CharactersToClear_; u8Index++)
    {
        au8LCDMessage[u8Index + 1] = ' ';
    }

    /* Queue the message */
    TwiWriteData(U8_LCD_ADDRESS, u8CharactersToClear_ + 1, au8LCDMessage, TWI_STOP);
} /* end LcdClearChars() */
```

As the LCD functional requirements evolve, it may make sense to add an LCD state machine to handle more complex behavior at which point the three API functions would probably become private to the LCD application and a slightly different interface would be provided to applications that would use the LCD service.

For example, perhaps you want to provide vertical or scrolling message capability native to the LCD driver rather than leaving it up to a client application to take care of that. An application could also send text continuously to the LCD which could buffer the strings and display them systematically on screen much like the way you would expect text to appear on a monitor. The first message could be displayed for one second on line 1, the 2nd message would be displayed on line 2, and then every subsequent message would be added on line two and bump up the current line 2 message to line 1. It is hard to know exactly what functionality would be desired, so, for now, we will leave the driver in this state with the basic API.

**12.16 • Chapter Exercise**

There are so many exercises that can be done now that you have an LCD! The following suggestions are entertaining and demonstrate some great problems to solve while reinforcing what you have just learned about the LCD.

1. Write a button-controlled 4-digit counter. Always display 4 digits including leading zeroes. When BUTTON1 is pressed, increment the counter. When BUTTON0 is pressed, decrement the counter. BUTTON2 should reset the counter to 0. BUTTON3 should preset the counter to 9999 so you can easily check rollover behavior.
2. While BUTTON3 is pressed, clear the counter and write your name in the middle of the screen. Animate it by “bouncing” back and forth between the screen edges while the button is held. The character updates should occur at about 4Hz to look good. When the button is released, the screen should return to show your counter.
3. In a new user application, figure out how to load custom characters to the LCD and create Pacman and Ghost characters each with three frames of animation. Make Ghost chase Pacman across the screen and on different lines. Use BUTTON0 to pop up a power pill in a random location for a short period of time. If Pacman gets the pill, reverse the chase direction and of course invert Ghost. Do this for 5 seconds or so and then change back. Neither Ghost nor Pacman should ever catch each other, though if you wanted to build on this game go for it!





## Chapter 13 • Analog to Digital Conversion

Working with digital microcontrollers allows us to do so many things. However, the real world is analog and ultimately we need to translate between these two domains. There is a myriad of signals from sensors that provide analog information that must be digitized before processing them. Very often we need to take the digital information in a computer or embedded system and feed it back into the analog domain.

The process of converting analog to digital (A to D) and digital to analog (D to A) is carried out by converters, thus we have the acronyms ADC and DAC. Some microcontrollers have ADCs and DACs. The SAM3U2 just has ADCs which will be the focus here.

There are many different types of ADCs, each with advantages and disadvantages. Typically two ADC types are found in microcontrollers: successive approximation (SAR) type or sigma-delta type. The SAM3U2 has two versions of SAR ADCs: a standard 10-bit SAR and a 12-bit “cyclic pipeline” converter which is like a SAR with a slightly different approach to conversion.

### 13.1 • ADC background

To discuss ADCs, we need to know some terms that will appear often. The configuration, behavior, and performance of ADCs are characterized by these terms, so it is important to understand what they mean as you select and work with the converter for your application.

#### 13.1.1 • Quantization

Quantization is the act of assigning discrete levels to a continuous analog signal. This inherently adds error to the system because you are changing from a continuous-time and continuous-amplitude analog domain, to a discrete-time and discrete-amplitude digital domain. This is basically rounding up or down to a specific level at a specific time. Unless you are exactly at a level, you will always chop off a bit of information and thus add some error at that sample. This is quantization error.

#### 13.1.2 • Sampling

Analog signals can change quickly, and they have an infinite number of voltage levels as they transition. This applies to everything from single, pure, sinusoidal signals, to compound collections of tones like music. For example, think of your favorite song and concentrate on the melody and all the different instruments that work together to produce sound. How quickly does it change? How many individual tones are present at any instantaneous time?

Audible frequencies that humans can hear extend from lows of about 50Hz all the way up to around 16kHz. Other mammals have a much wider range of hearing, so their experience of life is substantially more vibrant. We can’t even imagine! See Figure 13-1 on page 452.

“Tweeters” are speakers that specialize in reproducing high-frequency sounds and they are usually rated to 22kHz. Even though that is outside the frequency response of our ears, the signals can still influence the sound we hear. This may be one of the arguments that analog purists will use when discussing the benefits of analog record players over digital CDs. MP3s are even worse due to compression. Designing an audio system for a Dolphin would be much different than for a Tuna fish – keep that in mind if you ever end up in that situation. At least getting the headphones to stay on a fish is a mechanical

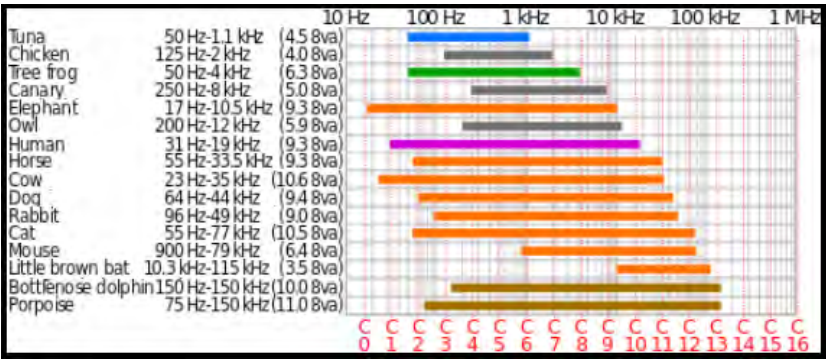


Figure 13-1 Audible frequency by species

We discuss all this because audio is the most intuitive signal to imagine coming into your ADC. If you think of just a single tone, a digital representation of that tone must include enough samples to capture an accurate digital image of the signal. If you sample too slowly, you will miss some of the detail in the signal. If you sample too quickly, you might be wasting precious power, memory, time, or combinations thereof. The ADC itself will have a physical limitation and you're not always working with audio.

13.1.3 • Bandwidth and Aliasing

Analog signals in nature are periodic (sinusoidal in the "Time domain") as shown in Figure 13-2. The "Frequency domain" representations of these signals are also shown but discussion is beyond the scope of this section. If we assume the signal source is always perfect sinusoids, then we can accurately recreate the signal by fitting a sinusoid to the samples that are collected assuming the samples occur at precisely the same time. The amplitude does not really matter since changes in volume do not affect the frequency.

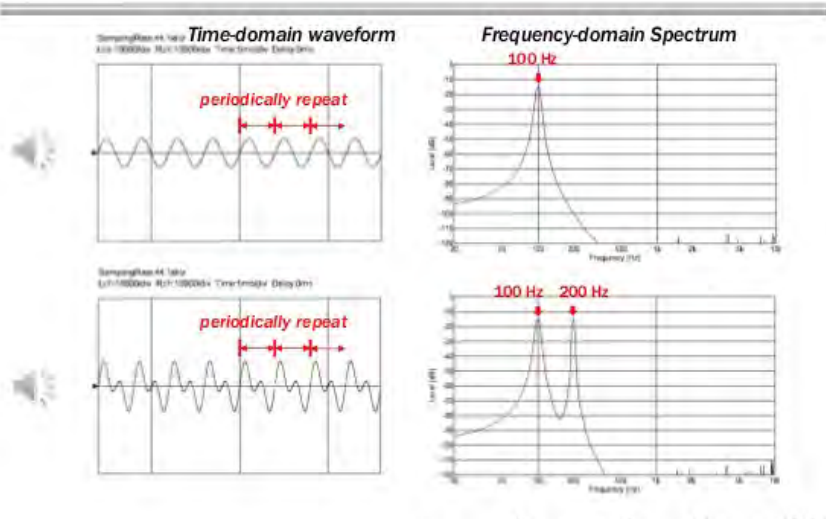


Figure 13-2 Sample 1-tone and 2-tone sounds

Time-varying analog signals in nature are periodic (sinusoidal). If we assume the signal source is always perfect sinusoids, then we can accurately recreate the signal by fitting a sinusoid to the samples that are collected assuming the samples occur at precisely the same time. The amplitude does not really matter since changes in volume do not affect the frequency.

There are infinite frequencies at higher and lower integer multiples of the original signal that would fit the sample points. Therefore, an assumption must be made about the original frequency and thus the frequency of the re-creation. The maximum signal frequency that an ADC can sample properly is called Bandwidth. That does not mean that higher frequency signals coming into the ADC will be completely invisible. The ADC will catch parts of higher frequency signals and end up capturing a signal that appears like a lower frequency. This is the essence of aliasing.

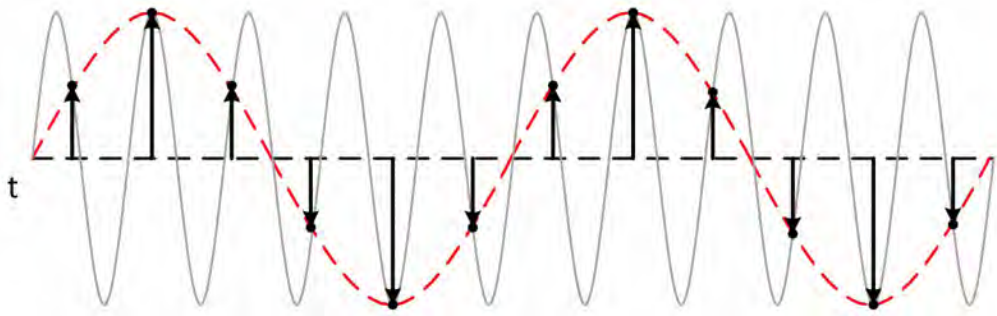


Figure 13-3 Aliasing

There will always be some interpolation required to fill in the gaps between samples. Straight-line approximations between the samples would not give great results! That being said, there are many embedded applications where you will need to sample fast enough to collect a complete picture of the signal. A good example is an oscilloscope that can sample at billions of samples per second to capture the input signals that are presented to it. Oscilloscopes still have a bandwidth specification and still must interpolate the signal to display between the samples. Even super-expensive high-end oscilloscopes are subject to aliasing just like every other analog to digital system.

#### 13.1.4 • Nyquist Frequency

So how do you know how fast you should sample to avoid aliasing? Or to put it another way, how do you specify the bandwidth of your system? It was shown by Nyquist that you must sample at 2x the highest frequency that you are trying to capture. This is called the Nyquist frequency. This is essentially why digital audio is typically sampled at 44.1kHz – about twice the frequency of the absolute highest frequencies that speakers can typically reproduce. If you're interested, take some time to research why it's 44.1kHz for CDs and 48kHz for DVDs.

With this assumption in place, then we have a limit on the maximum frequency used to recreate the signal and we are guaranteed to recreate it accurately. Realistically we cannot ever guarantee that signals and noise above the maximum frequency we're interested in are not going to be present at the ADC input. ADCs should have an "anti-aliasing" filter at the input to attempt to remove any high-frequency signals that could lead to aliasing. This is just a low pass filter with a steep roll-off at the maximum frequency of interest. If we assume that any high-frequency content is removed, then we can accurately recreate the original signal using Nyquist.

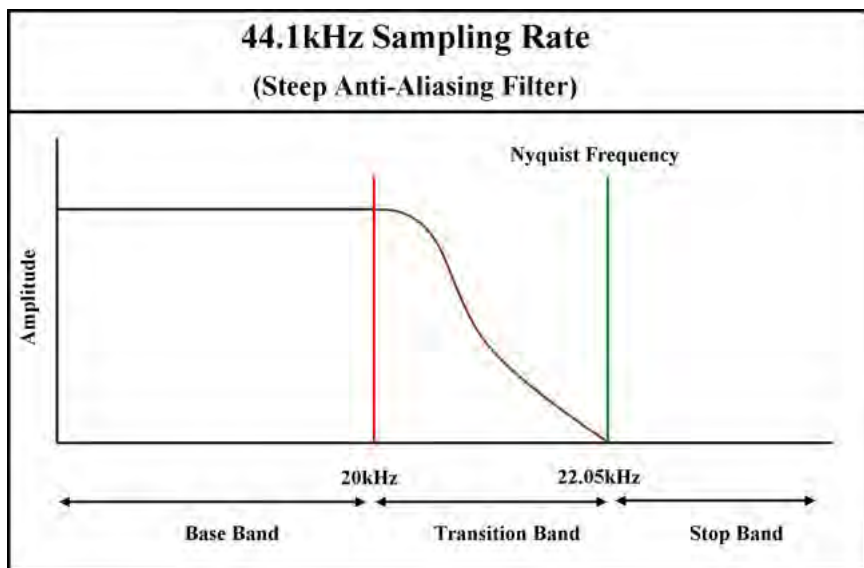


Figure 13-4 Low-pass sampling filter

In a lot of embedded applications where analog to digital measurements are required, none of the frequency-related problems will be a concern because the micro is simply measuring signals that are practically DC. Most sensors like temperature, weight, pressure, etc. are not going to change quickly and the goal of the measurement is to simply get the current level for display or perhaps trigger an alarm. Measuring battery voltage in portable systems is common, and battery voltage will always change very slowly. Such a system can still be impacted by noise so averaging and filtering are important. However, sampling rates probably only come in to play when thinking about how quickly you can possibly make a good measurement, so you can minimize the processor on time and maximize sleep time.

### 13.1.5 • Resolution

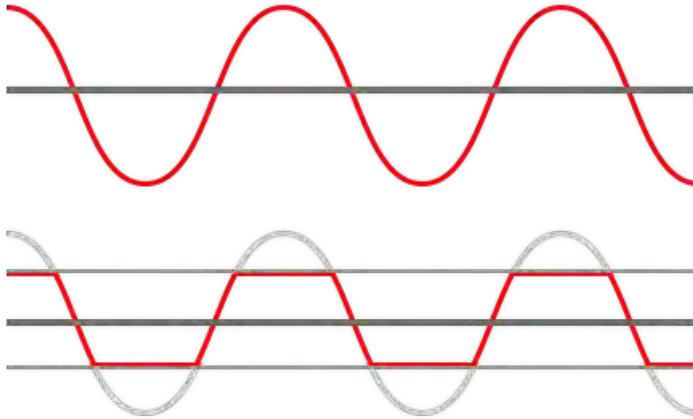
Sampling rate defines the frequencies that we can digitize, but resolution defines the accuracy and precision of each of those samples. The ADC you use will always have a minimum and maximum voltage it can detect which will, at the most, be the power supply rails of the ADC hardware. This is called the reference voltage since every ADC measurement is ultimately relative to known values. The resolution of the ADC is the number of bits that the sample is returned in. These are often called “counts.” A 10-bit ADC has 1024 “counts” (including 0) available between the low and high reference voltage. The highest value returned is  $2^{10} - 1 = 1023$ . If the low reference is 0V and the high reference is 3.3V, then each “count” represents  $3.3V / 1024 \text{ counts} = 3.223mV / \text{count}$ . For a 12-bit converter, each count is  $3.3V / 4096 \text{ counts} = 0.8mV / \text{count}$ . With more bits, you can resolve more differences in the signal.

Engineers always get the tough questions like, “how much is enough?” In the context of analog to digital conversion, this could refer to the number of bits that are required to get an accurate representation of the signal. Microcontrollers often come with 10-bit converters and some have 12-bit ADCs. Their speed usually depends on the clock speed of the MCU itself up to some maximum value that is limited by the peripheral. In a lot of cases, this is more than enough resolution and speed, especially for sensor applications and audio. If more is needed, there are a lot of external 24-bit ADCs, some of which can

run ultra fast. Like so many answers in the engineering world, the answer to “how many bits do you need?” is, “it depends.”

### 13.1.6 • Clipping

If the voltage level of the signal into the ADC is higher than the system voltage, the ADC saturates at its peak value and you get a phenomenon called “clipping.”



**Figure 13-5** Example of clipping

In the audio world, reproducing a clipped signal (or clipping in a purely analog signal chain) sounds like a loud pop. From a signal perspective, it’s even worse as not only are you missing the true content of the signal, but you get substantial noise from harmonics that can really distort the system.

Slightly overdriving the ADC input on a microcontroller is physically ok if you do not mind the saturated data points for any voltage above the maximum count of the ADC. If the voltage goes too high, then you will activate the protection diode on the input pin and potentially damage the signal source, the microcontroller, or both.

### 13.2 • Characteristics of ADCs

Now that you know some language, we can start looking at some characteristics of a microcontroller ADC. The first and most obvious is the resolution. Beyond the resolution, you must read the processor datasheet to determine how to properly use the ADC based on the physical limitations of the hardware.

ADCs have something called “sample and hold time” specifications. The sample time is how long the signal must be present to ensure that an accurate representation of the voltage being measured has a chance to charge up a capacitor so that a measurement can be made on a stable signal. The ADC literally has a little switch that connects the outside world to the sampling capacitor. Once the sampling time is done, the switch opens and the voltage will stay on the capacitor because there is nowhere for the charge to go. The circuit is specially designed to have very, very low leakage. The figure below is taken directly from the SAM3U user guide and shows the basic view of the hardware involved.

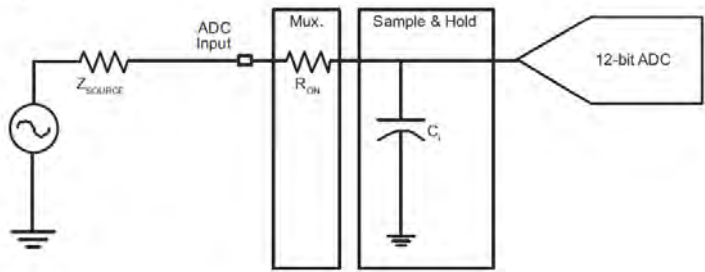


Figure 13-6 Simplified acquisition path

The hold time is the amount of time that the voltage charge must remain on the sample and hold capacitor while the conversion takes place. The digitization is a function of the clock that is running the ADC peripheral. The faster the clock, the faster the conversion with a trade-off of power. Together these specs define the absolute maximum speed at which the ADC can sample and thus the bandwidth of the ADC.

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
$f_{ADC}$	ADC Clock Frequency		1		20	MHz
$t_{CP\_ADC}$	ADC Clock Period		50		1000	ns
$f_S$	Sampling Frequency		0.05		1	MHz
$t_{START}$	ADC Startup Time	From OFF Mode to Normal Mode: - Voltage Reference OFF - Analog Circuitry OFF	20	30	40	$\mu s$
		From Standby Mode to Normal Mode: - Voltage Reference ON - Analog Circuitry OFF	4	8	12	
$t_{TRACK}$	Track and Hold Time	See <a href="#">Section 42.7.2.1 "Sample and Hold Time versus Source Output Impedance"</a> for more details	160			ns
$t_{CONV}$	Conversion Time			20		$t_{CP\_ADC}$
$t_s$	Settling Time	Settling time to change offset and gain	200			ns

Figure 13-7 SAM3U datasheet ADC specs

### 13.2.1 • Precision, Error, and ENOB

Every ADC will have variances and tolerances in its design. This is a function of the ADC itself and also the signal source and any external components in the input path. These errors will add up to give an overall expected accuracy which equates to an “effective number of bits” or ENOB. ENOB is often specified like “2 LSBs” which means the lowest 2 bits can be considered noise in the system.

In a lot of applications where embedded systems are reading sensors, the absolute accuracy is not immensely important but rather relative differences are what you’re concerned with. You may use basic (or some more complex) averaging algorithms to smooth out the noise and you may not even care about the lower few bits of resolution.

If you need perfectly accurate measurements, you may need to calibrate the ADC at least once in the factory. Some systems need regular calibration especially if the hardware in the signal chain is known to change its response over time, like a gas sensor that will become less sensitive over its life. Once more, it’s up to the engineer to decide what is necessary to ensure acceptable results. In the case of calibration, there are many factors

to consider beyond just the technical problem. Customers will not be very happy if they have to manually calibrate their equipment often, especially if it is in an unmanned, remote location.

### 13.2.2 • Missing codes

ADCs are never perfectly linear, and in some cases, the ADC hardware is incapable of ever returning certain results. These are called “missing codes.” Most of the ADCs in microcontrollers will claim to have “no missing codes.” If that matters, you need to check carefully and buy an MCU with a better ADC or consider an external ADC which will give you a lot more options.

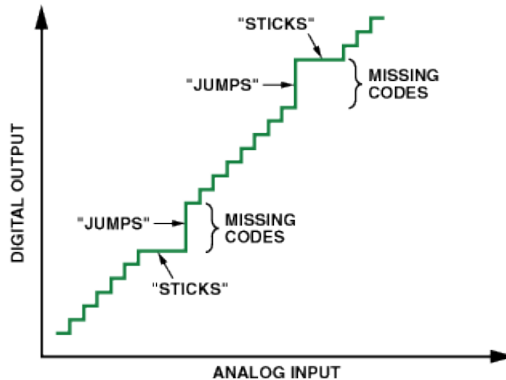


Figure 13-8 Missing codes

### 13.2.3 • Reference Voltages

For an ADC to work, it must have a known voltage level to compare the voltage of the input signal against. This is called the reference voltage. In the most basic configuration, a processor will use 0V (system ground) for the low reference signal and whatever its  $V_{CC}$  is for the high reference signal.

For better results, you can add an external precision reference voltage for the ADC. These are typically “band-gap” references and have voltages around 1.2V or 2.5V. They are very precise to being with, and because their voltage is quite a bit lower than the main supply rail, they are not subject to variations that may appear on  $V_{CC}$ . There should be a configuration bit in the ADC peripheral to tell the processor to use an external reference. Some microcontrollers have built-in references that can be enabled.

The downside of using a lower voltage is that some dynamic range is lost. Dynamic range is the total range in the signal from high to low. More is generally better. External references also cost money, power, and board space. If you’re a hardware designer and not quite sure if you’ll need an external reference, make sure you put the footprint on the board and connect it properly! You can always not populate it and use  $V_{CC}$ , but it’s there if needed.

### 13.2.4 • Noise

Even in almost-DC systems, analog to digital conversions will be influenced by noise. Power supplies in digital systems are notoriously noisy and very often have transients



of tens or even hundreds of mV as different parts of the circuits operate. If you use the power supply rails for your ADC reference, the readings can be significantly incorrect if you sample right at the time a transient is occurring. Most MCUs will have a separate analog power supply pin, so you can add some heavy filtering here to clean up the Vcc rail before it goes into the microcontroller. The ADC itself draws very little current, so with a large resistor and capacitor, you can create a pretty stable supply voltage.

Noise is something you can also address in firmware by choosing a quiet time in the program loop to read the ADC if your system is sufficiently deterministic. You can average readings which is the quickest way to smooth out noise as long as you can still sample fast enough since your sampling requirements will scale with the number of averages per reading. A quick hint: use a power of 2 for the number of samples so that the division is just bit shifting. Make sure that the variable that is accumulating the readings is large enough so even if all the readings are full scale the variable will not overflow. You need to sum up all the readings before doing any divisions to avoid massive division errors in an integer-based processor.

If you know that noise on Vcc is going to be a problem to use it for a reference voltage, you can add an external precision reference voltage for the ADC. These are typically “band-gap” references around 1.2V or 2.5V that are very, very precise even though they are powered from noisy supply rails. If the reference’s voltage is always below the noisy transients, then there will be no variation in the returned ADC results despite the power supply noise. There should be a configuration bit in the ADC peripheral to tell the processor to use the external reference. If you’re a hardware designer and not quite sure if you’ll need an external reference, make sure you put the footprint on the board and connect it properly! You can always not populate it and use Vcc, but if you do need it you can just put it on and avoid a board spin. Some microcontrollers have built-in references that can be enabled. The downside of using a lower voltage is that some dynamic range is lost. Dynamic range is the total range in the signal from high to low. More is generally better.

### 13.2.5 • Single vs. Differential Measurement

In most cases, a single signal is measured and digitized per the references used by the ADC. Another way to get ADC readings is to use differential input, where two signals are compared, and the ADC returns the difference. This is often a good way to deal with power supply noise since both signals should experience the same noise and thus cancel each other out. Many systems are inherently single-ended so this is not an option. Dealing with noise is left to filtering and the quality of the reference voltage. Not all processors offer differential measurement in hardware, but the SAM3U2 does on a pair of adjacent channels.

### 13.2.6 • Signal Conditioning

We have talked about noise and input filters which are an important part of the design to remove unwanted frequencies. The amplitude of the signal must also be considered. In a lot of applications, an input limit of 3.3V (or whatever your supply rail is) is not practically useful. For example, in a battery-powered device, the battery voltage will always be higher than the regulated Vcc rail unless the device uses a boosting power supply. Therefore, it is not possible to measure the battery voltage directly without saturating the ADC.

To measure voltages that will exceed the supply rail, you need to scale the input down to fit inside the reference voltage range you have in hardware. You can do this easily with a resistor divider, though the absolute accuracy is at the mercy of the tolerance of the resistors you use. The answer is usually not just “use tighter tolerance resistors” because

even the ADC itself has tolerance along with many other factors. 0.1% tolerance resistors are expensive compared to 1% and this can add up significantly in low cost / high volume products.

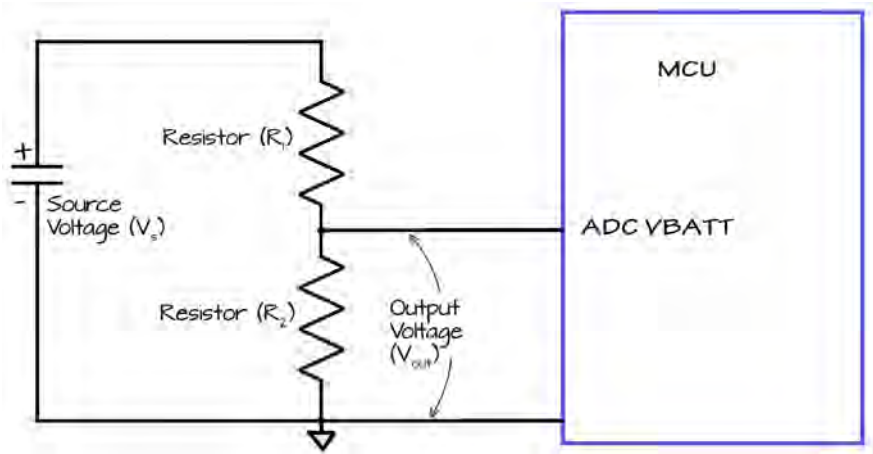


Figure 13-9 Signal conditioning

You also must consider the overall impedance of an input resistor divider and make a design trade-off with your conversion time, load to the analog circuit, and even power consumption. The resistor divider will consume some amount of current. You need the signal source being measured to see a very high impedance or else your resistor divider will load and disrupt it. However, your ADC wants low impedance, so it can quickly charge and discharge the sample and hold capacitor.

Turn to the datasheet for help. The table below shows the maximum source impedance values for the 12-bit ADC both for 12-bit accuracy and 10-bit accuracy based on different ADC clock speeds. Faster and more accurate implies lower source impedance.

$f_{ADC}$ = ADC clock (MHz)	$Z_{SOURCE}$ (k $\Omega$ ) for 12 bits	$Z_{SOURCE}$ (k $\Omega$ ) for 10 bits
20.00	10	14
16.00	14	19
10.67	22	30
8.00	31	41
6.40	40	52
5.33	48	63
4.57	57	74
4.00	66	85

Figure 13-10 Source impedance values

You can also add hardware filters to help reduce noise which is generally better than any software filtering or averaging that you can do in a standard microcontroller. High amplitude noise could saturate the input, and high-frequency noise can lead to aliasing especially if the noise source is periodic.



To check your understanding, consider the following system:

- 12-bit ADC
- 0V to 2.5Vref

- Input signal resistor divider with 200k and 100k resistors

What is the current load on the signal being measured? What is the maximum ADC clock rate you can set? If your ADC returns 896 counts, what is the actual signal voltage?

### 13.3 • EiE ADC Hardware

The EiE development boards have two ADC channels available at the Blade connector that you can connect analog signals into. Both channels are on the 12-bit converter with AN0 going to channel 2 and AN1 to channel 3. The boards use Vcc as the reference. The signal lines attach directly to the MCU's ADC inputs, so be sure to properly scale any voltage that would exceed the 3.3V Vcc supply rail before wiring into the Blade connector.

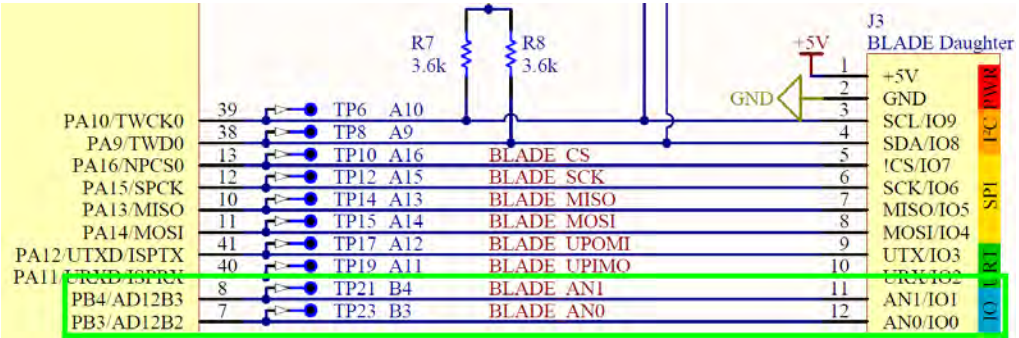


Figure 13-11 Wiring into the Blade connector

There is a third analog input on the ASCII development boards that have a trim pot attached so you can set any voltage from 0V to 3.3V for testing purposes. It connects to Channel 1 on the processor. A good first test is to set this value and display the number of counts that are read without any filtering. Check to see if you can get to 0 and if you can get to full scale. Also, compare the absolute measurements with a multimeter to get an idea about the accuracy. Experiment with the sampling speed and averaging.

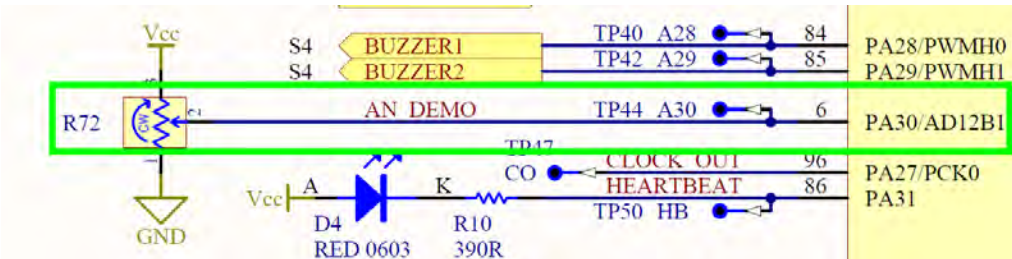


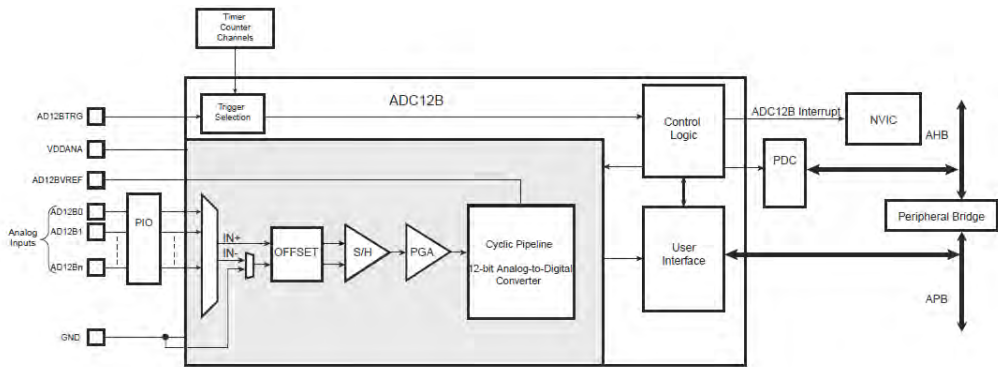
Figure 13-12 Comparing absolute measurement

You could also use the breakout pins to connect to other channels, though you would have to look at the circuits that are attached. There are three 10-bit ADC channels available on unused pins on the ASCII board. AD12B1 is available on the Dot Matrix board.

### 13.4 • SAM3U2 12-bit ADC Peripheral

With the hardware connections understood, we can look at using the ADC peripheral to read the signals. As with any peripheral, start by reading the whole section of the User

Guide to get a feel for what is involved in using the peripheral. The guide starts out with a bird's eye view of the hardware.



**Figure 13-13 ADC peripheral block diagram**

There is an analog supply input, a reference voltage input, a local ground, and multiple input channels. There's some sort of offset hardware, the sample and hold buffer and even a gain stage that might be useful for lower signal values.

ADCs typically need to be “triggered” to start a conversion. We can see that an external trigger or trigger from a counter is available. The user guide points out that inside the control logic is a software trigger, as you would expect. There's an interrupt that most likely is the “finished converting” flag. The ADC has DMA access and direct access through the peripheral registers.

The first few sections of the peripheral description are straightforward but still important for properly configuring the ADC. The section on Differential Inputs is significant as there are some hardware implications here. In our application, only single-ended mode is used. From reading the whole section, the most important details are:

- There is a power management bit that must be set to enable the ADC clock
- The analog and trigger lines must be set to the appropriate peripheral control during PIO configuration
- In addition to the sample and hold times required, 10 ADC clock cycles are required for the conversion. This will be important to determine if it is possible to simply block and wait during an A to D conversion, or if a wait state would be required.
- A gain stage is available in hardware. While there are only options of x2 and x4 for gain in a single-ended system, this could be helpful in some basic situations with a low amplitude signal. Amplifying the signal effectively increases your resolution, though it does not improve the signal-to-noise ratio since the noise is amplified with the signal.
- An offset is also available to shift the signal. This would decrease the overall dynamic range of the signal since you are still limited to  $V_{ref}$ , but it could help to avoid clipping on the low side if you have an input signal that can go slightly negative. The datasheet implies that the input is still range-bound to the offset less half the signal. If you wanted to use this some more investigation would be required on how it really works. While you might be tempted to think that you could measure “negative voltage signals” that is untrue since the input into the microcontroller pin is still limited by the Absolute Maximum Ratings which are -0.3V on the low side.

- Two bits are set when a conversion is complete, EOC and DRDY. Both can trigger the ADC interrupt.
- The ADC result is stored in two registers, CDR and LCDR. There is a CDR for every channel, so 8 in total. There is a single LCDR that stores the result of whatever the last channel was that completed.
- Reading a CDR clears the corresponding EOC bit; Reading LCDR clears DRDY.
- Overrun bits OVRE and GOVRE get set if a conversion is done when EOC or DRDY, respectively, are still set. These bits are cleared when the ADC status register is read.
- Starting a conversion in software is done by setting the START bit in the ADC control register, CR. Conversions are done automatically on all the enabled channels. Though the datasheet does not specify, we can assume that the sequence is from low to high if that matters for the application. If you want to convert more than one channel at a time, make sure you check the EOC bit for each channel to know it has completed.
- There are additional features for power control and channel sequencing that may be of interest.

#### 13.4.1 • ADC Registers

All register names for the 12-bit ADC are prefixed with ADC12B\_. From the AT91SAM3U4.h header file, the base address for the peripheral is AT91C\_BASE\_ADC12B and the register struct is AT91PS\_ADC12B. Most of the peripheral registers will be required to set up and use the ADC12. For some reason, the interrupt handler for the 12-bit ADC is called ADCC0.

For the EiE development board, we choose to run a 1MHz ADC clock, so values selected for register initialization are based on that.

**Control Register (ADC12B\_CR):** CR has basic functions to run and reset the ADC. Only two bits are here, the most important being the START bit that is used by firmware to begin conversion on the selected channel (or sequence of channels). There is also a peripheral reset bit, SWRST. It might be good to reset the ADC during initialization, but not necessary. We will not use this register during initialization.

**Analog Control Register (ADC12B\_ACR):** Just 6 bits in this register are required to control the analog features of the ADC such as the gain and offset values. The DIFF bit is set if the ADC runs in differential mode. GAIN is set with 2 bits for 4 different options which are different for single and dual-ended modes. OFFSET is a single bit and only applies for single-ended configurations.

There is also 2 bits to control bias current for the ADC peripheral. This is discussed in the Power Consumptions Adjustment part of the peripheral. The ADC can have better performance and sample more quickly if you give it more current. If you look closely, there is a note that you must use the “typical” setting (01) for sampling rates above 500kHz to 1MHz. There does not seem to be any comments about the other settings which leaves some decision up to the application and perhaps some testing. If you aren’t concerned with power consumption, max it out to ensure you are not adding any complications due to an underpowered ADC. In a product – especially one with limited power – you would explore this carefully to choose the optimum setting. The differences are in mA, which is substantial.

In the default application, no gain or offset is required, so both GAIN and OFFSET are 0. We’ll set IBCTL to ‘01’ for “typical” operation since the datasheet says this is sufficient for

1MHz ADC clock. The channels are Single Ended, so DIFF is 0. The resulting configuration word is:

```
#define ADC12B_ACR_INIT (u32)0x00000101
```

**Mode Register (ADC12B\_MR):** The mode register holds most of the configuration for the ADC12 peripheral. Trigger selection is done here. The hardware triggers can be completely disabled, or enabled and then must be selected using 3 bits. 12-bit vs. 10-bit resolution, and Normal vs. Sleep modes are configured. The ADC clock and timing settings are configured here, also. Follow the calculations carefully and choose the correct values for the parameters. These should be set when the ADC is first initialized.



If MCK is 48MHz, find PRESCAL to get a 1MHz ADC clock. Then determine STARTUP, and SHTIM to ensure proper conversion at the fastest rate possible. How long will each conversion take? Can the system block during this time and still meet the 1ms system loop requirements, or is a wait state required after the conversion is started?

For the EiE system,  $1\text{MHz ADC clock} = 48\text{MHz} / ((\text{PRESCAL} + 1) * 2)$ , so PRESCAL is 23. We want to maximize the sample and hold time so that that maximum input impedance is available, so SHTIM is set to 0xF. This means SHTIM is  $15 / 1\text{MHz} = 15\mu\text{s}$ . We will choose the start-up time as the maximum value for safe operation regardless of if the system is run in Standby Mode or Sleep mode. From the Channel Conversion and ADC Clock table in the Electrical Characteristics section of the user guide, the maximum start-up time in any mode is  $40\mu\text{s}$ . Therefore, we solve  $40\mu\text{s} = (\text{STARTUP} + 1) * 8 / 1\text{MHz}$  which gives  $\text{STARTUP} = 4$ .

We also want to run in Normal mode, use 12-bit resolution, and not configure any hardware triggers. This gives our final initialization value in adc12.h as:

```
#define ADC12B_MR_INIT (u32)0x0F041700
```

**Extended Mode Register (ADC12B\_EMR):** This register is used to enable the low power “Off” mode. Off mode is selected by setting the OFFMODES bit. The start-up time required when waking up from Off mode is programmed in 8 bits OFF\_MODE\_STARTUP\_TIME based on the calculation shown in the user guide.

Off mode can save power at the cost of more time to return the result because the ADC will wait for the programmed start-up time before converting the channel. The start-up time is clocked automatically based on the STARTUP value in EMR, so the user still simply waits for an EOC bit to be set. In a system where the processor is blocked while conversions occur, using Off mode could have significant impacts. Calculations should be made carefully, and worst-case scenarios should be accounted for or at least documented if there is a risk of disrupting the expected system timing.

The ADC consumes a lot of current when it is running so we will use Off mode. The value of OFF\_MODE\_STARTUP\_TIME is the same calculation used for the main STARTUP time, which is 4. With this and OFFMODES = 1, EMR is initialized to:

```
#define ADC12B_EMR_INIT (u32)0x00040001
```

**Status Register (ADC12B\_SR):** The status register holds all bits related to what has happened as a result of conversions in the ADC peripheral. The first 8 bits are the individual End of Conversion (EOC) bits for each of the 8 channels. The next 8 bits are the Overrun Error bits also per channel. There are two bits that monitor if any conversions have been completed since last time a result register was checked (DRDY) and another bit (GOVRE) that is set if any channel has overrun since the last time the status register was set. If using peripheral DMA for data transfers, the ENDRX and RXBUFF bits provide



status on the RCR and RNCR registers. SR is read-only and does not require initialization.

**Channel Enable / Disable / Status Registers (ADC12B\_CHER / ADC12B\_CHDR / ADC12B\_CHSR):** Channels used in hardware should be enabled here. Depending on your application, you may want to disable a channel that is not used to try and save power or time if you do not need a particular reading each time you read the ADC. Enabled channels are measured and converted once the START bit is set.

If you want to read a subset of the available channels each time the START bit is set, simply disable the channels you don't want. This register might be used often during program execution, therefore an API function would be a good idea to allow tasks to select the analog channel they are interested in.

In EiE, we will work with one channel at a time in the API, so at initialization all channels are disabled. We only need to write CHDR to configure this.

```
#define ADC12B_CHDR_INIT (u32)0x000000FF
```

**Last Converted Data Register (ADC12B\_LCDR):** This register holds the conversion result for whichever channel was converted most recently. LDATA is the 12-bit result and is right-justified to the register's LSB. There is no information about what channel the result belongs to, but there are many scenarios where that information would not be necessary and having a single register to read results is convenient.

You may ignore LCDR and the GOVRE bit in SR if you do not want to use this register and would prefer the individual channel result registers.

**Interrupt Enable / Disable / Mask Registers (ADC12B\_IER / ADC12B\_IDR / ADC12B\_IMR):** Interrupts are available for any of the status bits in the ADC12 SR. The IER/IDR/IMR register bits correspond to all the status bits in the SR register. Any flag bit that the ADC maintains can also trigger the overall ADC interrupt as they would all be ORed together to a single input to the NVIC.

There is no explicit indication about how to clear interrupt flags, so we can assume clearing the associated SR bit would also lower the interrupt flag bit. In other words, the interrupt flag bits are the SR bits themselves.

**Channel Data Registers 0 to 7 (ADC12B\_CDR0 to ADC12B\_CDR7):** Each channel has its own result register that is populated when the conversion is complete. The 12-bit result is right-justified in the lower 12-bits of each CDR. The result remains in the register even after it is read. The EOC bit could be used to check if the result has already been read. When the ADC writes a CDR register, the corresponding EOC bit is set. When the application reads the CDR register, the EOC bit is cleared.

### 13.5 • EiE ADC Driver

The driver task for the 12-bit ADC will set up the peripheral to perform individual channel measurements. To keep the task abstracted from other tasks, the driver will interrupt when the conversion is complete and pass the result to the task which initiated the conversion through a call-back function that the client will provide. A binary semaphore will be used to lock the peripheral when it is in use since only one conversion can take place at a time. Management of the lock will be handled entirely by the driver function.

The intent will be for the user to request a conversion on a specific channel. If the ADC task is available, the conversion will start, and the calling task can simply wait until its call-back function has run. It is up to the call-back function to ensure its task knows that a result has been provided.



Since the interrupt must pass the ADC result to the call-back, a new function pointer typedef is required in typedefs.h that includes a 16-bit parameter.

```
/* Type name for function pointer with one u16 argument */
typedef void(*fnCode_u16_type)(u16 x);
```

If the ADC task is busy, the request for conversion will fail and the task can try again later. The most likely problem to arise might be that a task that wants to use the ADC is continuously denied because another task repeatedly restarts a conversion immediately after one is complete. If starting another conversion is done in the call-back function, then the task would essentially have 100% control of the ADC. We can make a rule (or suggestion) for user tasks to avoid requesting the next conversion in the call-back function. New conversions should only be started within the task. This would allow a program cycle to run and give other tasks the opportunity to request a conversion.

Doing this also makes sense because if the system is running quickly the chance of overwriting a result in the task's buffer would be quite high. The downside here is that averaging  $n$  samples will take a minimum of  $n$  milliseconds due to the system loop time.



The ADC source files and main.c function calls are already in the project. Open adc12.h and notice the definition of Adc12ChannelType that enumerates all the available ADC channels.

```
/*!
@enum Adc12ChannelType
@brief Controlled list of available ADC channels used in the member functions.
*/
typedef enum {ADC12_CH0 = 0, ADC12_CH1 = 1, ADC12_CH2 = 2, ADC12_CH3 = 3,
              ADC12_CH4 = 4, ADC12_CH5 = 5, ADC12_CH6 = 6, ADC12_CH7 = 7}
Adc12ChannelType;
```

This enum provides some abstraction from the hardware and enables self-describing values. These are purposely sequentially numbered with the functional purpose of providing bit-shift offsets to the corresponding channel bits in the various ADC peripheral registers. You might have noticed that all of the channel bits in the registers are organized this way. To support an unknown number of implemented channels on a specific target device, we will build an array of the available channels on the dev board. Add the following section with channel definitions and board-specific array to eief1-pcb.h after the Buzzer segment.

```
/*-----
%ADC% Analog input channel Configuration
-----*/
Available analog channels are defined here. The specific channels are equated to
the general channel definitions in adc12.h. ADC_CHANNEL_ARRAY must include all the
available channels but does not have to be sequential.
*/
#define ADC12_POTENTIOMETER ADC12_CH1
#define ADC12_BLADE_AN0 ADC12_CH2
#define ADC12_BLADE_AN1 ADC12_CH3
#define ADC_CHANNEL_ARRAY {ADC12_POTENTIOMETER, ADC12_BLADE_AN0, ADC12_BLADE_AN1}
```

With these definitions, we can create a generic channel definition array in adc12.c and prepare for an associated array of call-back functions. Regardless of the target device, the array will adjust properly when the code is compiled. The two arrays are global to the adc12 task and we can add the binary semaphore for the access control at the same time in the adc12 global definitions.



```

/*****
Global variable definitions with scope limited to this local application.
Variable names shall start with "Adc12_<type>" and be declared as static.
*****/
static fnCode_type Adc12_pfnStateMachine; /* The state machine function pointer */

/* Available channels defined in board-specific header file */
static Adc12ChannelType Adc12_aeChannels[] = ADC_CHANNEL_ARRAY;

/* ADC12 ISR callback function pointers */
static fnCode_u16_type Adc12_apfCallbacks[U8_TOTAL_ADC_CHANNELS];

static bool Adc12_bAdcAvailable; /*!< ADC binary semaphore to control access */

```

With only 8 channels and where each channel is simply a single number, the memory footprint of setting up the driver this way is minimal while providing a clean and flexible abstraction.

### 13.5.1 • ADC Initialization

Adc12Initialize needs to write the INIT values for MR, CHDR, ACR, EMR and IDR registers. The call-back function array should be initialized to something safe, the semaphore should be made available, and the ADC interrupt should be enabled.



First define a private default call-back function in adc12.c that can be assigned to the array during initialization.

**void Adc12DefaultCallback(u16 u16Result\_)**

```

void Adc12DefaultCallback(u16 u16Result_)
{
    /* This is an empty function */
    DebugPrintf("\n\rDefault ADC call-back!\n\r");
} /* end Adc12DefaultCallback() */

```



Now write Adc12Initialize. Don't forget to add a debug message to indicate the driver is ready. Add `_APPLICATION_FLAGS_ADC` to `G_u32ApplicationFlags` and set the ADC function pointer to Idle. For now, there are no cases where the ADC initialize would fail. Think through this all and write the code, then compare with our solution.

**void Adc12Initialize(void)**

```

{
    u8 au8Adc12Started[] = "ADC12 task initialized\n\r";

    /* Initialize peripheral registers. ADC starts totally disabled. */
    AT91C_BASE_ADC12B->ADC12B_MR = ADC12B_MR_INIT;
    AT91C_BASE_ADC12B->ADC12B_CHDR = ADC12B_CHDR_INIT;
    AT91C_BASE_ADC12B->ADC12B_ACR = ADC12B_ACR_INIT;
    AT91C_BASE_ADC12B->ADC12B_EMR = ADC12B_EMR_INIT;
    AT91C_BASE_ADC12B->ADC12B_IDR = ADC12B_IDR_INIT;

    /* Set all the call-backs to default */
    for(u8 i = 0; i < (sizeof(Adc12_apfCallbacks) / sizeof(fnCode_u16_type)); i++)
    {
        Adc12_apfCallbacks[i] = Adc12DefaultCallback;
    }
}

```

```

/* Mark the ADC semaphore as available */
Adc12_bAdcAvailable = TRUE;

/* Check initialization and set first state */
if( 1 )
{
    /* Enable required interrupts */
    NVIC_ClearPendingIRQ(IRQn_ADCC0);
    NVIC_EnableIRQ(IRQn_ADCC0);

    /* Write message, set "good" flag and select Idle state */
    DebugPrintf(au8Adc12Started);
    G_u32ApplicationFlags |= _APPLICATION_FLAGS_ADC;
    Adc12_pfnStateMachine = Adc12SM_Idle;
}
} /* end Adc12Initialize() */

```



Make sure your solution is going to work the same. Build and debug the code. Pay attention to how the ADC peripheral registers get loaded and confirm the call-back array is properly initialized to the default function.

### 13.5.2 • ADC Interrupt

The ADC will trigger an interrupt when a conversion is complete. Even though we don't have code to start a conversion yet, we can write the ISR to handle it. This is where the channel constant numbering will be used. The end of conversion (EOC) bit for the channel that finished will be set. The EOC bits are in the 8 LSBs of the SR register. Therefore, a loop can be used to look through the channel array and shift a bit mask to the channel's bit number. When a set bit is found, the corresponding result register can be read and then passed to the task by indexing the call-back array to get the correct function. Since the driver is designed for only one conversion at a time, the bit that is found should be the only bit that is set.

From the User Guide, we know that reading CDR clears the EOC flag (which is also the acting peripheral interrupt flag). Don't forget that the debugger will clear the flags if you are looking at CDR and halting before the ISR firmware reads it.

When the call-back returns, the channel should be disabled. As the ISR exits, give the semaphore back since the ADC is now available for the next conversion. Releasing the semaphore after the call-back enforces our rule that tasks should not be trying to start another conversion in their call-back function.

```

void ADCC0_IrqHandler(void)
{
    u16 u16Adc12Result;

    /* Check through all the available channels */
    for(u8 i = 0; i < (sizeof(Adc12_aeChannels) / sizeof(Adc12ChannelType)); i++)
    {
        if(AT91C_BASE_ADC12B->ADC12B_SR & (1 << Adc12_aeChannels[i]))
        {
            /* Read the channel's result register (clears EOC bit / interrupt) */
            u16Adc12Result = AT91C_BASE_ADC12B->ADC12B_CDR[Adc12_aeChannels[i]];
            Adc12_apfCallbacks[Adc12_aeChannels[i]](u16Adc12Result);

            /* Disable the channel and exit the loop since only one channel can be set */
            AT91C_BASE_ADC12B->ADC12B_CHDR = (1 << Adc12_aeChannels[i]);
            break;
        }
    }
}

```

```

    }
}

/* Give the Semaphore back, clear the ADC pending flag and exit */
Adc12_bAdcAvailable = TRUE;
NVIC->ICPR[0] = (1 << AT91C_ID_ADC12B);
} /* end ADCC0_IrqHandler() */

```

### 13.5.3 • ADC State Machine



Our simple ADC task does not need to do anything in the Idle state. Brainstorm some ways that the task could be improved. For example, the task could keep track of the number of conversions on each channel and/or the number of rejected ADC requests. These could be output as debug messages. It could also ensure that conversions are taking place quickly or not getting stuck. The results could be improved by automatically averaging several samples. To ensure flexibility, the number of samples in each average could be set in an API function so the client task has control.

## 13.6 • EiE ADC API

The private driver functionality is complete, so all that remains are two public functions to allow tasks to set their call-back function and trigger the conversion on the channel of interest. We assume that different tasks will be interested in different channels. Right now, the semaphore will only protect multiple accesses to starting a conversion, so tasks could change the target call-back function even though they do not have control of the peripheral. This could be solved by adding a separate peripheral request function like in the cases of the communications peripherals. If the semaphore was controlled at that level, then `Adc12AssignCallback` could be updated to check if the task had control and prevent changing the function if not.

### 13.6.1 • `void Adc12AssignCallback( )`



Design the call-back assignment function to take the ADC channel and call-back function pointer as parameters. Load the function pointer into the call-back array at the correct index to overwrite the default. The header is shown for reference.

```

/*!-----
@fn void Adc12AssignCallback(Adc12ChannelType eAdcChannel_,
                             fnCode_u16_type pfUserCallback_)

@brief Assigns call-back for the client application.

This is how the ADC result for any channel is accessed. The call-back function
must have one u16 parameter where the result is passed. Define the function that
will be used for the call-back, then assign this during user task initialization.

Different call-backs may be assigned for each channel.

*** To mitigate the chance of indefinitely holding control of
the ADC resource, new conversions shall not be started in this call-back. ***

Requires:
@param eAdcChannel_ is the channel to which the call-back will be assigned
@param pfUserCallback_ is the function address (name) for the user's call-back

Promises:

```

```
- Adc12_fpCallbackCh<eAdcChannel_> ADC global value loaded with pfUserCallback_
*/
```

In our solution, the requested channel is verified to ensure it is one of the allowed channels for the development board. We can't validate if the task has requested the correct channel number, but it is hoped that the channel name enumerations would make that obvious to the programmer.

```
void Adc12AssignCallback(Adc12ChannelType eAdcChannel_, fnCode_u16_type pfUserCallback_)
{
    bool bChannelValid = FALSE;

    /* Check to ensure the requested channel exists */
    for(u8 i = 0; i < (sizeof(Adc12_aeChannels) / sizeof (Adc12ChannelType)); i++)
    {
        if(Adc12_aeChannels[i] == eAdcChannel_)
        {
            bChannelValid = TRUE;
        }
    }
}
```

With a valid channel, the channel parameter indexes the call-back function array to load the pointer at the correct index. If the channel is not valid, a debug message is printed but no other action is taken.

```
/* If the channel is valid, then assign the new call-back function */
if(bChannelValid)
{
    Adc12_apfCallbacks[eAdcChannel_] = pfUserCallback_;
}
else
{
    DebugPrintf("Invalid channel\n\r");
}

} /* end Adc12AssignCallback() */
```

### 13.6.2 • bool Adc12StartConversion()

The last function is used to start the conversion and only needs the channel name as a parameter. The function returns TRUE if the ADC peripheral is available, otherwise, it returns FALSE. This leaves it up to the task to retry at another time if the conversion request fails because the ADC is busy. If the peripheral is available, the function can proceed to start the conversion. Since the conversion will start, the peripheral is now busy.

This semaphore is only taken in this function, and this function is not accessed via interrupts. Therefore, it is safe to modify the semaphore with interrupts enabled. If a counting semaphore was used, interrupts would need to be disabled around this.



The channel parameter is used to enable the correct channel, enable the end of conversion interrupt, and then start the conversion. Remember that the parameter corresponds to the bit shift required to hit the correct bit for the channel of interest. Try to code the function then check out the solution.

```

bool Adc12StartConversion(Adc12ChannelType eAdcChannel_)
{
    if(Adc12_bAdcAvailable)
    {
        /* Take the semaphore so we have the ADC resource */
        Adc12_bAdcAvailable = FALSE;

        /* Enable the channel and its interrupt */
        AT91C_BASE_ADC12B->ADC12B_CHER = (1 << eAdcChannel_);
        AT91C_BASE_ADC12B->ADC12B_IER = (1 << eAdcChannel_);

        /* Start the conversion and exit */
        AT91C_BASE_ADC12B->ADC12B_CR |= AT91C_ADC12B_CR_START;
        return TRUE;
    }

    /* The ADC is not available */
    return FALSE;
}

/* end Adc12StartConversion() */

```

No other functions are required for this basic driver. It could be improved by queuing conversion requests in a FIFO, but there are lots of implications there and given the intended application this is unnecessary complexity in the driver.

At the time of writing, the first result returned by the ADC task always is high by up to 30 counts. This is consistent across all the dev boards tested. If subsequent conversions are done quickly, they are accurate. The longer the delay between conversions, the higher the difference becomes (until it reaches some peak value). This implies that there is charge accumulating on the sample and hold capacitor when the channel is not in use. This is obviously something that would need to be addressed on a production device, and averaging is not the solution. The error is not a function of noise, so it would be irresponsible to Band-Aid the problem like that. Throwing out the first reading is a poor choice, also. When problems are identified during development, they must be solved before release. If the problem was discovered later and was hardware-related, then a solution that reliably mitigates the issue would make sense (but it really should be fixed in hardware!).

### 13.7 • Chapter Exercise

For this exercise, we will create a very basic voltage meter. For simplicity, it will use the potentiometer on the EIE ASCII development board. However, you could wire up a circuit into the daughter board connection and go as far as using real probes. Make sure you scale the input voltage to a safe level into the board (less than 3.3V). Any resistor divider should be as high-impedance as possible but remember you must meet the input impedance requirements of the ADC. The scaling factor must also be included in the calculation.

It is so important to take a moment to consider the use case for the outcome of a design. A voltmeter measures an analog signal that isn't changing too quickly. A commercial multimeter typically updates its display every 500ms or so. Faster update rates are not necessary if a human is looking at it. Use three decimal points of precision on the LCD. The LCD should also say "Voltage" and include a 'V' unit. A good multimeter logs data, so this design should output the readings to the terminal in formatted ASCII. Even if you were logging the data and sampling 10 or 100 times per second that's still slow relative to what the ADC can do. So, speed is not an issue, but stability is a priority and filtering out noise is useful.

Design and document the application that you will write. At the very least you need to run the ADC, LCD, and debug functions. Will you use a single task or several tasks? If you use multiple tasks, make sure that any data exchange is done in an appropriate way for object-oriented design. Do not just use a global variable. If you want to get really fancy, add a calibration routine in and provide a nice interface for the user.

What is likely the most difficult part for those new to ADC is correctly calculating the voltage being read since it is reported in counts. If you use an external circuit with an input resistor divider, this factor will have to be included in the calculation. Be careful with any math to ensure that the integer division of the processor does not drop too much resolution as the result of divisions. Very often we must scale up values during operations. In that case, be careful to not overflow the numbers. With a 12-bit ADC, overflow is unlikely unless you have decided to use a lot of averaging.



## Chapter 14 • ANT Radio System

No matter how many times you listen to a song on a radio, talk on your cell phone, or check your location on your GPS, you must appreciate the fact that massive amounts of information are whizzing around you, and somehow, somehow, the device in your hand can pick out the signal you want from everything else and show you meaningful data. The radio frequency (RF) world of wireless data transfer is truly amazing and is becoming more and more accessible to incorporate into embedded systems. Though there will always be an element of “black magic” in any RF design, the global knowledge and experience base and sheer number of companies and individuals who can wield that magic is growing every day.

To penetrate the market with a radio system that will be widely adopted is a sizeable task. The system must be viable in every way: usability, function, performance, cost, accessibility, and popularity. These characteristics are almost exactly orthogonal to each other at the beginning – essentially a Catch-22-22-22! How many wireless technologies do you know? AM/FM radio, Wi-Fi, Bluetooth, CDMA, TDMA, GSM, UMTS, HSPA, LTE may all be in your repertoire. Of these, only Bluetooth is suitable for smaller, personal networks, but the protocol is complicated, the overhead is substantial, and the actual number of real applications – though theoretically quite sizeable – has been essentially limited to data transfer, cell phone headsets, and wireless music players. Many proprietary technologies also exist that are not typically available to, or practical for, designers to implement like those used in garage door openers, car remotes and other wireless gadgets (take a moment to think about how many wireless products you depend on in life!).

Consumer and commercial markets have started to see growing demand for personal area networks (PANs) that are ultra-low power, easily deployable, highly interactive and low in cost. A PAN is intended to serve individuals within a limited sphere of coverage – perhaps 30 feet or so – with the intent of seamlessly interacting with the user’s immediate environment so their surroundings can be customized when they are present. In many applications, data rates have become less of a concern as devices are looking for more passive interaction and can make things happen with sometimes as little as only a few bytes of data exchanged. For example, your front door can unlock as you approach it if you carry a key fob in your pocket. The same key fob can unlock your car doors and disarm your security system, sign you into work, and identify you at the local coffee shop all by transmitting just a few bytes of data. Many of these devices are the essence of the “Internet of Things” (IoT). Smartphones can act as one side of these connections, but there is typically an embedded device on the other side.

The PAN concept is gaining momentum as more and more applications for wireless technology emerge and the capability to integrate that technology grows. A leading example is in the fitness world, where users are demanding smarter interfaces to exercise equipment as well as feeding a growing obsession with data monitoring. There is also a growing interest from a social networking perspective that is already serviced over long distances through cellular technology, mid-range distances with Wi-Fi but is still looking for a standard solution for the short-range leg. The race is on to offer a solution that will be adopted by the world.

The Zigbee consortium has been running since the early 2000s and has achieved reasonable market adoption and penetration amongst engineers but not so much in the consumer space. Zigbee radios sit on the fringe of the PAN and the wider-area network, offering a solution that works but is not ideal.

Bluetooth is trying to adapt to meet the growing needs of the PAN and developed a standard called Bluetooth Low Energy (BLE). It took years to formalize the standard but was finally released as part of the Bluetooth 4.0 spec in 2010. BLE is marketed as



Bluetooth SMART, though it has no compatibility with Bluetooth Classic despite being part of the same standard spec. A company in Cochrane, Alberta, Canada called Dynastream developed an ULP radio protocol called ANT+ in the mid-2000s and it became hugely popular in fitness applications particularly cycling long before the BLE spec was ratified. Bluetooth SMART devices have been rapidly catching up to ANT+ devices and the protocols compete head-to-head in many markets. Though Bluetooth had the initial advantage of being supported natively in all smartphone chipsets and operating systems that support Bluetooth 4.0, ANT+ has worked tirelessly to be picked up by the major vendors and incorporated natively as well. Dynastream has since been acquired by Garmin.

Dynastream's ANT protocol coupled with Nordic Semiconductor's low power 2.4GHz radios have emerged in the last few years and offer an excellent solution to meet PAN and IoT needs. Nordic Semiconductor is an industry leader in 2.4GHz radio technology and has come up with very innovative solutions to support these devices. Their nRF51 and nRF52 series of chips are tremendous system-on-chip devices that not only support dual-mode ANT and BLE but also offer Cortex microcontrollers with a good chunk of system resources available for a user application. The current EiE development boards feature the nRF51422 the supports both protocols with a Cortex-M0 core.


The ANT Alliance has grown at a remarkable pace, and companies are finding ways to use the technology in an increasing number of applications and devices that further spurs demand. ANT meets all the needs of the PAN and continues to develop in popularity and application. We choose to focus on ANT as it is capable of better power performance vs. BLE. It can easily exist as both a Master and Slave / Transmitter / Receiver configuration and is significantly easier to understand than BLE.

This chapter takes an in-depth look at the ANT protocol as it is presented from Dynastream, then examines the drivers and application code that will be used to establish and maintain a radio link for an application using ANT. The main purpose is to guide you through the driver code that has been written already to help you become more fluent with ANT. This is a critical step before moving on to building an application as it will be imperative that you know how to debug your system as you write code.

#### 14.1 • ANT Wireless Radio

Communicating with the ANT protocol is quite simple. On the hardware side, the development board contains everything necessary to implement the transceiver including the Nordic nRF51422 chip and a PCB antenna. This hardware can achieve line-of-site range up to 100 feet, though for very reliable connection a range of up to 30 feet is more appropriately specified. The work we will do to implement ANT is the development of the Host system that will interact with the ANT protocol stored on the nRF51422. The hardware takes care of a large portion of the communication, but we still need to understand how to write firmware to interface to this system.

When starting to use any new piece of hardware, the first thing you should do is a lot of reading to try to understand the new technology from the manufacturer's perspective since they are the experts at it. Datasheets and application notes should be gathered and organized, making sure you have the latest revisions and all the information. This material should be kept in an easily accessible location as you will no doubt be referencing it frequently as you are first integrating the device. This chapter will not repeat all the information that is already published on ANT, but it will reference a few of the most important pages and reiterate or expand on key concepts.

 Take some time to download and review two important documents from the ANT website, [www.thisisant.com](http://www.thisisant.com). Navigate to the Developer > Resources > Downloads page and you should see the documents near the top of the Documents tab:

1. ANT Message Protocol and Usage
2. Interfacing with ANT General Purpose Chipsets and Modules

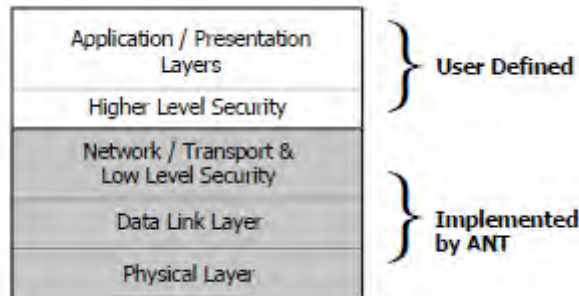
The ANT Message Protocol and Usage guide and the Interface document make up the essential documentation for ANT and should be read. It is recommended that you read through the entire document before continuing so that you get into the context of the rest of the discussion. We have already referenced the Interface doc to set up the hardware, but it contains additional information about the signaling required to send and receive ANT messages and is thus essential to reference while writing the driver code in this module.

It is also handy to have a hard copy of the ANT documentation you will use the most. You do not need to print all of it, but it is highly recommended to print the ANT Message Summary table and the Channel Response Message Codes (pages 51-56, 115-117 in rev 5.1 of the protocol document). You will reference these pages a countless number of times as you develop with ANT.

## 14.2 • Building the ANT Stack

If you were learning ANT on your own, you would have to spend more time with the various documents from Dynastream to understand what applies to what you want to do. Since we already know what we need, a great way to get you into ANT quickly is with a guided tour on setting up and testing the ANT driver for the EiE system. A completed driver is called a “stack” and the only thing you really need to know about the stack to use it is the API (application programming interface). However, knowing the underlying code gives you the depth of knowledge to support using the technology. You can also make improvements or write your own version.

The ANT reference material includes documentation and example source code for getting an ANT system up and running. From the diagram below, you can see how ANT fits into the OSI model and what parts the ANT protocol takes care of and what is up to the Host application.



**Figure 14-1** ANT from the perspective of the OSI model

Your application is entirely up to you including microcontroller peripheral drivers that will properly interface with the ANT device you are connecting to. The nRF51422 was the first ANT system on chip where a lot of the ANT protocol processing was offloaded to the proprietary ANT “soft device” which shares the Cortex-M0 processor. The whole Host can be implemented directly on the nRF51422 and run without any additional hardware – a nice solution for compact, low cost, low power systems. In our case, we wanted to interface using the classic Host-ANT relationship. Dynastream provides a starting firmware stack for the nRF51422 which facilitates this. We modified it slightly, but as of



“Slave” in the context of the SPI communication between the local ANT hardware and the Host MCU with “Master” and “Slave” in the context of the ANT radio system between two ANT devices that this chapter will describe.

Additional handshaking lines are implemented called “SEN,” “SRDY” and “MRDY.” “SEN” is essentially the SPI chip select line from ANT (SPI Master) to Host (SPI Slave). SRDY is “system ready” which the Host uses to tell the ANT device that it is ready for data to be clocked in (since the Host is SPI Slave). MRDY is “message ready” which the Host uses to tell ANT that it has a message to send to ANT and thus needs to be clocked to transmit. The handshaking lines behave similarly to RTS and CTS in typical asynchronous communications.

### 14.3 • ANT Message Protocol and Usage

The ANT Message Protocol and Usage document is the key specification for implementing ANT. It covers everything from how different networks can be formed, to the exact bytes that will be exchanged as messages between your Host controller and an ANT device. Since it covers the full ANT protocol, some of the information will not apply directly to how ANT will be used in our application, but it is a good idea to review the material so you know the full capabilities of ANT.

#### 14.3.1 • ANT Protocol: Sections 1 thru 4

Sections 1 and 2 are a brief introduction to the ANT system and available hardware used to realize an ANT-based PAN. Section 3 gives you a full overview of the network topologies that can be supported by ANT – start dreaming of your applications! Section 4 clarifies some terminology and emphasizes that any implementation of ANT requires a Host controller along with the ANT engine for each “Node” in an ANT network. The nRF51422 processor can function on its own as both the Host and ANT hardware, though we will use the SAM3U2 as the Host. We will denote the nRF51422 simply as “ANT” and the SAM3U2 as the “Host.”

So how does data flow in the system? At the very least, two nodes are required to communicate. In most discussions of an ANT system, “Master” and “Slave” refer to ANT nodes where communication takes place over the air. Imagine we have two EiE boards that we want to send data between using a single ANT channel. One of the boards is set up as the Master, and the other is set up as a Slave. The Host on the Master side wishes to send some data, so it gives that data to ANT which in turn broadcasts it into the air. If there is a paired Slave device, the data is received and passed to the Host on the Slave node.

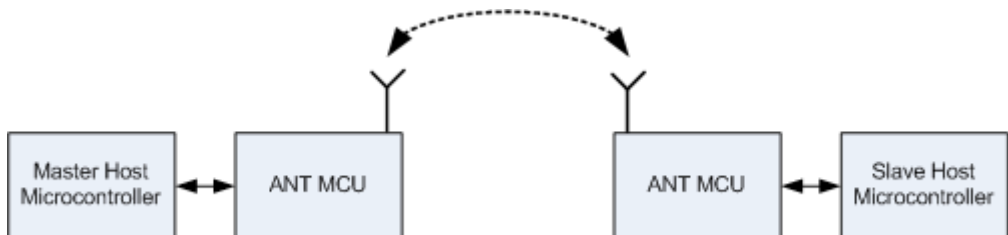


Figure 14-3 ANT system

### 14.3.2 • ANT Protocol Section 5: Channel Parameters

Section 5 in the ANT Protocol document is very important as the core concepts of the ANT Master and Slave relationship are described, along with the key channel parameters that come together to allow two devices to communicate.

One of the important take-aways here is the list of configurable parameters available to define an ANT node:

1. Channel Type
2. RF Frequency
3. Channel ID
4. Channel Period
5. Network Number

To get two ANT nodes to talk to each other over the air, they must be configured on the same radio Frequency and with Channel Types set such that there is at least one Master and at least one Slave in the network. Determining which should be Master and which should be Slave is influenced by data and power requirements. The device you assign as a Master might not be what you would initially expect since “Master” tends to imply the more powerful device. From other protocols like SPI, we are used to single Masters and multiple Slaves. With ANT, the Master tends to be the lower power and simpler device. In a multi-node ANT system, there are often multiple Masters with a single Slave.

For example, if you had an array of remote sensors that were communicating temperature readings every minute back to a base station, which device would be the Master? The sensors are the Masters in this scenario since they initiate communication. The Master always initiates communication in the ANT protocol. In most ANT deployments, the ANT Master is the low power device since it can start broadcasting when it needs to and shut down and sleep when it is done. The receiver, on the other hand, must spend more time actively listening for traffic so it will use more power to keep its receiver on so it does not miss any messages. The whole Master/Slave relationship in the ANT world may seem counter-intuitive at first, but as you start to explore different applications you will see that it makes a lot of sense.

If you have the radio settings correct and a Master/Slave device pair is in close enough proximity to each other, then messages will be exchanged between the two devices when they have open channels. At this point, the proprietary operation of the ANT protocol kicks in. This is also where the other channel configuration parameters come in. The radio side of a receiving ANT device will hear essentially every message in range if it is at the same frequency. It will parse all the data it sees and picks out messages that match the channel parameters that have been configured.

There is a level of security that has something to do with the Network key set in the device – probably some sort of encryption based on the Network key that will scramble the message data. For development purposes, the Network key is always 0, referred to as the “Public network” that is not protected. For specific applications, network keys are assigned and certain networks have an associated set of rules that define how devices in that network should behave. This is the basis of the ANT+ ecosystem which allows interoperability between ANT devices from multiple vendors with multiple products. If you do not want to play with others, you can purchase your own Private Network key to operate your devices privately from any other ANT devices and therefore set your own rules.

The message period also comes in to play, which is the rate you define for messaging to occur. The standard / recommended rate is 4 messages per second (4 Hz). It takes two

devices one to three seconds to “pair” when a 4 Hz message period is in use. If you have access to an ANT development kit, you can observe the pairing process as you turn on a device, then in a few seconds see data messages. A Slave device with 4 Hz messaging period can pair with a Master with 2 Hz messaging, though the Slave will report missed messages every second period. If you want two devices to talk, the best way is to make sure all their channel parameters match exactly.

### 14.3.3 • ANT Channel ID

The other very important element described in Section 5 of ANT Message Protocol and Usage is that of the Channel ID. Try not to get mixed up when talking about channel setup and channel parameters and channel type and Channel ID. Channel ID is a specific group of parameters that ultimately determine whether two ANT systems will talk. The Channel ID is made up of three fields:

1. Device Number (16 bits).
2. Transmission Type (8-bits)
3. Device Type (8 bits: 7 bits for a device type, 1 bit for a “pairing bit” that will be discussed later)

Once two devices are synchronized and messages are passing the security layer, ANT will check the Channel ID and decide to pass on each message to the Host Controller. If the Channel ID does not match the device settings, the message is simply discarded. As far as your system is concerned, it was never received. For a given application, it is likely that all devices will have matching Transmission Type and Device Type. Therefore, the 16-bit Device Number allows 65,535 unique devices.

For a Master device, all three fields must be set explicitly before the channel is opened. On the Slave side, values that match the intended Master can be set, or any field can be set to “0” for a wildcard. This allows you to control how your devices behave and pair up to communicate. If you know the exact Channel ID of the nodes that need to talk, then you can set all three fields and be assured that the ANT protocol will exchange messages only between the devices with matching Channel IDs. If you are not sure about whom you want to communicate with, or if you purposely want to communicate with multiple devices, then you can make use of wildcards.

If you set wildcards on a Slave device and the Slave then pairs with a Master, the wildcard fields in the Slave will be changed automatically to the Channel ID of the first Master device that communicates with the Slave. The Channel ID will remain that way until the Master stops broadcasting and the Slave search times out. If the Slave has not yet timed out and a new Master with a different Channel ID tries to communicate, it will be ignored because the Slave will still be trying to communicate using the previous Master’s Channel ID. The timeout period is a configurable parameter that can be set from a few seconds to infinite. That setting needs to be considered for the system you deploy if you rely on the timeout to occur before communicating to another device. If the Slave finds and pairs with a Master, exchanges data and knows that it does not need any more data, then the Slave can close the channel and reopen it with a new wildcard search.

### 14.3.4 • Transmit Data Types

The type of Data that is exchanged between two ANT nodes falls into one of three categories:

1. Broadcast Data: one-way fire and forget messages.
2. Acknowledged Data: receiver responds to confirm that the message was



received. It is up to the Host software to resend the message if an ACK is not received.

3. **Burst Data:** large amounts of data sent as acknowledged data messages. ANT automatically retries any messages within the packet that are not ACKed. The Host receives confirmation of success or failure for the complete packet of data. This is designed for high data rates but is still limited to 8-byte payloads. However, there is very little delay between each message.

Both Master and Slave can send any of the data types essentially full duplex (it's not truly full duplex, but it looks like it). Broadcast and Acknowledged data messages always have 8 bytes of payload data. The Master sends data first at the set message rate, and the Slave will respond with data immediately following the end of the Master's data if response data has been queued up. In other words, both the Master and Slave can exchange data each message period. However, it is not possible for the Slave to parse the Master's message and queue a response that would be sent on the same period. The Slave's message must be queued prior to the next message period, and it is this message that will be sent by ANT immediately after the Master's message is received before the Master's message is presented to the Host.

Probably the most difficult part of ANT to get used to is that when a channel is open on the Master, data is always sent from the Master at the defined message period to maintain the channel synchronization. That means that regardless of if you have new or meaningful data to transmit, the ANT protocol demands that data is sent. If you do not give the Master new data to send, it will simply repeat sending whatever the last data packet was. Given this information, you must ensure that your Slave device will properly handle the continuous message stream even if the message data is stale.

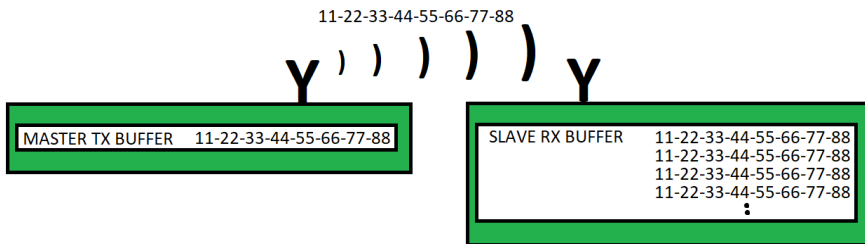


Figure 14-4 Continuous message stream

There are a lot of ways to manage the required messaging, and the rules that you establish for your system will likely differ from others. Your system may be able to handle the repeated data without any problems. For example, if you had a sensor network where a Master was sending the current temperature as data, then as long as the continuous messages had the latest temperature reading the system would function well. If the system could not handle repeated data, your application could define its own application protocol such that the data sent contained some sort of flag byte, message number, or enumeration byte that could be compared by the Slave to the last message received to determine if the message is new. All that would happen in your application because ANT relays anything it receives to the Host. If you are coding directly on the nRF51422, it can be configured to repress repeated messages. In the EiE system, the nRF51422 simply forwards every message to the Host.

Broadcast messages are the default type and are what the system will send at the message period if you do not tell ANT to send Acknowledged or Burst messages. Since Broadcast messages do not get acknowledged by the Slave, the Master has no way of knowing if any Slave received the message. In a lot of applications that does not matter.

Broadcast messages are good because the Slave does not have to use power to send an acknowledgment back, and the Master does not have to have firmware to deal with it. Many ANT systems are unidirectional.

An Acknowledged data message is structurally the same as a Broadcast message, but it triggers a response back from ANT. The Master will receive an event message that indicates either the message was sent and acknowledged, or the message was sent but not acknowledged. This occurs on the same message period that the message was sent. If the message fails, it is NOT automatically retried. The Host can decide to resend the message or send a new message. If the Host does not queue a new message, the next message the ANT Master sends will be a Broadcast data message with whatever the last Broadcast data message content was to maintain the channel.

Using an Acknowledged data message is useful in a system where data reception is critical, but it doesn't totally solve the problem of guaranteed message transfer. Why? The Slave that sends the Ack back to the Master does not know that the Ack was received by the Master, and there is no guarantee that the Slave's response is received by the Master. Better schemes are required to have a truly robust data transfer system. This is a fundamental problem to solve in RF data communications.

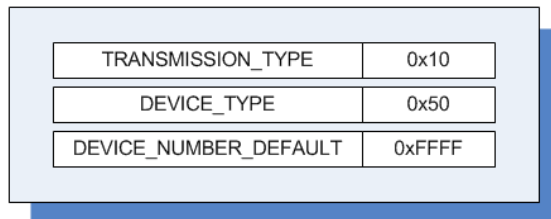
A more common use for Acknowledged messages is for a Master to decide if it can stop transmitting because the data that it is trying to pass has been received successfully. In this scenario, it does not matter if the Master misses some Acknowledgements. The Slave knows that the ACK has been received simply because the Master changes its behavior. For example, it stops sending data.

Burst data messages allow very fast data transfer to occur over the ANT channel. Data rates up to 60kbts/second can be achieved. ANT message and protocol usage describes the details of Burst transfers, and there is also an application note that further explains it. Since this course will not use Burst transfers, no further discussion will occur here.

The remainder of section 5 in the ANT protocol document talks about some other interesting features of ANT. Though we will not get into those, the reader is highly encouraged to review and understand these capabilities as their applicability is very high for other ANT usage scenarios.

#### 14.3.5 • ANT Protocol Section 6: Pairing

Section 6 talks about the concept of pairing a Master and Slave ANT node and the different pairing relationships that can exist. Again, the important detail to remember is that for two ANT nodes to communicate, their Channel IDs must match. A Master's Channel ID must be completely set and, in some way, unique to any other Master within range. A Slave Channel ID can have "wildcard" parameters that will allow it to search for Masters within a group based on the different parameters of the Channel ID. Once the Slave is communicating, its own Channel ID needs to be unique or the information it sends must identify the Slave in some way so the Master knows what it is talking to.



The diagram shows a light blue rectangular box with a blue shadow to its right. Inside the box is a table with three rows and two columns. The first row has 'TRANSMISSION\_TYPE' and '0x10'. The second row has 'DEVICE\_TYPE' and '0x50'. The third row has 'DEVICE\_NUMBER\_DEFAULT' and '0xFFFF'.

TRANSMISSION_TYPE	0x10
DEVICE_TYPE	0x50
DEVICE_NUMBER_DEFAULT	0xFFFF

Figure 14-5 Channel ID unique identifiers



If the pairing bit is used, then many pairing conflicts can be avoided since the pairing bit is another unique part of the Channel ID though only used during the initial pairing operation. Pairing errors could still occur if two sets of devices were trying to pair at exactly the same time with matching channel parameters.

There are some further notes on pairing in Section 6 of the protocol document that is worth reading, especially the information on Proximity Pairing that helps to further mitigate the problem of incorrect pairing relationships forming.

#### 14.3.6 • ANT Protocol Section 7: ANT Interface

We have talked a lot about the messages that are exchanged in the ANT protocol but have not yet looked at the actual message frame used by ANT. Section 7 in the Protocol document fully explains how messages are put together, but we will emphasize the most important parts here.



Find the ANT Message Summary table in the ANT protocol document and print the seven pages. Also, find the Message Codes table in the Channel Response / Event Messages section and print these three pages. Keep this very helpful information close by when working with ANT.

The Message Summary table breaks down the message types into various classes that are self-describing enough:

- Config Messages
- Notifications
- Control Messages
- Data Messages
- Channel Messages
- Requested Response Messages
- Test Mode
- Extended Data Messages (Legacy)

Regardless of the message class, all ANT messages follow the same format shown in the figure taken directly from the protocol document.

Sync	Msg Length	Msg ID	Message Content (Bytes 0 – (N-1))	Check sum
------	------------	--------	--------------------------------------	-----------

Figure 14-6 The ANT message Frame

**Sync:** A known first character to indicate the start of a message. For synchronous transmission like in the course development board, the sync byte is 0xA4 for a write or 0xA5 for a read, with respect to the ANT processor. A message parser should throw away any frame that does not start with the correct sync byte and keep looking through a data buffer until a Sync byte is found. The ANT device should never pass an improperly formatted message to the Host, so an incorrect message is an error in the SPI data transfer and/or associated driver functions that are responsible for receiving messages from the ANT device.



Write out 0xA5 in binary and ask yourself why this number (or 0xAA, 0x55) is often used for synchronization, test, or sentinel data.

**Msg Length** (“Length” in the protocol spec): the number of payload bytes in the message. Since this byte appears immediately after the Sync byte, it can be used to condition a message parser to read the remaining bytes in the message since messages are variable length. It can be checked for an absolute maximum but otherwise, it cannot be determined if it is correct until the checksum is calculated.

**Msg ID:** The message ID corresponds to the message number in the ANT Message Summary table. Probably every ANT firmware project will have a header file with mnemonics for the IDs along with their associated message lengths. This field is the key byte of information used to process the messages received and will generally be the parameter in a big switch statement that handles all the different messages.

**Message Content:** the payload or content bytes of the message. The number of bytes varies with each message. Don’t make any assumptions about the values. In most cases of ANT status messages, the first content byte is the channel number to which the message applies, but not always.

Data Messages	Broadcast Data 9.5.5.1 page 97	No	Host/ ANT	Varies	0x4E	Channel Number	Payload (8 bytes)	Extended Data (optional) Refer to section 7.1.1
---------------	-----------------------------------	----	--------------	--------	------	-------------------	----------------------	---

**Figure 14-7 Broadcast ANT Message Summary table showing the message bytes**

**Checksum:** The last byte in the frame is the verification byte. The checksum is calculated as the XOR of all the previous bytes in the message starting with the Sync byte. This is the only data checking mechanism that the ANT protocol provides but is sufficient for most purposes. When parsing or building messages, the checksum can be built up with each byte. An XORed checksum like this is better than a single parity bit, but it still could be subject to false positives if bit errors within the message occur in pairs and at the same bit locations. If ever a system required more comprehensive error detection it could be added at the application level at the cost of some overhead bytes used in the data messages sent. In over a decade using ANT, we have yet to observe an error.

#### 14.3.7 • ANT Protocol Sections 8 and 9: Examples and Appendix

The last full section of the ANT Protocol document provides numerous examples of different network configurations and how they would be set up to operate. If you are still unclear on any of the concepts of how an ANT-based system works, then reading through the different examples may help you to better understand the requirements for setting up various Master and Slave nodes.

Beyond Section 8 is the Appendix that provides detailed information about every command in the protocol. The most important parts of this section have already been mentioned, which are the ANT message summary table and the table of Channel Response/Event codes. Reading the whole Appendix in order is not going to help you much – it is much better to reference command information as you need it either the first time you use a new command or if you experience any problems using a message.

#### 14.4 • Message Handling

Though it took a while to articulate all the details of the ANT protocol, you are now ready to start thinking about implementing ANT. Don’t worry if you feel a little overwhelmed. As engineers, our job is to take information and break it down into manageable pieces.

When designing a Host state machine to interface with ANT, you must handle the different scenarios of messaging that will occur. The ANT-Host relationship has two sorts of “modes” of operation depending on whether a channel is open or not. Though the ANT documentation does not make this distinction, when you use ANT you will likely notice it and may alter your application’s message handling approach depending on what mode the system is in.

#### 14.4.1 • Messaging with Channel Closed

When there is no active channel in the ANT system (Master or Slave), the Host will never receive a message from ANT unless it is a response to a message sent by the Host. The only exception is if the power is cycled on the ANT device that will cause it to send a Startup message (0x6f).

During this closed-channel mode, only Configuration and Control messages from the Host should be sent to ANT. Trying to send a data message does not make sense as the data has nowhere to go and ANT will respond with an error message that indicates it is not able to send data messages. The only exception is that the Broadcast data buffer can be preset with a message to send when the channel is opened so the default 0-0-0-0-0-0-0-0 is not sent.

When a configuration message is sent, a single response will be received from ANT in reply to that message to indicate whether or not the configuration message is accepted. This reply comes in the form of a Channel Response message (0x40) as shown here.

SYNC	LENGTH 3	MSG ID 0x40	CHANNEL	MSG ID THAT THIS RESPONSE IS TO	RESPONSE CODE	CHECKSUM
------	-------------	----------------	---------	---------------------------------------	------------------	----------

Figure 14-8 Format of the Channel Response message

Message 0x40 is a bit confusing because it doubles as a “Channel Response” but is also a “Channel Event” message. The Channel Response is typically a reply to a message sent by the Host. A Channel Event is a status or event message from ANT to the Host when there is a channel active - more discussion in the next section. The key is the “MSG ID THAT THIS RESPONSE IS TO” which will be 1 for a Channel Event or the ID of the message to which the response applies for the Channel Response.

In either case, message 0x40 contains a Response Code that must be referenced against the Message Codes of the ANT protocol document. You should have these 3 pages printed already. The table combines response codes that can come when the system is idle and thus when Channel Response is a reply to a command you are issuing, but also contains the codes for when the message is a Channel Event.

As a Channel Response message, the response code will either be 0 meaning the message was successfully sent or will be an error code that provides a reason why the command was not acceptable. Since not every possible reason for a failed message can have a distinct error code, you may have to do a bit of digging to understand the error source. In any case, your system must parse a few bytes of the message to handle it appropriately and decide how to react. For small systems, you can handle common responses and leave everything else to a generic handler.

A Channel Response message should be used to ensure that the channel setup you want based on the configuration messages you send is successfully set up. If any Channel Response indicates that your command is not accepted, your system must react and

either retry, change parameters, or reset the ANT system. Communicating to ANT when the channel is not open will involve messages that are critical to the setup and your system should not proceed if any of the messages fail.

If a message does fail for some reason, it is unlikely that retrying the same message is going to work. If the message was formatted correctly enough to get a response from ANT, there is likely something incorrect about the message parameters or the state of the system.. The brute-force approach of resetting the ANT device after a message is denied works when the message you are trying to send normally works but for whatever reason is not currently working. You could try to send different messages to ANT to see if you can correct its state. For example, try to reconfigure other channel parameters that may be the cause of the conflict of what you are currently trying to configure. Keeping track of the current state or coding a system that is able to try to recover can lead to some complex code to try and implement.

Most likely you will find that bad responses to configuration messages result from programming errors and will be up to you to fix long before the device ever leaves your workbench. If you cannot figure out why a message fails just by reading the ANT documentation, a good way to solve the problem is with a breakpoint, debugger, and watch window showing the received message from ANT so you can manually figure out what is wrong and make the correction. Troubleshooting further is critically dependent on whether the response message has made it into your buffer. If not, that's a different problem to solve. Assuming it has, the key is the Response Code that is returned in the Channel Response message. If the Response Code is anything but 0 (no error), then take note and figure out how to make the correction.

Beyond simply getting the parameters of a message wrong, probably the most common error comes from trying to do something on a device that is not supported by the ANT part you have in your system. For example, the course development board uses an 8-channel nRF51422 IC. Fortunately, this part supports almost all the available ANT features, though as ANT continues to evolve this will change. To save cost, some ANT parts are only single channel, so if you tried to open anything but Channel 0 then it would fail.

All Configuration messages sent by the Host will receive a reply from ANT, so a system is usually set up as a state machine to send a message and wait for a good response before proceeding. Only one message is sent at a time. If the configuration stream you send has been verified to work properly, then you can be reasonably assured that it will always work when it runs. If the system gets into a state where something that normally works no longer works, a system reset is often the best approach to getting around the problem. If the error happens consistently, then the application should be examined to find the root cause of the failure.

If you try to send Configuration Messages for a channel when the channel is open, they will fail because channel parameters cannot be updated on open channels. If you try to send data messages when the channel is closed, you will, in fact, get a response from ANT even though the ANT protocol document indicates there is no reply to these messages. The response is a Channel Response message and the Response Code will tell you that the channel is not in the correct state to receive data. This is not technically a reply to the message.

#### 14.4.2 • Messages When Channel is Open

The messaging that takes place when a channel is open is quite different on a Master compared to a Slave. If the system is left idle with the channel open, the Master ANT device will continually send the Host Channel Event messages every message period indicating that Broadcast messages have been sent. Whenever one of these messages is

received by the Host, it should be used as the indicator to queue new data for the next Broadcast or Acknowledged data message. If the Host can prepare the data and send the data message to ANT in less time than a message period, new data can be sent with each message. If the Host is too slow or does not have new data to send, ANT will send the same Broadcast Data as it did on the last transmission over the air and continue to send the Host Channel Event messages to indicate the Broadcast message event.

When the channel is open on the Master, most of the messages received will be message 0x40 in the form of a Channel Event message that gives you status about what is happening. The Event Code is "EVENT\_TX" (0x03) for every successful message.

SYNC	LENGTH 3	MSG ID 0x40	CHANNEL	1 (indicates this is a Channel Event message)	EVENT CODE	CHECKSUM
------	-------------	----------------	---------	---	---------------	----------

Figure 14-9 Message 0x40 as a Channel Event

On the Slave side of the system, the receiver is active and waiting to get messages from a Master when the channel is first opened. However, other than a "Channel Open" confirmation message, an ANT Slave will not report any messages to the Host as long as no Master is present. If no messages are received within the timeout period set up for the system, then ANT will automatically close the channel and indicate the timeout with a message to the Host. However, if a Master is present and the devices have paired, then the Slave should be getting data messages at the system message rate. The Slave will pass this data with messages to the Host. The message is either 0x4e (Broadcast Data) or 0x4f (Acknowledged Data). Note that a Slave that has its channel open will be consuming power, so very low power devices must intelligently manage their receivers.

When the Slave has been receiving data from a Master and a message is missed, the Slave will still communicate to the Host on the message period but will tell the Host that an expected message was missed. A missed message that was expected usually occurs because of the RF environment where the message does not reach the receiver or is corrupted in some way. This is a regular occurrence with wireless communications.

The Slave will report missed messages for at least 8 consecutive message periods. If a message from the Master is received in the interim, then the data is passed to the Host and the missed message count is reset. If too many messages are missed, the Slave will automatically drop back to search mode and send the Host an EVENT\_RX\_FAIL\_GO\_TO\_SEARCH (Event Code 0x08) to indicate this. Depending on the configuration, the Slave can close the channel if no Master is found for a certain time. The default search time is 30 seconds, but it can be changed. An infinite time can be specified so the Slave never automatically closes. If the Slave times out and closes the channel, it notifies the Host with an EVENT\_RX\_SEARCH\_TIMEOUT (Event Code 0x01) message.



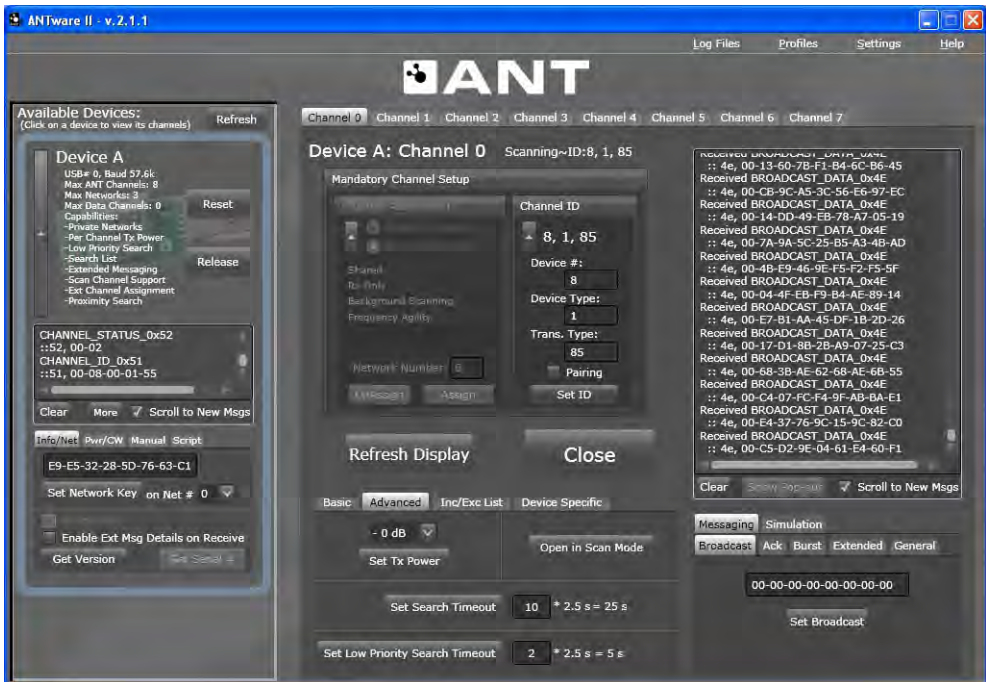
An extremely beneficial exercise to do now would be to draw an event diagram for the Master and Slave in operation. If you can understand how the messaging works, reading and writing the code will be substantially easier.

### 14.5 • Debugging an ANT System

Though the ANT protocol is straightforward if you have not used it before you will be surfing the learning curve just like with everything else new you learn. Seeing how a system operates is by far the best way to start fully understanding it. A key factor in working with a whole new protocol is having a known working device that you can use to get going. Trying to troubleshoot brand new hardware with brand new firmware is very

difficult as you never know what variable might be causing trouble. Adding in wireless communication to the mix makes it even harder.

Development kits are available for ANT products and are great if you have no other resources. However, the course ANT hardware implementation is also a proven, working design and the steps used to program and test the device will guide you. The ANT USB-M module for the PC is a low cost, important piece of the puzzle. The PC software for the USB-M is called ANTware II and is provided free to ANT developers. ANTware provides a GUI to quickly setup channels, open and close, and view messages that are present.



**Figure 14-10** Screen-shot of ANTware II from Dynastream Innovations

A great way to start to see if anything is happening is to use a Slave Scanning channel in ANTware with a full wildcard Channel ID. Though this still requires the Network ID and radio frequency to match what you're sending, at least if you have something else wrong in the system you'll still see your messages.

Eventually, you will have to run just your system to make sure both of your ANT nodes properly pair and communicate. Debugging can be more challenging here because of the number of messages that will be coming in from ANT. Though 4 Hz may not seem very fast, if you are trying to read 4 messages per second it gets overwhelming very quickly. Even logging 25 seconds of communications will result in over a hundred messages.

If you are using a hardware debugger for your Host you can set breakpoints and halt the Host processor, but ANT will keep running and sending messages at the system message rate since it is not tied into your debugger. ANT will buffer a few messages and then start sending "lost data" messages which can sometimes upset debugging even further with another message type to deal with. A good method for debugging is to have a large receive buffer so that you can capture enough of a representation of the messages being received so you can find out where your problems are.



Another excellent tool is to have a serial output of all messages that ANT receives and communications between ANT the Host. As we develop the ANT API, we will develop debug functionality that will be used to further support learning and debugging efforts with ANT.

#### 14.6 • Programming the ANT Sub-System

There are many details of an ANT system that will come together to make it work. The core ANT driver needs to provide all the functionality and services required by an application and ensure they will run reliably. Since we already have the SPI and messaging drivers written, programming the ANT sub-system will focus on initialization of the ANT device and sending and receiving ANT messages to allow configuration and communication to take place. All the code will be kept in the source files `ant.c` and `ant.h`.

Dynastream provides code examples for a simple 8-bit MCU to implement the serial and interface layers on the Host. The functions we wrote used those examples as basic starting points but were significantly customized to work within our architecture. We use the ANT header files `antdefines.h` and `antmessage.h` for the predefined symbols for message numbers and channel parameters. These are important when implementing the protocol to make code readable. It's also a good idea to keep a vendor's naming conventions where possible in the event you end up troubleshooting with them. The less they must learn about your system, the better. Take a moment to open those two header files and look through them. Find the relationships between the symbols and the Message Summary and Event Codes in the tables from the ANT protocol document.



The `antdefines.h` file has values for all the channel configuration elements that must be specified in various messages. It also has the Response and Event codes that are used in status messages from ANT. These match up to the Message Codes table from the ANT Protocol document.

The `antmessage.h` file holds the symbol names for all the messages and their fields. You should try to get familiar with the ANT message structure and think in terms of offsets to locate the various bytes within the message frame as this will make coding easier. The definition in the file for the two parameters that each message type requires is shown here as an example. Note that we do not typically change other company's coding styles and conventions to match our own.

```
#define MSG_BROADCAST_DATA_ID      ((UCHAR)0x4E)
#define MSG_DATA_SIZE              ((UCHAR)9)
```

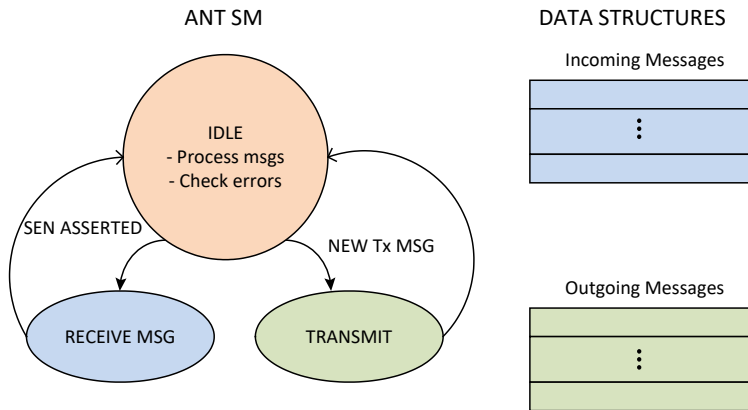
##### 14.6.1 • Firmware Design `ant.c` and `ant_api.c`

It is difficult to explain how a source file nearly 2000 lines long came to be, though in hindsight it becomes easier. Our driver encompasses the low-level hardware interface to the ANT device, the implementation of the ANT protocol, and all the setup and operation of an ANT system. We need to be able to receive messages from ANT that will arrive asynchronously as we have almost no control over when the ANT device will send the Host a message. All we know is that at any time, the SEN line is asserted which means ANT has something to tell the Host.

Various tasks in the system may want to use the ANT radio to transmit messages, though, in most applications of the development board or a product using ANT, only one task would use the radio. We do not expect tasks to have detailed knowledge of how ANT works, so our application will provide message buffers that can be written and read for both incoming and outgoing messages. We will make the decision that these buffers will use the standard ANT message protocol, and thus rely on the application to work with the

ANT standard 8-byte payloads. The application will also be responsible for handling data transmission errors or events that are provided by ANT, so this information needs to be relayed through the ANT driver to the application.

When the driver was first written, only `ant.c` was provided. It was quickly found to be too low-level to be usable to most people wanting to get going quickly with ANT in the EiE program. So, `ant_api.c` was added to further abstract the details of using ANT. We'll take the time to go through both sources and talk about the complete design, starting with `ant.c`. Capturing all the code and decisions made will hopefully provide good insight into the kind of considerations made in designing such a system.



**Figure 14-11** ANT task state machine and message buffers

The picture above is ultimately where we want to get to. We could continue to work top-down and start to build the interfaces to the incoming and outgoing message buffers, but it is a bit difficult to predict what we will end up with, especially if you have never worked with ANT before. So instead, we will tackle it this way:

1. Define data structures for all the required ANT configuration information, SPI parameters and incoming and outgoing messages.
2. Write the initialize functions and lowest level drivers to work with the SPI and messaging tasks to be able to exchange messages with the ANT device.
3. Write functions to read received messages and correctly set up and send messages to transmit per the ANT message protocol.
4. Send and receive messages to initialize the ANT device, set up each channel we want to communicate on, and process data when the ANT device is operating with the radio.

Each of the above will help to define the next step in the sequence. We have a general idea about how each layer will interact but trying to refine details at this point is less valuable than writing some code and testing to see how the system responds. We need to understand how the data will look when we get it, and then what makes sense for presenting it to the other tasks that need that data. We'll also discover what the appropriate level of abstraction is to make using the ANT subsystem as easy as possible but still provide the necessary access to ANT features.



### 14.6.2 • Data Structures

The easy and obvious place to start is with variables that we need. ANT communicates to the Host over SPI so we need an SSP peripheral pointer and configuration pointer. The variable of type `SspConfigurationType` will be initialized to all the ANT-specific requirements to interface properly to the SSP module.

```
static SspConfigurationType Ant_sSspConfig; /* Configuration for SSP peripheral */
static SspPeripheralType* Ant_Ssp;         /* Pointer to SSP peripheral object */
```

Next, we need a way for a task to define the ANT channel parameters it wants to use. We create a type `AntAssignChannelInfoType` for this purpose. All the members of this struct should now be familiar to you except for the `AntFlags` field which holds our ANT application's status information about the ANT channel. Those flag definitions are also shown, and they should make sense without further explanation.

```
typedef struct
{
    AntChannelNumberType AntChannel; /* The ANT channel number */
    u8 AntChannelType; /* ANT channel type */
    u8 AntNetwork; /* Network number */
    u8 AntNetworkKey[8]; /* Network key */
    u8 AntDeviceIdLo; /* Device ID low byte */
    u8 AntDeviceIdHi; /* Device ID high byte */
    u8 AntDeviceType; /* Device type byte */
    u8 AntTransmissionType; /* Transmission type byte */
    u8 AntChannelPeriodLo; /* Low byte of Channel Period */
    u8 AntChannelPeriodHi; /* High byte of Channel Period */
    u8 AntFrequency; /* RF frequency value */
    u8 AntTxPower; /* RF power level */
    u8 AntFlags; /* Flag byte for tracking status */
} AntAssignChannelInfoType;

/* Channel flags used for AntFlags in AntAssignChannelInfoType */
#define _ANT_FLAGS_CHANNEL_CONFIGURED (u8)0x01
/* Set when the ANT channel is configured and ready to be opened */
#define _ANT_FLAGS_CHANNEL_OPEN_PENDING (u8)0x02
/* Set when the ANT channel open request has been made */
#define _ANT_FLAGS_CHANNEL_OPEN (u8)0x04
/* Set when the ANT channel is open */
#define _ANT_FLAGS_CHANNEL_CLOSE_PENDING (u8)0x08
/* Set when a request to close the ANT channel has been sent */
#define _ANT_FLAGS_GOT_ACK (u8)0x10
/* Set when an Acked data message gets acked */
```

Two linked lists will be used for messages: one for incoming and one for outgoing. Outgoing messages are much simpler because the channel on which they are sent contains a lot of information already. So the struct is:

```
typedef struct
{
    u32 u32TimeStamp; /* Current G_u32SystemTimeIs */
    u8 au8MessageData[MESG_MAX_SIZE]; /* Array for message data */
    void *psNextMessage; /* Pointer to next node */
} AntOutgoingMessageListType;
```

Incoming messages (any message from ANT to the Host) must contain the data plus all of the channel information. We chose “incoming messages” to include both data messages

(that we generalize as ANT\_DATA) and status messages (that we generalize as ANT\_TICK). The enum AntApplicationMessageType is used for this and is the first parameter of a message that a task should look at.

```
typedef enum {ANT_EMPTY, ANT_DATA, ANT_TICK} AntApplicationMessageType;
```

An “ANT\_TICK” message is a status of some sort – usually an event code. This is a simplified indicator to the task to maintain the message period timing that is required for the task to understand if its channel has a good connection. For a Master channel, this is typically just a “message sent” indicator every time a message is broadcast. This is used to trigger the task to load new data in time for the next broadcast. Slaves need to watch this more closely, as it will contain event codes that indicate the status of the connection. For both Master and Slave, the regular, periodic occurrence of these messages is essentially the ANT heartbeat or system tick, thus the name ANT\_TICK. We will come back to this later on.

Having the driver pre-read the messages to determine whether they are data messages or tick messages offloads a lot of processing that higher-level tasks would need to do. However, those tasks still need a lot of information to understand what the message is and where it came from. This data is parsed out in the ant.c task and presented in the main message type called AntApplicationMsgListType. This, in turn, has an additional member struct for “extended data” that is available only on data type messages.

```
typedef struct
{
    u32 u32TimeStamp;                /* Current G_u32SystemTimeIs */
    AntApplicationMessageType eMessageType; /* Type of data */
    u8 u8Channel;                    /* Channel which data applies */
    u8 au8MessageData[ANT_APPLICATION_MESSAGE_BYTES]; /* Array for message data */
    AntExtendedDataType sExtendedData; /* Extended message data */
    void *psNextMessage;             /* Pointer to next list item */
} AntApplicationMsgListType;
```

These linked lists act as a queue and in most cases, will be FIFO. Using a linked list, though, allows us to pull out messages as required. This would support applications that may have multiple tasks using different ANT channels at different messaging speeds, or even a single task that may need to prioritize messages or be particularly busy at times.

### 14.6.3 • Serial Drivers and ANT hardware interface

An SSP peripheral is attached to ANT in hardware, so somewhere we must link it up in firmware. The ANT processor uses SPI with flow control (SRDY and MRDY), and it is the Master in the SPI system (it’s uncommon to come across a Master device!). Having flow control is both good and bad. It’s good because we can regulate the flow of messages both to and from the ANT device. It’s bad because our SPI driver must accommodate flow control. The way ANT uses the SPI CS is a bit of a hack to make it work, so we also need to manage that. ANT refers to CS as “SEN” and the signaling is close but not identical to CS in an official SPI. There is also a hardware reset line that the Host can use to hard boot the ANT processor.

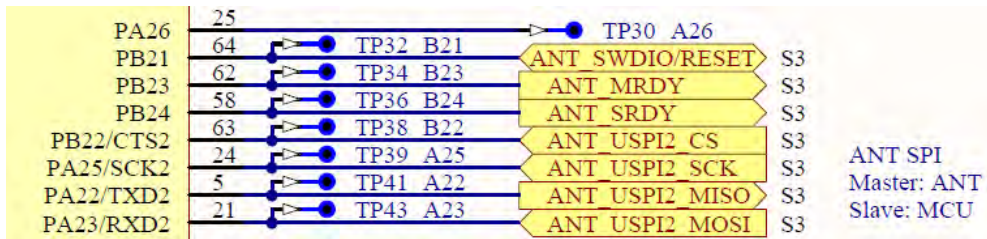


Figure 14-12 ANT hardware connections

We also consider that ANT messaging is slow, relatively speaking. Even a system running flat-out at 60kbps is not that fast. Most ANT systems will run only a few Hz, so we have tons of time to respond to message send requests. When ANT does send a message frame, it does so at 2MHz, so to get the actual bytes we can go very fast but still must manage the flow control and how ANT uses SEN.

The system we designed is still interrupt-based for SEN and still uses the SPI peripheral. However, we manually manage the SEN line and mirror it with a flag bit called `_SSP_CS_ASSERTED` because the signal needs to be used in several parts of the program to control the state machine. To manage flow control at high speed, callback functions for both transmit and receive are used inside the ISR.

First, we wrote several macros to handle the lowest level code to the ANT hardware pins. You can find these in `ant.h`.

```

/*****
Macros
*****/
#define IS_SEN_ASSERTED()      (ANT_SSP_FLAGS & _SSP_CS_ASSERTED)
#define ACK_SEN_ASSERTED()    (ANT_SSP_FLAGS &= ~_SSP_CS_ASSERTED)

#define IS_MRDY_ASSERTED()     (ANT_MRDY_READ_REG == 0)
#define SYNC_MRDY_ASSERT()    (ANT_MRDY_CLEAR_REG)
#define SYNC_MRDY_DEASSERT()  (ANT_MRDY_SET_REG)

#define SYNC_SRDY_ASSERT()     (ANT_SRDY_CLEAR_REG)
#define SYNC_SRDY_DEASSERT()  (ANT_SRDY_SET_REG)

#define ANT_RESET_ASSERT()     (ANT_RESET_CLEAR_REG)
#define ANT_RESET_DEASSERT()   (ANT_RESET_SET_REG)

```

These in turn reference the specific hardware pins set in `configuration.h` so that `ant.c` is abstracted as much as possible from the hardware.

```

#define ANT_MRDY_READ_REG      (AT91C_BASE_PIOB->PIO_PDSR & PB_23_ANT_MRDY)
#define ANT_MRDY_CLEAR_REG    (AT91C_BASE_PIOB->PIO_CODR = PB_23_ANT_MRDY)
#define ANT_MRDY_SET_REG      (AT91C_BASE_PIOB->PIO_SODR = PB_23_ANT_MRDY)

#define ANT_SRDY_CLEAR_REG     (AT91C_BASE_PIOB->PIO_CODR = PB_24_ANT_SRDY)
#define ANT_SRDY_SET_REG      (AT91C_BASE_PIOB->PIO_SODR = PB_24_ANT_SRDY)

#define ANT_RESET_CLEAR_REG    (AT91C_BASE_PIOB->PIO_CODR = PB_21_ANT_RESET)
#define ANT_RESET_SET_REG      (AT91C_BASE_PIOB->PIO_SODR = PB_21_ANT_RESET)

```

The MRDY pin can be controlled in the main 1ms loop since ANT needs to respond and the system needs to take other actions. In other words, it's quite slow and things need to happen between when it gets asserted and when it is de-asserted. This is also true for the

RESET pin. In both cases, no additional functions are needed to operate these signals.

SRDY is different – this is a short signal pulse that is provided to ANT to indicate our Host is ready after some event like SEN being asserted or the reception of a byte. The pulse can't be TOO short or else the ANT device may not see the pulse. It also can't happen too quickly after a byte is received – this is documented in the ANT specification. We wrote a function to assert and de-assert SRDY with appropriate time padding using constants that could easily be changed if required.

```
static void AntSrdyPulse(void)
{
    for(u32 i = 0; i < ANT_SRDY_DELAY; i++);
    SYNC_SRDY_ASSERT();

    for(u32 i = 0; i < ANT_SRDY_PERIOD; i++);
    SYNC_SRDY_DEASSERT();
} /* end AntSrdyPulse() */
```

This function simply blocks and thus drags out our ISR, but it is only about 20us long in total so no worries about system timing here.

The SSP task must be able to communicate to the ANT system, so next, we must tie in the SSP interrupts to the ANT task. Since configuration.h is our “melting pot” of definitions to abstract specific hardware details from more generic driver functions, we claim the SSP2 Application Flags variable for the ANT task with this definition since that is how the Host is physically connected to the nRF51422:

```
#define ANT_SSP_FLAGS          G_u32Ssp2ApplicationFlags
/* Assigns the correct global Application Flags to a self-documenting symbol */
```

G\_u32Ssp2ApplicationFlags is how the SSP ISR communicates what's happening from the ISR to an external task. It does this through the following flags:

```
/* G_u32Ssp2ApplicationFlags */
#define _SSP_CS_ASSERTED          (u32)0x00000001
/* INTERRUPT CONTROLLED ONLY: mirrors the CS line status to the application */

#define _SSP_TX_COMPLETE          (u32)0x00000002
/* Set when expected bytes have been transmitted; cleared automatically when new
message begins or can be cleared by application */

#define _SSP_RX_COMPLETE          (u32)0x00000004
/* Set when expected bytes have been received; cleared automatically on CS or can
be cleared by application */

#define _SSP_RX_OVERFLOW          (u32)0x00000008
/* Set if receiver overflows; cleared by application */
/* end G_u32Ssp2ApplicationFlags */
```

### AntTxFlowControlCallback(void) and AntRxFlowControlCallback(void)

When coding the ANT sub-system, we had to introduce the SPI\_SLAVE\_FLOW\_CONTROL SSP mode. Most of the data transmitted is done one byte at a time with interrupts. After each byte, some ANT-specific code must run. Callbacks are a common solution to a problem like this. We consider them Protected functions since they are used in very specific circumstances.

Whenever a byte finishes sending, `AntTxFlowControlCallback` is called. The main job of this function is to pulse the SRDY line per the ANT SPI interface. The function also keeps track of how many times it's called for debugging purposes.

```
void AntTxFlowControlCallback(void)
{
    /* Count the byte and toggle flow control lines */
    Ant_u32TxByteCounter++;
    AntSrdyPulse();
} /* end AntTxFlowControlCallback() */
```

Receiving data is very similar, with most of the data being received through the SSP interrupts. Just like the transmit case, the SSP ISR uses the callback function that is assigned when the SSP module is requested for ANT.

```
void AntRxFlowControlCallback(void)
{
    /* Count the byte and safely advance the receive buffer pointer; this is called from
    The RX ISR, so it won't be interrupted and break Ant_pu8AntRxBufferNextChar */
    Ant_u32RxByteCounter++;
    Ant_pu8AntRxBufferNextChar++;
    if(Ant_pu8AntRxBufferNextChar == &Ant_au8AntRxBuffer[ANT_RX_BUFFER_SIZE])
    {
        Ant_pu8AntRxBufferNextChar = &Ant_au8AntRxBuffer[0];
    }

    /* Only toggle SRDY if a reception is flagged in progress */
    if( G_u32AntFlags & _ANT_FLAGS_RX_IN_PROGRESS )
    {
        AntSrdyPulse();
    }
} /* end AntRxFlowControlCallback() */
```

The Rx callback function increments the ANT Rx buffer pointer. The SSP peripheral has a pointer to this pointer, so it is safe to increment it here. This is how we keep as clean a separation between the SSP and ANT tasks as possible. The callback function also calls `AntSrdyPulse()` but this is conditional on a flag. This detail came out of a problem to solve that emerged as the ANT protocol was integrated. We'll discuss that next.

### Managing ANT-Host messaging

In both the transmit and receive cases, we said: “most” of the data is exchanged with interrupts. Herein lies a bit of a trick with ANT SPI messaging. Regardless of Host to ANT transmit or ANT to Host receive, the first byte is always the ANT message SYNC byte, and it is always sent from ANT to the Host. The SYNC byte will indicate who gets to send data. This is the necessary detail to resolve the inevitable problem of, “what if ANT and the Host both want to send each other data at the same time?”

There are three cases to consider:

1. Just ANT wants to send a message:
  - a. ANT asserts SEN
  - b. Host pulses SRDY
  - c. ANT sends MESH\_TX\_SYNC byte (0xA4)

- a. Host keeps pulsing SRDY until all bytes are received and SEN is de-asserted
2. Just the Host wants to send a message:
  - a. Host asserts MRDY
  - b. ANT asserts SEN
  - c. Host pulses SRDY (and de-asserts MRDY)
  - d. ANT sends MESH\_RX\_SYNC byte (0xA5)
  - e. Host loads data and keeps pulsing SRDY until all bytes are sent and SEN is de-asserted
3. Both the Host and ANT want to send a message (the Host is slightly ahead of ANT):
  - a. Host asserts MRDY
  - b. ANT asserts SEN
  - c. Host pulses SRDY (and de-asserts MRDY)
  - d. ANT sends MESH\_TX\_SYNC byte (0xA4)
  - e. The Host must cancel its idea of sending a message to ANT and instead receive the message that ANT wants to send.

The first two scenarios are simple, but the last is a bit tricky to deal with. The Host is essentially interrupted, as ANT gets priority, and hence the introduction of our first ANT flag. We will also introduce an RX and TX IN\_PROGRESS flag. These are all part of `G_u32AntFlags` in `ant.h`

```
/* G_u32AntFlags */
/* Control flags */
#define _ANT_FLAGS_RX_IN_PROGRESS      (u32)0x01000000
/* Set when an ANT frame reception starts */

#define _ANT_FLAGS_TX_IN_PROGRESS      (u32)0x02000000
/* Set when an ANT frame transmission starts */

#define _ANT_FLAGS_TX_INTERRUPTED      (u32)0x04000000
/* An attempt to transmit was interrupted */
/* end G_u32AntFlags */
```



Draw a picture of the low-level data flow, function calls, and flags. Based on your knowledge of ANT so far, plan out how you would receive an ANT message.

### AntRxMessage()

```
/*!-----
@fn static void AntRxMessage(void)

@brief Completely receive a message from ANT to the Host.

Incoming bytes are deposited directly into the receive buffer from the SSP ISR
which should be extremely fast and complete in a maximum of 500us.

Requires:
- _SSP_CS_ASSERTED is set indicating a message is ready to come in
- G_u32AntFlags _ANT_FLAGS_TX_INTERRUPTED is set if the system wanted to transmit
```

```
    but ANT wanted to send a message at the same time (so MSG_TX_SYNC has already
    been received); _SSP_RX_COMPLETE must still be set from this.
```

```
- Ant_pu8AntRxBufCurrentChar points to the first byte of the message
```

```
Promises:
```

```
- If a good new message has been received, then Ant_u8AntNewMessages is incremented
  and the message is at Ant_pu8AntRxBufUnreadMsg in Ant_au8AntRxBuf
```

```
- If a good message is not received, then Ant_u8AntNewMessages is unchanged.
```

```
- In both cases, Ant_pu8AntRxBufNextChar points at the next empty buffer loca-
  tion
```

```
*/
```

Coding AntTxMessage and AntRxMessage are probably the most difficult part of the whole driver. Both of these functions are Private to ant.c because the API will provide simplified access to sending and receiving message data. We start with Rx since receiving is involved in both and it's slightly simpler.

First, we need a place for the raw data to go that we can connect to the SSP driver. We'll use a circular buffer large enough to handle enough incoming bytes – this is set by ANT\_RX\_BUFFER\_SIZE and 256 bytes should be plenty as that's at least 15-20 messages. Since every ANT message starts with a SYNC byte, contains a length variable, and ends in a checksum, managing the incoming data with this style of buffer makes sense as we can always parse it out and find the complete message frames. A "NextChar" pointer is also created for use with the SSP.

```
static u8 Ant_au8AntRxBuf[ANT_RX_BUFFER_SIZE];
```

```
/* Space for verified received ANT messages */
```

```
static u8 *Ant_pu8AntRxBufNextChar;
```

```
/* Pointer to next char to be written in the AntRxBuf */
```

We also need pointers to keep track of the current byte we're parsing, and where the next unread message is. Both pointers are local to the ANT task and not used by the SSP module. We will also keep a count of how many messages we think are in the buffer.

```
/* Pointer to the current char in the AntRxBuf */
```

```
static u8 *Ant_pu8AntRxBufUnreadMsg;
```

```
/* Pointer to unread chars in the AntRxBuf */
```

```
static u8 Ant_u8AntNewRxMessages;
```

```
/* Counter for number of new messages in AntRxBuf */
```

The code for AntRxMessage can be nicely designed in a flow chart that is developed from the messaging protocol documentation. We can match up the code with comments directly from the function to see what is happening. Let's annotate a few sections further to explain some of the considerations and finer details that were required.

Since AntRxMessage is triggered when ANT is ready to send the message, the function will receive all of the bytes while it blocks any other tasks (SSP interrupts and the ANT callbacks will run to get the bytes). ANT clocks the SPI bus at about 2MHz, so the data takes no time and even with all the SRDY pulses, the longest message should take a maximum of 600us. ANT messages occur relatively infrequently in the overall system, so this is a pretty safe assumption that our loop timing will not be violated.

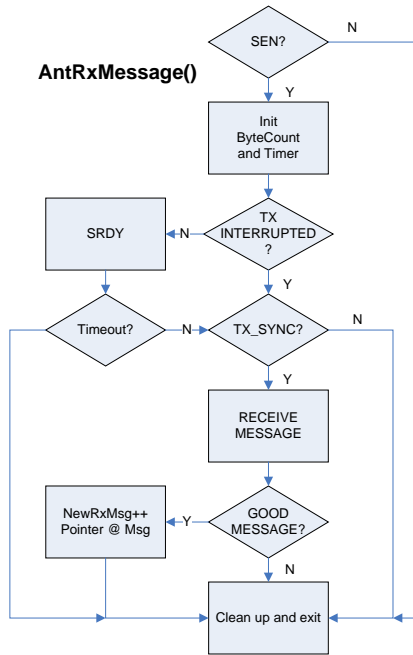


Figure 14-13 AntRxMessage() flowchart

If the flag `_ANT_FLAGS_TX_INTERRUPTED` is set, that means that `AntTxMessage()` has run before this but when the Host asked for the SYNC byte ANT told the Host it has a byte to send.

```

/* If the Global _ANT_FLAGS_TX_INTERRUPTED flag has been set, then we have already read the
TX_SYNC byte */
if(G_u32AntFlags & _ANT_FLAGS_TX_INTERRUPTED)
{
    /* Clear flag and adjust the starting byte counter*/
    G_u32AntFlags &= ~_ANT_FLAGS_TX_INTERRUPTED;
    u32CurrentRxByteCount--;
}

```

That SYNC byte would be sitting in the `RxBUFFER` unprocessed, and the subsequent SRDY pulse is not yet provided. This is really the most confusing part about managing ANT transmission and reception. It occurs all the time, so it's not something you can ignore. Due to the way we're counting bytes, we reduce `u32CurrentRxByteCount` by 1 since we initialized it to the current `Ant_u32RxByteCounter`. `Ant_u32RxByteCounter` is incremented every time a byte is received but since that first byte was received outside of `AntRxMessage()` it is off-by-one. This is quite a dangerous piece of code because it requires knowledge about a different function that was run. Therefore, we document it carefully in the code.

If an attempted Host-to-ANT transfer was not interrupted, then this is a regular ANT to Host message so the SYNC byte must be read. This is done under the watch of a timeout. When complete, the `RxBUFFER` should have the correct SYNC byte regardless of what happened before the call to `AntRxMessage()`. The byte counter is also synchronized now. In other words, we can receive the rest of the message without having to worry about whether or not a transmit was interrupted.



```

/* Otherwise we need to first read the sync byte */
else
{
    /* Cycle SRDY to get the first byte */
    AntSrdyPulse();

    /* Read the first byte when it comes in */
    while( !(ANT_SSP_FLAGS & _SSP_RX_COMPLETE) &&
           (Ant_u32RxTimer < ANT_ACTIVITY_TIME_COUNT) )
    {
        Ant_u32RxTimer++;
    }
}

```

AntRxMessage must manage the SSP flags and the `_ANT_FLAGS_RX_IN_PROGRESS` is used in the RxCallback function. Again, we have code that requires a fair bit of knowledge about what's happening in different functions, so we must document it carefully. With the SSP and AntFlags managed properly, the remaining reception takes places automatically through interrupts and is monitored by a timeout check.

```

/* _SSP_RX_COMPLETE flag will be set and the Rx callback will have run.
The callback does NOT toggle SRDY yet. _SSP_RX_COMPLETE should still
be set from AntTxMessage if that's what got us here. */
ANT_SSP_FLAGS &= ~_SSP_RX_COMPLETE;

/* One way or the other, we now have a potential SYNC byte at
Ant_pu8AntRxBufferCurrentChar. */
if (*Ant_pu8AntRxBufferCurrentChar == MSG_TX_SYNC)
{
    /* Flag that a reception is in progress */
    G_u32AntFlags |= _ANT_FLAGS_RX_IN_PROGRESS;

    /* Cycle SRDY to get the next byte (length) */
    AntSrdyPulse();
}

```

When ANT is finished sending it will de-assert SEN. If you are writing your own ANT code, this is a really important step in the state machine. If SEN remains asserted, then ANT has not clocked out all the data it wants to send. That means it needs more SRDY pulses to send more bytes. A stuck SEN line suggests you are not properly calculating or otherwise receiving the data from ANT. If ANT appears to be stuck, an error routine that pulses SRDY until SEN de-asserts is recommended.

```

/* We know the message is fully received when SEN is de-asserted. */
while( IS_SEN_ASSERTED() && (Ant_u32RxTimer < ANT_ACTIVITY_TIME_COUNT) )
{
    Ant_u32RxTimer++;
}

/* One way or another, this Rx is done! */
G_u32AntFlags &= ~_ANT_FLAGS_RX_IN_PROGRESS;
ANT_SSP_FLAGS &= ~_SSP_RX_COMPLETE;

```

Validating the data and taking care of all the counters, pointers, and flags can be confusing. While the code is simple, the number of things happening makes it difficult. We have a saved value of `Ant_u32RxByteCounter` stored in `u32CurrentRxByteCount` (and adjusted if we had `TX_INTERRUPTED`). Now we can get the received byte count by the difference of the current minus the saved counters. This is important because it gives us the length of the received message even though the message itself should contain

the LENGTH field, but we don't know if there were errors. At this point, you can't really rely on the checksum because to find the checksum byte you need to know the length of the message. We are going to assume the length byte is correct and then pick out the checksum byte based on that, and then verify everything.

```
/* Check that the above loop ended as expected and didn't time out */
if(Ant_u32RxTimer < ANT_ACTIVITY_TIME_COUNT)
{
    /* Update counter to see how many bytes we should have */
    u32CurrentRxByteCount = Ant_u32RxByteCounter - u32CurrentRxByteCount;

    /* RxBufferCurrentChar is still pointing to the SYNC byte. Validate what should
    be a complete message now. */
    u8Checksum = *Ant_pu8AntRxBufferCurrentChar;
    AdvanceAntRxBufferCurrentChar();
    ...
}
```

We use a lot of commenting to keep track of the code, especially when adding in seemingly random numbers to make adjustments. Once we verify the LENGTH byte, we can start to use it to finish verifying the message. Frankly, in years of using this code, we've never seen a bit error here, but that doesn't mean it can't happen. Any important system that you code must have excellent data integrity checking.

```
/* Read the length byte and add two to count the length byte and message ID but
not checksum as length will be our checksum counter */
u8Length = *Ant_pu8AntRxBufferCurrentChar + 2;

/* Optional check (u8Length does not include SYNC Checksum so add 2) */
if(u32CurrentRxByteCount != (u8Length + 2) )
{
    /* Could throw out the message right away */
    G_u32AntFlags |= _ANT_FLAGS_LENGTH_MISMATCH;
}

/* Validate the remaining bytes based on u8Length*/
do
{
    u8Checksum ^= *Ant_pu8AntRxBufferCurrentChar;
    AdvanceAntRxBufferCurrentChar();
} while (--u8Length);
```

Once the message checksum has been calculated, we can check it. At this point, we have sufficient redundancy and verification to have confidence in the number, though anyone could argue that a single byte checksum still has a 1 in 256 chance of being accidentally correct. Notice the careful management of pointers and documentation of where they're at.

```
/* AntRxBufferCurrentChar is pointing to the last received byte that should be
the checksum. */
if (u8Checksum == *Ant_pu8AntRxBufferCurrentChar)
{
    Ant_u8AntNewRxMessages++;
    Ant_DebugTotalRxMessages++;
}
/* If message isn't good, move Ant_pu8AntRxBufferUnreadMsg passed garbage data */
else
{
    Ant_pu8AntRxBufferUnreadMsg = Ant_pu8AntRxBufferCurrentChar;
    AdvanceAntRxBufferUnreadMsgPointer();
}
```

```

    }
} /* end if(Ant_u32RxTimer < ANT_ACTIVITY_TIME_COUNT) */

```

The rest of `AntRxMessage()` handles various errors that might have been found in the process. Where this system lacks is responding to different errors. There are a few flags that are reported back, but really the main piece of information is `Ant_u8AntNewRxMessages` that is incremented if no errors have occurred. Again, in a system that must be more robust, we would design in better error handling, though at this low level of firmware, you are indeed limited in what you can do.

### AntAbortMessage()

```

/*-----
static void AntAbortMessage(void)

Kills the current message in progress with ANT and resets all the pointers.

Any existing received buffer data is lost. Interrupts are disabled
during this time to ensure pointers do not get moved by
an outside function.

Requires:
- NONE

Promises:
- Ant_pu8AntRxBufferNextChar, Ant_pu8AntRxBufferCurrentChar, Ant_pu8AntRxBufferUn-
readMsg, and Ant_u8AntNewRxMessages reset.
*/

```

`AntAbortMessage` is an interrupt-safe function to call to quit in the middle of trying to receive a message from ANT. It is only called from `AntRxMessage` in the event of a timeout and essentially should never happen.

There is no attempt to recover anything beyond the message buffers and counters for receiving ANT messages, and since it zeros everything any other unprocessed message would be lost. We could have a large debate about what should or could happen in this case, but that would likely be more task-specific with everything from complex recovery attempts, to simply resetting ANT or the whole system if something goes awry.

### AntTxMessage()

```

/*!-----
bool AntTxMessage(u8 *pu8AntTxMessage_)

Send a message from the Host to the ANT device.

To do this, we must tell ANT that we have a message to send by asserting MRDY,
wait for ANT to acknowledge with SEN, then read a byte from ANT to confirm the
transmission can proceed. If ANT wants to send a message at the
same time, the byte it sends will be an Rx byte so the AntTxMessage must suspend
and go read the incoming message first. The Tx process would restart after
ANT's message has been received by the Host.

Once ANT confirms that the Host may transmit, the message to transmit is queued
and data is sent byte-by-byte with SRDY used for flow control after each byte.
Due to the speed of the chip-to-chip communications, even the longest ANT message
should be able to send in less than 500us so it will likely be done on the main
program cycle that immediately follows this call.

```

**Requires:**

- pu8AntTxMessage\_ points to an Ant formatted message where the first data byte is the length byte (since ANT sends the SYNC byte) and the last byte is the checksum.

**Promises:**

- MRDY is de-asserted
- Returns TRUE if the transmit message is queued successfully;  
Ant\_u32CurrentTxMessageToken holds the message token
- Returns FALSE if the transfer couldn't start or if receive message interrupted (G\_u32AntFlags \_ANT\_FLAGS\_TX\_INTERRUPTED is set;  
AntRxBufferCurrentChar pointing to the received byte).

\*/

To transmit a message, we need the message itself and we should tell the calling task if the transfer is successful. As you saw with many data transfer tasks in this system, “successful” does not mean the data has been sent, but instead it means it has been successfully queued to the messaging task. That is true here, though AntTxMessage also starts the initial preamble with the ANT device and receives the SYNC byte to confirm it can proceed to transfer. We do everything we can to get the system set up to be able to hand off to the SSP task to send the data (with handshaking). Again, let’s start with a flowchart to capture the high-level steps of the function.

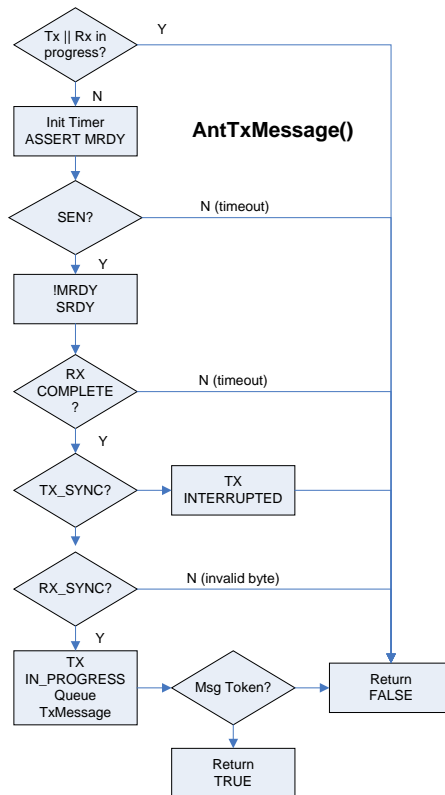


Figure 14-14 AntTxMessage() flowchart

The first four steps in the flowchart are straightforward. We make use of the `G_u32AntFlags` as a basic protection against tasks calling `AntTxMessage` more than once at a time. The flags must be carefully maintained. All functions dependent on the ANT device have a timeout assigned. We follow the protocol to manage MRDY and SRDY to signal ANT that we have a message and to start the transfer of the first byte from ANT to Host which is the SYNC byte confirming we can proceed.

```
/* Check G_u32AntFlags first to ensure a transmission is not already taking place */
if(G_u32AntFlags & (_ANT_FLAGS_TX_IN_PROGRESS | _ANT_FLAGS_RX_IN_PROGRESS) )
{
    DebugPrintf(au8TxInProgressMsg);
    return FALSE;
}

/* Initialize timeout and notify ANT that Host wishes to send a message */
Ant_u32RxTimer = 0;
SYNC_MRDY_ASSERT();

/* Wait for SEN to be asserted indicating ANT is ready for a message */
while ( !IS_SEN_ASSERTED() && (Ant_u32RxTimer < ANT_ACTIVITY_TIME_COUNT) )
{
    Ant_u32RxTimer++;
}

/* If we timed out, then clear MRDY and exit */
if(Ant_u32RxTimer > ANT_ACTIVITY_TIME_COUNT)
{
    SYNC_MRDY_DEASSERT();
    DebugPrintf(au8TxTimeoutMsg);
    return(FALSE);
}
/* Else we have SEN flag; queue to read 1 byte after delay before toggling SRDY */
AntSrdyPulse();

/* Wait for the first byte to come in via the ISR / Rx Callback*/
while( !(ANT_SSP_FLAGS & _SSP_RX_COMPLETE) &&
        (Ant_u32RxTimer < ANT_ACTIVITY_TIME_COUNT) )
{
    Ant_u32RxTimer++;
}

/* Ok to deassert MRDY now */
SYNC_MRDY_DEASSERT();

/* If we timed out now, then clear MRDY and exit */
if(Ant_u32RxTimer > ANT_ACTIVITY_TIME_COUNT)
{
    DebugPrintf(au8TxTimeoutMsg);
    return(FALSE);
}
```

Remember that the SYNC byte is from the ANT's perspective. If it sends `MESG_TX_SYNC` that means that ANT wants to send a message to the Host, and thus our attempted transfer gets interrupted (`_ANT_TX_INTERRUPTED` gets set). If ANT confirms that we may transfer our byte, it tells the Host it is ready to receive by sending `MESG_RX_SYNC`. Notice the long comment about what is happening with the SSP flag and the RX call back.

```

/* When the byte comes in, the SSP module will set the _SSP_RX_COMPLETE flag and
also call the Rx callback but does not toggle SRDY at this time. We must look at
this byte to determine if ANT initiated this communication and is telling us that
a message is coming in, or if we initiated the communication and ANT is allowing us
to transmit. */

/* Read the byte - don't advance the pointer yet */
u8Byte = *Ant_pu8AntRxBufferCurrentChar;

/* If the byte is TX_SYNC, ANT wants to send a message which must be done first */
if (u8Byte == MSG_TX_SYNC)
{
    G_u32AntFlags |= _ANT_FLAGS_TX_INTERRUPTED;
    return(FALSE);
}

```

Manage the pointers carefully. The two functions used to advance the pointers check for proper wrap-around in the circular Rx buffer. These functions are a little too long to inline, and we code one for each variable to avoid passing parameters. We'll leave looking at the code as an exercise.

```

/* Rx byte is in our Rx buffer, advance both pointers since it's not an
incoming message */
AdvanceAntRxBufferCurrentChar();
AdvanceAntRxBufferUnreadMsgPointer();

/* Clear the status flag and process the byte */
ANT_SSP_FLAGS &= ~_SSP_RX_COMPLETE;

```

Once the RX\_SYNC is received, we manage the TX\_IN\_PROGRESS flag and queue the message data for transmit using the calculated length. The system is carefully coded to only allow one transfer at a time, and we make use of this by capturing the message token in a global variable `Ant_u32CurrentTxMessageToken` that we can monitor in the ANT state machine. We know this by experience with the SSP module so it was part of our initial design plan when the high-level state diagram was created.

```

/* If the byte is RX_SYNC, then proceed to send the message */
if (u8Byte == MSG_RX_SYNC)
{
    /* Flag that a transmit is in progress */
    G_u32AntFlags |= _ANT_FLAGS_TX_IN_PROGRESS;

    /* Read the message length + three for length, message ID and checksum bytes */
    u32Length = (u32)(pu8AntTxMessage_[0] + 3);

    /* Queue the message to the peripheral and capture the token */
    Ant_u32CurrentTxMessageToken = SspWriteData(Ant_Ssp, u32Length, pu8AntTxMessage_);
    /* Return TRUE if we received a message token for the queued message */
    if (Ant_u32CurrentTxMessageToken != 0)
    {
        return(TRUE);
    }
    else
    {
        DebugPrintf(au8TxNoTokenMsg);
        return(FALSE);
    }
}

```

We can build all of this code but we don't have a system that can be tested yet. The next thing we need to do is initialize and test the communication between the Host and ANT.

### AntSyncSerialInitialize()

```

/*!-----
static void AntSyncSerialInitialize(void)

Properly sets up the ANT SPI interface and tests Host <-> ANT communications.

Requires:
- ANT_SPI is configured
- !CS (SEN) interrupt should be enabled

Promises:
- Ant_pu8AntRxBufferNextChar is initialized to start of AntRxBuffer
- Ant_pu8AntRxBufferUnreadMsg is initialized to start of AntRxBuffer
- Message counter Ant_u8AntNewRxMessages reset to 0;
- If ANT starts up correctly and responds to version request, then
  G_u32SystemFlags _APPLICATION_FLAGS_ANT is set and Ant_u8AntVersion is populated
  with the returned version information from the ANT IC.

*/

```

The whole serial-layer is made ready by a call to this initialization function. This function is intended to run during system initialization, so there is no worry about timing or blocking. It does require the SSP driver which we know is configured to clock out complete messages during initialization mode.

The nRF51422 is easy to initialize since the SAM3U2 has control of the nRF51422 reset line. We assert RESET and make sure both MRDY and SRDY are de-asserted. After a short amount of time, the RESET line is released. The ANT device will boot and within a few milliseconds, it should send a message to say it has reset. Monitor SEN to check for this message.

```

/* Initialize buffer pointers */
Ant_pu8AntRxBufferNextChar = Ant_au8AntRxBuffer;
Ant_pu8AntRxBufferCurrentChar = Ant_au8AntRxBuffer;
Ant_pu8AntRxBufferUnreadMsg = Ant_au8AntRxBuffer;
Ant_u8AntNewRxMessages = 0;

/* Reset the 51422 and initialize SRDY and MRDY */
u32EventTimer = G_u32SystemTime1ms;
ANT_RESET_ASSERT();
SYNC_MRDY_DEASSERT();
SYNC_SRDY_DEASSERT();
while( !IsTimeUp(&u32EventTimer, ANT_RESET_WAIT_MS) );
ANT_RESET_DEASSERT();
u32EventTimer = G_u32SystemTime1ms;
while( !IsTimeUp(&u32EventTimer, ANT_RESTART_DELAY_MS) );

/* ANT should want to send message 0x6F now to indicate it has reset */
u32EventTimer = G_u32SystemTime1ms;
while( !IS_SEN_ASSERTED() && !bErrorStatus )
{
    bErrorStatus = IsTimeUp(&u32EventTimer, ANT_MSG_TIMEOUT_MS);
}

```

If that is successful without any timeouts, then we can receive the message from ANT with a call to `AntRxMessage()`.

```
/* SEN is asserted if bErrorStatus is FALSE */
if (!bErrorStatus)
{
    /* Receive and process what should be the restart message */
    AntRxMessage();
}
```

`AntRxMessage()` should return after receiving a complete message so you can set up a breakpoint immediately after the function call. This is the first opportunity to test if the fundamental communication with ANT is working. The “RESET” message ID is 0x6F, so you can see below that we have one new message with SYNC byte 0xA4 (ANT to Host), length 1, message 0x6F, startup reason 0x01 (hardware reset line per the message description on page 93 in the ANT protocol document) and checksum.

Expression	Value	Location
Ant_u8AntNewRxMessages	1	0x2000263B
Ant_pu8AntRxBufferUnreadMsg	0x20001F50 "..."	0x2000252C
Ant_pu8AntRxBufferCurrentChar	0x20001F55 ""	0x20002528
Ant_pu8AntRxBufferNextChar	0x20001F55 ""	0x20002524
Ant_au8AntRxBuffer	<array> "..."	0x20001F50
[0]	'.' (0xA4)	0x20001F50
[1]	'.' (0x01)	0x20001F51
[2]	'o' (0x6F)	0x20001F52
[3]	'.' (0x01)	0x20001F53
[4]	'.' (0xCB)	0x20001F54
[5]	'\0' (0x00)	0x20001F55

Figure 14-15 ANT to Host data

There is a lot going on here on many layers in the firmware, so celebrate when this works! When we first wrote the code, of course it didn't work so we back-tracked until we found out why. There were various problems with the SSP driver as we updated it to work in `SPI_SLAVE_FLOW_CONTROL` mode but all were relatively easy to fix. We kept a logic analyzer on all the ANT – HOST hardware lines so we could watch all the signals and determine if the hardware was behaving, or if it appeared to be the firmware not functioning correctly.

We should do something with that message or else it will just be sitting around in the receive buffer. We'll write a really fun function called `AntProcessMessage()` to do that. For now, you can just put in the declaration and this hook so the code builds.

```
AntProcessMessage();
```

Finally, send a message that requests the ANT version ID string. Though we know what device we should be talking to, verifying it – or at least having the information in the system – is a very wise thing to do. This is our first use of `AntTxMessage` and we must supply the correct message data to send.

Since messages take up a reasonable amount of space and we might want to use them at any time by any task in the system, we define them globally in `ant.c`. The version request message is available through a `MESG_REQUEST_ID`. This symbol can be found in `antmessage.h` (it is 0x4D). We can look up the format for this message in the ANT Message Summary table in the ANT Message Protocol and Usage document.



Class	Type	Reply	From	Length	Msg ID	Message Content		
Control Messages	Request Message 9.5.4.4 page 95	Yes	Host	Varies	0x4D	Channel Number/ Sub Message ID	Requested Message ID	Additional fields used with some requests

**Figure 14-16 Message summary**

Some of the fields vary depending on what information is being requested. Look back to the ANT Message Summary table to find the Requested Message ID. Find “Requested Response Messages” and the code for ANT Version (0x3E). You can find the symbol in `antmessage.h` which is `MESG_VERSION_ID`.

From there we can build the message in an array that follows the ANT message format. The byte order is:

- [0]: LENGTH which is `MESG_REQUEST_SIZE`
- [1]: `MSG_ID` which is `MESG_REQUEST_ID`
- [2]: CHANNEL which is 0 when a specific channel doesn’t apply
- [3]: Requested message ID which is `MESG_VERSION_ID`
- [4]: CHECKSUM which we can set as 0 because we have a function to calculate it for us.

All together in the array it looks like this:

```
u8 G_au8ANTGetVersion[] = {MESG_REQUEST_SIZE, MESG_REQUEST_ID, 0, MESG_VERSION_ID, 0};
```

That is probably one of the worst first examples for creating an ANT message because it is quite complicated with all the variables and page flipping. Don’t worry, most of the ANT messages are quite straightforward as we’ll see shortly.

Once the message is constructed, calculate the checksum and queue it to send with `AntTxMessage`.

```
/* Send out version request message and expect response */
G_au8ANTGetVersion[4] = AntCalculateTxChecksum(&G_au8ANTGetVersion[0]);
AntTxMessage(&G_au8ANTGetVersion[0]);

if(AntExpectResponse(MESG_VERSION_ID, ANT_MSG_TIMEOUT_MS))
{
    DebugPrintf("ANT init Version ID message failed\n\r");
    bErrorStatus = TRUE;
}
```

When the code above runs successfully, you’ll see the 0xA5 byte from the Transmit request in the `AntRxBuffer` followed by the full 0xA4 received message with the version string (message length 11, message 0x3E, “BDD0.02B01”).

[5]	'.'	(0xA5)	0x20001F55
[6]	'.'	(0xA4)	0x20001F56
[7]	'\v'	(0x0B)	0x20001F57
[8]	'>'	(0x3E)	0x20001F58
[9]	'B'	(0x42)	0x20001F59
[10]	'D'	(0x44)	0x20001F5A
[11]	'D'	(0x44)	0x20001F5B
[12]	'0'	(0x30)	0x20001F5C
[13]	'.'	(0x2E)	0x20001F5D
[14]	'0'	(0x30)	0x20001F5E
[15]	'2'	(0x32)	0x20001F5F
[16]	'B'	(0x42)	0x20001F60
[17]	'0'	(0x30)	0x20001F61
[18]	'1'	(0x31)	0x20001F62
[19]	'\0'	(0x00)	0x20001F63
[20]	'.'	(0x8C)	0x20001F64

Figure 14-17 Successful transmission (AntRxBuffer)

With all this working, you can have a high level of confidence that the low-level code for the ANT driver is functioning correctly. We simply print out the version string on the UART port during system initialization so it can be visually verified by the user. On a production system, this string should be compared to known, accepted values and an error flagged if it does not match. In any generic case, the Host system's functionality may be different depending on the sub-system versions that it discovers on startup.

Let's look deeper into the other functions that support AntSyncSerialInitialize.

### AntExpectResponse()

```

/*-----
u8 AntExpectResponse(u8 u8ExpectedMessageID_, u32 u32TimeoutMS_)

Waits a specified amount of time for a particular message to arrive from ANT in
response to a message sent to ANT. ***This function violates the 1ms system rule,
so should only be used during initialization.***

Requires:
- A message had been sent to ANT to which a response should be coming in
- Ant_u8AntNewRxMessages == 0 as this function is meant to run one-to-one with
  transmitted messages.
- SSP task should be in manual mode so it is busy sending the Tx message to which
  this function will wait for the ANT response.

- u8ExpectedMessageID_ is the ID of a message to which a response is expected
- u32TimeoutMS_ is the maximum value in ms to wait for the response

Promises:
- Returns 0 if the message is received and was successful
- Returns 1 if response never received or indicates message was not successful.
*/

```

You probably noticed the call to the function AntExpectResponse(). Since all the ANT startup code is running in the initialization section of the system, it made sense to create this function which gives time for the Tx message to send and then receives the expected message and verifies it.

AntExpectResponse() monitors the SEN line to determine when the transmit communication is done. If the system did not timeout, it cleans up the outgoing message buffer and waits for the expected SEN signal from ANT that indicates the response

message is ready to be sent to the Host.

```
while( IS_SEN_ASSERTED() && !bTimeout )
{
    bTimeout = IsTimeUp(&u32StartTime, ANT_MSG_TIMEOUT_MS);
}
if( !bTimeout )
{
    /* Done with this message token, so it can be cleared */
    G_u32AntFlags &= ~_ANT_FLAGS_TX_IN_PROGRESS;
    AntDeQueueOutgoingMessage();
    Ant_u32CurrentTxMessageToken = 0;

    /* Wait for SEN */
    u32StartTime = G_u32SystemTime1ms;
    while( !IS_SEN_ASSERTED() && !bTimeout )
    {
        bTimeout = IsTimeUp(&u32StartTime, ANT_MSG_TIMEOUT_MS);
    }
}
```

If the response message from ANT is signaled as expected, `AntRxMessage()` is called and then `AntRxBuffer` is inspected directly to check for the expected response. The expected response must be the correct type, have the correct ID, and include a `RESPONSE_NO_ERROR` code.

```
/* If no timeout then read the incoming message */
if( !bTimeout )
{
    AntRxMessage();

    /* If there is a new message in the receive buffer, then check that it is a response
    to the expected message and that the response is no error */
    if(Ant_u8AntNewRxMessages)
    {
        /* Check if the response is an Event, the event is a reply to the expected
        message, and the reply is good. Since Ant_pu8AntRxBufferUnreadMsg is pointing to
        the SYNC byte, add 1 when using BUFFER_INDEX values. */
        if(
            (*(Ant_pu8AntRxBufferUnreadMsg + MSG_ID_OFFSET) == MSG_RESPONSE_EVENT_ID)
            &&
            (*(Ant_pu8AntRxBufferUnreadMsg + MSG_RESPONSE_MSG_ID_OFFSET) ==
            u8ExpectedMessageID_)
            &&
            (*(Ant_pu8AntRxBufferUnreadMsg + MSG_RESPONSE_CODE_OFFSET) ==
            RESPONSE_NO_ERROR)
        )
        {
            u8ReturnValue = 0;
        }
    }
}
/* Process any message in the RxBuffer and return the result value */
AntProcessMessage();
return(u8ReturnValue);
```

The function exits by processing the message to clear it. We do not use this function as part of the main API because it blocks and because it is specific to ANT functioning correctly at startup. If we know ANT is alive and ready for other tasks in the system to use, then we offload message processing and confirmation to a higher level and within the regular 1ms processing loop.

**AntProcessMessage()**

```

/*!-----
static u8 AntProcessMessage(void)

Reads the latest received Ant message and updates system information accordingly.

Requires:
- Ant_u8AntNewRxMessages holds the number of unprocessed messages in the message queue
- Ant_pu8AntRxBufferUnreadMsg points to the first byte of an unread verified ANT message

Promises:
- Returns 1 if Ant_u8AntNewRxMessages == 0 or message exceeds maximum allowed length
- Returns 0 otherwise and:
  - Ant_u8AntNewRxMessages--
  - Ant_pu8AntRxBufferUnreadMsg points to first byte of next unread ANT message
  - System flags are updated

*/

```

Since we have referred to it several times already, let's look at `AntProcessMessage()` now. It is responsible for parsing out any message received from ANT and deciding what the message content is and therefore what to do with it. Note that the function requires that messages in the buffer are already verified per the ANT protocol. This is critically important for the success of such a function.

The function starts by confirming the presence of at least one new message and keeps a debug counter of all messages it processes. A copy of the message is taken from the circular Rx Buffer to ensure we can easily index it with standard indexing values already provided in the various ANT header files. Processing the message around the end of the buffer would otherwise be very awkward.

```

{
    u8 u8MessageLength;
    u8 u8Channel;
    u8 au8MessageCopy[MESG_MAX_SIZE];
    AntExtendedDataType sExtendedData;

    /* Exit immediately if there are no messages in the RxBuffer */
    if (!Ant_u8AntNewRxMessages)
    {
        return(1);
    }
    Ant_DebugProcessRxMessages++;

    /* Otherwise decrement the new message counter, and get a copy of the message
    since the rx buffer is circular and we want to index the various bytes using
    the ANT byte definitions. */
    Ant_u8AntNewRxMessages--;
    AdvanceAntRxBufferUnreadMsgPointer();
    u8MessageLength = *Ant_pu8AntRxBufferUnreadMsg;
    /* Check to ensure the message size is legit. !!!!! Clean up pointers if not */
    if (u8MessageLength > MESG_MAX_SIZE)
    {
        return(1);
    }

    /* Copy the message so it can be indexed easily */
    for (u8 i = 0; i < (u8MessageLength + MESG_FRAME_SIZE - MESG_SYNC_SIZE); i++)
    {

```

```
    au8MessageCopy[i] = *Ant_pu8AntRxBufferUnreadMsg;
    AdvanceAntRxBufferUnreadMsgPointer();
}
/* Note: Ant_pu8AntRxBufferUnreadMsg is now pointing at the next unread message */
```

Once we have the message copy, the function starts reading it as we know the message format is correct and follows the ANT protocol.

Like many protocol message parsing functions, `AntProcessMessage()` is implemented with a large switch statement. The condition of the switch is the message ID. From the introduction of the ANT protocol, we know `MSG_RESPONSE_EVENT_ID` is a special case. Depending on the message payload, the message can be either a:

1. Channel Response (typically a response to a Host-Ant message) or,
2. Channel Event (typically an unsolicited message from the ANT radio to the Host).

To determine this, look at the `RESPONSE_MSG_ID` in `au8MessageCopy`.

```
/* Decide what to do based on the Message ID */
switch( au8MessageCopy[BUFFER_INDEX_MSG_ID] )
{
    case MSG_RESPONSE_EVENT_ID:
    {
        /* Channel Message received: it is a Channel Response or Channel Event */
        if( au8MessageCopy[BUFFER_INDEX_RESPONSE_MSG_ID] != MSG_EVENT_ID )
        {
            /* We have a Channel Response: parse it out based on the message ID to which
             the response applies and post the result */
            G_stAntMessageResponse.u8Channel = u8Channel;
            G_stAntMessageResponse.u8MessageNumber =
                au8MessageCopy[BUFFER_INDEX_RESPONSE_MSG_ID];
            G_stAntMessageResponse.u8ResponseCode =
                au8MessageCopy[BUFFER_INDEX_RESPONSE_CODE];
        }
    }
}
```

For Channel Response messages, the global variable:

```
AntMessageResponseType G_stAntMessageResponse
```

is used to capture the channel, message number, and response code for the Channel Response. This is used by `ant_api.c` to determine if an expected message has been successful. This is the real-time functionality that is available during the main program instead of using `AntExpectResponse` as was used during initialization. It could also be accessed by another task if that task needed the information.

In many cases, `ant.c` can directly use a Channel Response to determine the state of the system and propagate information to `ant_api` or another user task in a more abstracted and thus understandable way. The Global `G_asAntChannelConfiguration` that holds all configured channel information also has an `AntFlags` member that is updated based on messages processed. Therefore, any task can get real-time status of any channel in the system without complicated API calls and interpreting messages that may or may not have originated from a traceable source. To parse and update this information, an additional switch statement is nested. This makes this part of the function quite long even though we have chosen only a few selected messages that require specific responses and actions. A full message processor would handle every possible message response.

```

switch(au8MessageCopy[BUFFER_INDEX_RESPONSE_MESG_ID])
{
    case MSG_OPEN_SCAN_CHANNEL_ID:
        DebugPrintf("Scanning ");
        /* Fall through */

        case MSG_OPEN_CHANNEL_ID:
            G_au8AntMessageOpen[12] = u8Channel + 0x30;
            DebugPrintf(G_au8AntMessageOpen);

            /* Only change the flags if the command was successful */
            if( au8MessageCopy[BUFFER_INDEX_RESPONSE_CODE] == RESPONSE_NO_ERROR )
            {
                G_asAntChannelConfiguration[u8Channel].AntFlags |= _ANT_FLAGS_CHANNEL_OPEN;
                G_asAntChannelConfiguration[u8Channel].AntFlags &=
                    ~_ANT_FLAGS_CHANNEL_OPEN_PENDING;
            }
            break;

    case MSG_CLOSE_CHANNEL_ID:
        G_au8AntMessageClose[12] = au8MessageCopy[BUFFER_INDEX_CHANNEL_NUM] + 0x30;
        DebugPrintf(G_au8AntMessageClose);

        /* Only change the flags if the command was successful */
        if( au8MessageCopy[BUFFER_INDEX_RESPONSE_CODE] == RESPONSE_NO_ERROR )
        {
            G_asAntChannelConfiguration[u8Channel].AntFlags &=
                ~(_ANT_FLAGS_CHANNEL_CLOSE_PENDING | _ANT_FLAGS_CHANNEL_OPEN);
        }
        break;

    case MSG_UNASSIGN_CHANNEL_ID:
        G_au8AntMessageUnassign[12] = au8MessageCopy[BUFFER_INDEX_CHANNEL_NUM] + 0x30;
        DebugPrintf(G_au8AntMessageUnassign);

        /* Only change the flags if the command was successful */
        if( au8MessageCopy[BUFFER_INDEX_RESPONSE_CODE] == RESPONSE_NO_ERROR )
        {
            G_asAntChannelConfiguration[u8Channel].AntFlags &=
                ~(_ANT_FLAGS_CHANNEL_CONFIGURED |
                  _ANT_FLAGS_CHANNEL_OPEN_PENDING |
                  _ANT_FLAGS_CHANNEL_CLOSE_PENDING |
                  _ANT_FLAGS_CHANNEL_OPEN);
        }

        break;
}

```

To date, we have not required specific handling of any other messages, so a Default case is used to capture any unhandled messages and simply display the information on the debug port. If a task relied upon a certain message, that would be an appropriate time to consider editing `ant.c` to handle the message internally. However, if it is very task specific then using `G_stAntMessageResponse` would suffice.

```

default:
    G_au8AntMessageUnhandled[12] =
        au8MessageCopy[BUFFER_INDEX_CHANNEL_NUM] + NUMBER_ASCII_TO_DEC;
    G_au8AntMessageUnhandled[24] =
        HexToASCIILower((au8MessageCopy[BUFFER_INDEX_RESPONSE_MESG_ID] >> 4) & 0x0F);
    G_au8AntMessageUnhandled[25] =
        HexToASCIILower((au8MessageCopy[BUFFER_INDEX_RESPONSE_MESG_ID] & 0x0F));
    G_au8AntMessageUnhandled[36] =
        HexToASCIILower((au8MessageCopy[BUFFER_INDEX_RESPONSE_CODE] >> 4) & 0x0F);
    G_au8AntMessageUnhandled[37] =

```

```

    HexToASCIILower( (au8MessageCopy[BUFFER_INDEX_RESPONSE_CODE] & 0x0F) );
    DebugPrintf(G_au8AntMessageUnhandled);
    break;
} /* end switch */

```

We also print out a confirmation message for messages to facilitate debugging.

```

/* All messages print an "ok" or "fail" */
if( au8MessageCopy[BUFFER_INDEX_RESPONSE_CODE] == RESPONSE_NO_ERROR )
{
    DebugPrintf(G_au8AntMessageOk);
}
else
{
    DebugPrintf(G_au8AntMessageFail);
    G_u32AntFlags |= _ANT_FLAGS_CMD_ERROR;
}
}

```

If the function is processing a `MESG_RESPONSE_EVENT_ID` that is a Channel Event (`MESG_EVENT_ID`), this means that the byte at `BUFFER_INDEX_RESPONSE_CODE` is an Event Code which is very important. Again, we choose to implement specific actions only against a few of the many available Event Codes. You'll see that we added a "verbose" mode to enable sending a debug message every time an Event Code is processed. This is useful for debugging a new task that is supposed to be reading ANT Events but might not be doing it correctly. Instead of a compiler switch, this would make a great debug task option.

```

/* The message is a Channel Event, so the Event Code must be parsed out */
else
{
    switch ( au8MessageCopy[BUFFER_INDEX_RESPONSE_CODE] )
    {
        case RESPONSE_NO_ERROR:
        {
            AntTickExtended(au8MessageCopy);
#ifdef ANT_VERBOSE
            DebugPrintf("\n\rRESPONSE_NO_ERROR\n\r");
#endif
            break;
        }

        case EVENT_RX_FAIL: /* Slave did not receive a message when expected */
        {
            /* The Slave missed a message it was expecting: communicate this to the
            application in case it matters. Could also queue a debug message here. */
            if(++Ant_u8SlaveMissedMessageLow == 0)
            {
                if(++Ant_u8SlaveMissedMessageMid == 0)
                {
                    ++Ant_u8SlaveMissedMessageHigh;
                }
            }

            /* Queue an ANT_TICK message to the application message list. */
            AntTickExtended(au8MessageCopy);
#ifdef ANT_VERBOSE
            DebugPrintf("\n\rEVENT_RX_FAIL\n\r");
#endif
            break;
        }
    }
}

```

```

    }

    case EVENT_RX_FAIL_GO_TO_SEARCH: /* Slave lost sync with Master (still open) */
    {
        /* Slave missed enough consecutive messages so goes back to search */
        AntTickExtended(au8MessageCopy);
#ifdef ANT_VERBOSE
        DebugPrintf("\n\rEVENT_RX_FAIL_GO_TO_SEARCH\n\r");
#endif
        break;
    }

    case EVENT_TX: /* ANT has sent a data message */
    {
        /* If Master device, EVENT_TX means it's time for next message */
        if(G_asAntChannelConfiguration[u8Channel].AntChannelType ==
            CHANNEL_TYPE_Master)
        {
            AntTickExtended(au8MessageCopy);
        }
#ifdef ANT_VERBOSE
        DebugPrintf("\n\rEVENT_TX\n\r");
#endif
        break;
    }

    case EVENT_TRANSFER_TX_COMPLETED: /* ACK received from acknowledged data */
    {
        G_asAntChannelConfiguration[u8Channel].AntFlags |= _ANT_FLAGS_GOT_ACK;

        AntTickExtended(au8MessageCopy);
#ifdef ANT_VERBOSE
        DebugPrintf("\n\rEVENT_TRANSFER_TX_COMPLETED\n\r");
#endif
        break;
    }

    case EVENT_TRANSFER_TX_FAILED: /* ACK not received from ack data message */
    {
        /* Regardless of complete or fail, it is time to send the next message */
        AntTickExtended(au8MessageCopy);
#ifdef ANT_VERBOSE
        DebugPrintf("\n\rEVENT_TRANSFER_TX_FAILED\n\r");
#endif
        break;
    }

    case EVENT_RX_SEARCH_TIMEOUT: /* Channel will close due to search timeout */
    {
        /* Forward this to application */
        AntTickExtended(au8MessageCopy);
#ifdef ANT_VERBOSE
        DebugPrintf("\n\rEVENT_RX_SEARCH_TIMEOUT\n\r");
#endif
        break;
    }

    case EVENT_CHANNEL_CLOSED: /* The ANT channel is now closed */
    {
        DebugPrintf("Channel closed\n\r");
        G_asAntChannelConfiguration[u8Channel].AntFlags &=
            ~(_ANT_FLAGS_CHANNEL_CLOSE_PENDING | _ANT_FLAGS_CHANNEL_OPEN);
#ifdef ANT_VERBOSE

```



```

        DebugPrintf("\n\rEVENT_CHANNEL_CLOSED\n\r");
    #endif
        break;
    }

    /* All other messages are unexpected for now */
    default:
        DebugPrintNumber(au8MessageCopy[BUFFER_INDEX_RESPONSE_CODE]);
        DebugPrintf(": unexpected channel event\n\r");

        G_u32AntFlags |= _ANT_FLAGS_UNEXPECTED_EVENT;
        break;
    } /* end Ant_pu8AntRxBufferUnreadMsg[EVENT_CODE_INDEX] */
} /* end else RF event */

break;
} /* end case MSG_RESPONSE_EVENT_ID */

```

Notice that almost all Channel Events processed result in a call to `AntTickExtended()` which will be discussed as soon as we finish `AntProcessMessage`. Like the Channel Responses, unhandled Channel Events are flagged in `G_u32AntFlags` (`_ANT_FLAGS_UNEXPECTED_EVENT`) and a debug messages is printed.

`AntProcessMessage` also handles other message IDs including Broadcast and Acknowledged data messages received and some system-level messages that come up with initialization and channel status. All other messages are unhandled and result in `_ANT_FLAGS_UNEXPECTED_MSG`. We will just look at the data message processing here:

```

case MSG_ACKNOWLEDGED_DATA_ID: /* An acknowledged data message was received */
/* Fall through */

case MSG_BROADCAST_DATA_ID: /* A broadcast data message was received */
{
    /* Parse the extended data and put the message to the application buffer */
    AntParseExtendedData(au8MessageCopy, &sExtendedData);
    AntQueueExtendedApplicationMessage(ANT_DATA,
        &au8MessageCopy[BUFFER_INDEX_MSG_DATA], &sExtendedData);

    break;
} /* end case MSG_BROADCAST_DATA_ID */

```

The content of Acknowledged or Broadcast data messages are identical, so they are processed in the same way. If a system needed to differentiate between the two, that would require some additional processing. A task that is interested in the ACK from an Acknowledged message would use the associated Event to determine if the message was received successfully. That event comes as its own message.

Two functions are called:

- `AntParseExtendedData`
- `AntQueueExtendedApplicationMessage`

These functions extract all the useful information from the received message, formats it for `ant_api.c`, and loads it into the message buffer so it is ready to be read by an application using the ANT API.

**AntParseExtendedData()**

```

/*!-----
static bool AntParseExtendedData(u8* pu8SourceMessage_, AntExtendedDataType*
psExtDataTarget_)

Reads extended data based on the flags that are set.

Loads AntExtendedDataType which currently has these fields:
{
    u8 u8Channel;
    u16 u16DeviceID;
    u8 u8DeviceType;
    u8 u8TransType;
    u8 u8Flags;
    s8 s8RSSI;
} AntExtendedDataType;

Requires:
- pu8SourceMessage_ points to an ANT message buffer that holds a complete ANT data
message structure except for SYNC byte; therefore buffer indices from antmessage.h
can be used.
- psExtDataTarget_ points to the target AntExtendedDataType structure

Promises:
- Returns TRUE if extended data is present; all values are read into local
variables and then loaded into psExtDataTarget_.
- Returns FALSE if no extended data is present; psExtDataTarget_ is set to default
*/

```

The ANT protocol processor has access to information about each message that is not normally reported in typical data applications. Any extra data contributes to power consumption both for the ANT processor and the Host so it is off by default. But this data can be very useful and since the EiE system is not devoted to being ultra-low power we enable extended data by default. It should not be turned off since the code assumes it is present. We also assume the format of the extended data does not change from the defaults set which is safe as long as the ANT protocol does not change. It occasionally does, however up to this point, ANT has been very careful to make changes in a way that legacy systems will not be upset.

With extended data on, every Broadcast and Acknowledged data message forwarded to the Host from the ANT processor includes several bytes of information that we can pick out and queue alongside the data payload for the message. We check that the message size falls within expected limits as a weak attempt to confirm we have the extended data expected.

```

/* Get generic data */
u8MessageSize = *(pu8SourceMessage_ + BUFFER_INDEX_MESG_SIZE);
u8Channel = *(pu8SourceMessage_ + BUFFER_INDEX_CHANNEL_NUM);

/* Check to see if the message is the regular size (MESG_MAX_DATA_SIZE) */
if(u8MessageSize == MESG_MAX_DATA_SIZE)
{
    bReturnValue = FALSE;
}

/* Check for a message that is too big or too small */
else if( (u8MessageSize > MESG_MAX_SIZE) ||
        (u8MessageSize < MESG_MAX_DATA_SIZE) )
{
    DebugPrintf("\n\rUnexpected ANT message size\n\n\r");
    bReturnValue = FALSE;
}

```

The extended data has a strict format which we can parse to get the following information:

- **u16DeviceID:** 2-byte Device ID from which the message originated
- **u8DeviceType:** 1-byte Device Type from which the message originated
- **u8TransType:** 1-byte Transmission type from which the message originated
- **s8RSSI:** 1-byte (signed) Signal strength from the device which the message originated
- **u16RxTimeStamp:** 2-byte system time stamp (relative to ANT system time). This is when ANT received the message based on its internal system timing and is therefore typically used to check message arrival time for a larger queue of messages if they are not processed as quickly as they come in.

There are two other parameters that are available in the extended data: a measurement type and a threshold value. We have yet to find a use for these in a typical application so they are not included in the extended data structure and only hooks to parse the data are in the code but commented out.

```

/* Otherwise we have some extended message data */
else
{
    /* The byte after data must be flag byte */
    u8Flags = *(pu8SourceMessage_ + BUFFER_INDEX_EXT_DATA_FLAGS);
    bReturnValue = TRUE;

    /* Channel ID information is always first if it's there */
    if(u8Flags & LIB_CONFIG_CHANNEL_ID_FLAG)
    {
        u16DeviceID =
            (u16)(*(pu8SourceMessage_ + BUFFER_INDEX_EXT_DATA + u8BufferOffset)) & 0x00FF;
        u8BufferOffset++;
        u16DeviceID |=
            ((u16)(*(pu8SourceMessage_ + BUFFER_INDEX_EXT_DATA + u8BufferOffset)) << 8 )
            & 0xFF00;
        u8BufferOffset++;
        u8DeviceType = *(pu8SourceMessage_ + BUFFER_INDEX_EXT_DATA + u8BufferOffset);
        u8BufferOffset++;
        u8TransType = *(pu8SourceMessage_ + BUFFER_INDEX_EXT_DATA + u8BufferOffset);
        u8BufferOffset++;
    }
}

```

```

}

/* RSSI information is always next if it's there */
if(u8Flags & LIB_CONFIG_RSSI_FLAG)
{
    u8BufferOffset++;
    s8RSSI = *(pu8SourceMessage_ + BUFFER_INDEX_EXT_DATA + u8BufferOffset);
    u8BufferOffset++;
    //u8Threshold = *(pu8SourceMessage_ + BUFFER_INDEX_EXT_DATA + u8BufferOffset);
    u8BufferOffset++;
}

/* Timestamp information is always last */
if(u8Flags & LIB_CONFIG_RX_TIMESTAMP_FLAG)
{
    u16RxTimestamp =
        (u16)*(pu8SourceMessage_ + BUFFER_INDEX_EXT_DATA + u8BufferOffset) & 0x00FF;
    u8BufferOffset++;
    u16RxTimestamp |=
        ((u16)*(pu8SourceMessage_ + BUFFER_INDEX_EXT_DATA + u8BufferOffset)) << 8 )
        & 0xFF00;
}
}
}

```

Once all the data has been parsed, we load it back to the calling function through the function parameter pointer:

```

/* Load psExtDataTarget_ (either real or default data) and return */
psExtDataTarget_>u8Flags      = u8Flags;
psExtDataTarget_>u8Channel    = u8Channel;
psExtDataTarget_>u16DeviceID  = u16DeviceID;
psExtDataTarget_>u8DeviceType = u8DeviceType;
psExtDataTarget_>u8TransType  = u8TransType;
psExtDataTarget_>s8RSSI       = s8RSSI;

return bReturnValue;

```

Extended data is obviously useful the moment there are more than two ANT nodes in your system. You can rely on this data to know the source of messages, and with RSSI you can even get a very rough estimate of the proximity of the device. This is particularly useful if you pair based on wildcards or use a scanning channel to monitor traffic.

This takes care of the `ant.c` functions that are involved in ANT to Host communication and working with all the ANT data. Now we look at some design decisions, data structures, and private and public functions that `ant.c` uses to interface to system tasks.

#### 14.6.4 • Task access to send and receive

Any task in the system must be able to read messages from ANT and must also be able to send messages to ANT. This will be done through two linked lists of structs that contain all the relevant information. We chose to create two different buffers as the incoming and outgoing mailboxes for these messages:

1. `G_psAntApplicationMsgList` is a globally available linked list of messages from ANT that `ant.c` processes and queues for user tasks
2. `Ant_psOutgoingMsgList` is a linked list of messages available to `ant.c` from user tasks that will ultimately get sent out to ANT.

In both cases, we need some private functions to setup and move the data accessed by ant.c. We also need some public functions that the user tasks will use to work with the data on their side.

#### 14.6.5 • ANT\_TICK and ANT\_DATA

Before we can discuss the ANT-to-task message handling functions, we need to further describe a design decision that was made to offload a necessary step from any task that uses the ANT API. There are dozens of different ANT messages especially if you factor in the compound messages that can reference event codes or other messages. Handling a subset of those messages is quite sufficient to provide the majority of ANT functionality that a typical system would use. There are still many messages to work with, but as was explained early in the chapter, all of these messages can be grouped into two basic types:

- **ANT\_TICK:** timing, status or event messages
- **ANT\_DATA:** raw message data plus meta data (extended data) about the sender

This is proprietary to the EiE stack and a distinction that we chose to make – it is not explicitly part of the ANT protocol. Having ant.c parse all the messages it queues to the system’s tasks removes the burden of this job from any client task. It is also more efficient because ant.c already knows what kind of message it is processing, so we do not have to have redundant checks in every ANT task.

To keep processing simple, both message types use the same structure to be presented to the application. This is done through a linked list of AntApplicationMsgListType. Though we have looked at that struct type definition before, let’s take another look to refresh your memory:

```
/*
AntApplicationMsgListType
Data struct for the ANT application API message information
*/
typedef struct
{
    u32 u32TimeStamp;                /* Current G_u32SystemTimeIs */
    AntApplicationMessageType eMessageType; /* ANT_TICK or ANT_DATA */
    u8 u8Channel;                    /* Channel which data applies */
    u8 au8MessageData[ANT_APPLICATION_MESSAGE_BYTES]; /* Array for message data */
    AntExtendedDataType sExtendedData; /* Extended message data */
    void *psNextMessage;             /* Pointer to next list item */
} AntApplicationMsgListType;
```

ANT\_DATA messages are straight forward. The 8 bytes of payload that every ANT data message comes with are stored in the au8MessageData member. ANT\_DATA messages will also have Extended data.

ANT\_TICK messages are proprietary and communicate regular information from ANT very much like a system tick from the ANT subsystem, thus the name “ANT\_TICK”. Since we wanted to use the same storage type for ANT\_DATA and ANT\_TICK messages, the information passed in ANT\_TICK or any other message we created had to be defined to fit in the 8 bytes of au8MessageData. Upon initially designing the ANT API, we decided to leave space for 256 different message types with ANT\_TICK being the first since we had no idea what kind of functionality various tasks would need. To date, ANT\_TICK messages have been sufficient to provide everything needed, so that’s the only one we need to explain.

The first byte of an ANT\_TICK message is this message ID which leaves 7 bytes for data payload. These 7 bytes would have specific meaning depending on the message ID if more messages beyond ANT\_TICK were added. The Figure shows the byte description for ANT\_TICK.

MSG_NAME	MSG_ID	D_0	D_1	D_2	D_3	D_4	D_5	D_6
ANT_TICK	0xFF	CHANNEL	RESPONSE	EVENT	0xFF	MISSED	MISSED	MISSED
			TYPE	CODE		MSG #	MSG #	MSG #
						HIGH	MID	LOW

**Figure 14-18 ANT\_TICK byte definition**

The ANT\_TICK message ID is 0xFF, and then the first 3 data bytes are copied from the message that the Host received from ANT. In most cases, ant.c will take the necessary action (if any is required) as a result of receiving a message from ANT, but this information is still forwarded to the task. If nothing else, this is used just for system timing since the majority of ANT\_TICK messages will occur once per message period. It is up to the task to decide if any action is required against the message received.

The middle byte (D\_3) is just a sentinel value 0xFF. The last three bytes combine to form a message counter that tracks the number of missed messages which is automatically incremented inside the ANT driver. This applies only to Slave channels.

Regardless of whether a Slave is communicating to a Master or the other way around, ANT\_TICK messages will be sent at the ANT message period so the Host task can react appropriately. Currently these events will all trigger an ANT\_TICK message:

- RESPONSE\_NO\_ERROR
- EVENT\_RX\_FAIL
- EVENT\_RX\_FAIL\_GO\_TO\_SEARCH
- EVENT\_TX
- EVENT\_TRANSFER\_TX\_COMPLETED
- EVENT\_TRANSFER\_TX\_FAILED
- EVENT\_RX\_SEARCH\_TIMEOUT

We can summarize that using the ANT\_TICK message is important when confirming Host > ANT messages (RESPONSE\_NO\_ERROR), for Slave message status and channel period timing, and to confirm Acknowledged Data messages.

Now it is easy to describe the two functions used to add to this list which ultimately passes ANT information up to the tasks.

#### AntTickExtended()

```
/*!-----/
static void AntTickExtended(u8* pu8AntMessage_)
```

Queues an ANT\_TICK message to the application message queue.

This is expected to run for a Channel Event ANT message though it will work for a standard Channel Response (although the names of the indexes will not be quite right). Currently this function is only called for Channel Events, so if that changes it should be thought about carefully.

Requires:

```
- pu8AntMessage_ points to an ANT message that starts with the LENGTH
  (i.e. no SYNC byte)
```

Promises:

```
- A MESSAGE_ANT_TICK is queued to G_sAntApplicationMsgList
*/
```

AntTickExtended prepares a bunch of data fields in the API mailbox data structure that are relevant to an ANT\_TICK message. An ANT\_TICK message NEVER has extended data, so those fields are always set to 0xFF. The only exception is the Channel number which is always valid since ANT\_TICK messages will almost always be associated with a specific channel.

### AntQueueExtendedApplicationMessage()

```
/*-----*/
static bool AntQueueExtendedApplicationMessage(AntApplicationMessageType
eMessageType_, u8* pu8DataSource_, AntExtendedDataType* psExtData_)

Creates a new ANT message structure and adds it to G_sAntApplicationMsgList. The
Application list is used to communicate message information between the ANT driver
and the ANT_API simplified interface task. It has room for
ANT_APPLICATION_MESSAGE_BUFFER_SIZE messages. The messages are either ANT_DATA or
ANT_TICK message and include all information and data from the original ANT
message.

Requires:
- Enough space is available on the heap
- eMessageType_ specifies the type of message
- pu8DataSource_ is a pointer to the first element of an array of 8 data bytes
- psTargetList_ is a pointer to the list pointer that is being updated

Promises:
- A new list item in the target linked list is created and inserted at the end
  of the list.
- Returns TRUE if the entry is added successfully.
- Returns FALSE if the malloc fails or the list is full.
*/
```

This function is responsible for queuing a new entry in the application buffer. It is simply a function that adds a node to a linked list using dynamic allocation.

```
/* Allocate space for the new message - always do maximum message size */
psNewMessage = malloc( sizeof(AntApplicationMsgListType) );
if (psNewMessage == NULL)
{
    DebugPrintf(Ant_au8AddMessageFailMsg);
    return(FALSE);
}
```

It makes sense to use the Heap since messages from the list do not necessarily have to be removed in order although currently they always are. The number of messages that would be queued are negligible, so we have plenty of space. The size of the messages is also small and always the same size so there should be no fragmentation issues. A more robust system would monitor the Heap usage more carefully and make sure that this list does not end up with any stranded messages that don't get processed out and consume the available Heap over time.

Data is copied in from the provided function parameters:

```
/* Fill in all the fields of the newly allocated message structure */
for(u8 i = 0; i < ANT_APPLICATION_MESSAGE_BYTES; i++)
{
    psNewMessage->au8MessageData[i] = *(pu8DataSource_ + i);
}

/* Fill basic items */
psNewMessage->u32TimeStamp = G_u32SystemTime1ms;
psNewMessage->eMessageType = eMessageType_;

/* Copy all extended data fields */
psNewMessage->sExtendedData.u8Channel = psExtData->u8Channel;
psNewMessage->sExtendedData.u16DeviceID = psExtData->u16DeviceID;
psNewMessage->sExtendedData.u8DeviceType = psExtData->u8DeviceType;
psNewMessage->sExtendedData.u8TransType = psExtData->u8TransType;
psNewMessage->sExtendedData.u8Flags = psExtData->u8Flags;
psNewMessage->sExtendedData.s8RSSI = psExtData->s8RSSI;

psNewMessage->psNextMessage = NULL;
```

The new node is inserted at the end of the list (or is the first node in an empty list). Nothing fancy here, just the typical loop through the list links. Since we cannot be sure how many tasks there are and if the tasks are functioning correctly, the size of this list is limited to ANT\_APPLICATION\_MESSAGE\_BUFFER\_SIZE.

```
/* Insert into an empty list */
if(G_sAntApplicationMsgList == NULL)
{
    G_sAntApplicationMsgList = psNewMessage;
    Ant_u32ApplicationMessageCount++;
}

/* Otherwise traverse the list to find where the new message will be inserted */
else
{
    psListParser = G_sAntApplicationMsgList;
    while(psListParser->psNextMessage != NULL)
    {
        psListParser = psListParser->psNextMessage;
        u8MessageCount++;
    }

    /* Check for full list */
    if(u8MessageCount < ANT_APPLICATION_MESSAGE_BUFFER_SIZE)
    {
        /* Insert the new message at the end of the list */
        psListParser->psNextMessage = psNewMessage;
        Ant_u32ApplicationMessageCount++;
    }
    /* Handle a full list */
    else
    {
        DebugPrintf(Ant_au8AddMessageFailMsg);
        return(FALSE);
    }
}
}
```



The function leaves it up to the calling task to deal with a failed queuing attempt. A very busy system could simply re-try after a few hundred milliseconds. In a quieter system or one that is more deterministic, this should flag that a more serious problem exists in the design. In general, a system will unlikely have more than one or two messages in this queue at any time.

The next function is complementary to `AntQueueExtendedApplicationMessage` and provided Public to user tasks to remove an application message once the task has used it.

### **AntDeQueueApplicationMessage()**

```
/*-----  
void AntDeQueueApplicationMessage(void)  
  
Releases the first message in G_psAntApplicationMsgList  
  
Requires:  
- G_psAntApplicationMsgList points to the start of the list which is the entry to  
  remove  
  
Promises:  
- G_psAntApplicationMsgList = G_psAntApplicationMsgList.  
*/
```

This is possibly a dangerous function to leave to a user task to use as it is equivalent to the C function `free()` because when it's forgotten the system is going to leak memory. However, it must be public because only a task pulls messages off the application message buffer.

```
AntApplicationMsgListType *psMessageToKill;  
  
if(G_psAntApplicationMsgList != NULL)  
{  
    psMessageToKill = G_psAntApplicationMsgList;  
    G_psAntApplicationMsgList = G_psAntApplicationMsgList->psNextMessage;  
  
    /* The doomed message is properly disconnected, so kill it */  
    free(psMessageToKill);  
    Ant_u32ApplicationMessageCount--;  
}
```

Though `G_psAntApplicationMsgList` is set up so that messages can be removed from anywhere in the list, this implementation works as a FIFO. Possible improvements to this system could include some type of garbage collection or time-to-live functionality that would remove messages automatically if they were stale.

That takes care of the functions for pushing messages up to the task through `psAntApplicationMsgList`. Now let's look at the functions for moving ANT messages from the task to `ant.c`.

### **AntCalculateTxChecksum()**

```
/*-----  
u8 AntCalculateTxChecksum(u8* pu8Message_)  
  
Calculates and returns the checksum for a Host to ANT message.  
  
Requires:  
- the message to transmit is a complete ANT message without the SYNC byte
```

```
(it starts with length byte)
- pu8Message_ points to the message to transmit

Promises:
- Returns the ANT checksum for the Tx message
*/
```

The checksum is critical to any ANT message sent from the Host to ANT, so obviously writing a function to calculate it is important. This function is specifically for Tx messages (from Host to ANT). Many alternative ways of implementing this function were considered, including what format the input message should have and how the checksum should be provided back. Returning the checksum directly made the most sense for all the use cases. There is a very strict requirement that the message on which to calculate the checksum is properly formatted and starts at the length byte.

```
/* Adjust for the true size and automatically include MESH_RX_SYNC */
u8 u8Size = *pu8Message_ + 2;
u8 u8Checksum = MESH_RX_SYNC;

/* Find the checksum per the ANT checksum algorithm */
for(u8 i = 0; i < u8Size; i++)
{
    u8Checksum ^= *pu8Message_;
    pu8Message_++;
}

return(u8Checksum);
```

Since we know it is a Host to ANT message, we can seed the MESH\_RX\_SYNC byte to start. We also know that transmit messages are always 2 bytes larger than their length byte.

Even if the user provides a bad pointer or the input message is not correct, the function will execute safely. In the worst-case error, an invalid Length byte will cause this to loop 255 times and deliver an invalid checksum. Garbage in, garbage out, but no risk to system stability.

You could argue that calculating and adding the checksum could be taken out of the user's responsibility and handled by ant.c as it sent the message. However, there is merit to making this the task's job to do as it helps to ensure a level of understanding of the ANT protocol by the user and thus does not obfuscate important details in the event that messages are failing because they have not been constructed correctly.

### AntQueueOutgoingMessage()

```
/*-----
bool AntQueueOutgoingMessage(u8 *pu8Message_)

Creates a new ANT message structure and adds it into Ant_psDataOutgoingMsgList.

If the list is full, the message is not added.
The Outgoing message list is the list of messages sent from the Host to the ANT
chip.

Requires:
- Enough space is available on the heap
- pu8Message_ is an ANT message starting with LENGTH and ending with CHECKSUM

Promises:
```

```
- A new list item in the outgoing message linked list is created.
- Returns TRUE if the entry is added successfully.
- Returns FALSE on error.
*/
```

An outgoing message can be any of the defined ANT messages that the protocol supports including configuration, control, and data messages. Once the message is properly constructed, it is added to the global `Ant_psDataOutgoingMsgList` linked list that will be processed through by the ANT state machine and sent over SPI using the SSP and Message tasks like any other communication transfer in the system.

Outgoing messages are of type `AntOutgoingMessageListType` which are simpler than `AntApplicationMsgListType` message:

```
typedef struct
{
    u32 u32TimeStamp;           /* Current G_u32SystemTime1s */
    u8 au8MessageData[MESG_MAX_SIZE]; /* Array for message data */
    void *psNextMessage;       /* Pointer to next list item*/
} AntOutgoingMessageListType;
```

To use `AntQueueOutgoingMessage`, only a pointer to the message to send is provided but this message must be a properly formatted ANT message including the checksum. No error checking is done on the message, and the Length byte is assumed correct as it is used to parse the data as it's copied to the queue. A `TimeStamp` is added so the age of the message can be monitored in debugging or if a more robust system is required that would ensure that messages are indeed being sent in a timely matter.

```
/* Allocate space for the new message - always do maximum message size */
psNewDataMessage = malloc( sizeof(AntOutgoingMessageListType) );
if (psNewDataMessage == NULL)
{
    DebugPrintf(Ant_au8AddMessageFailMsg);
    return(FALSE);
}

/* Add to the number of queued message */
Ant_DebugQueuedDataMessages++;

/* Fill in all the fields of the newly allocated message structure */
u8Length = *pu8Message_ + 3;
for(u8 i = 0; i < u8Length; i++)
{
    psNewDataMessage->au8MessageData[i] = *(pu8Message_ + i);
}

psNewDataMessage->u32TimeStamp = G_u32SystemTime1ms;
psNewDataMessage->psNextMessage = NULL;
```

The code to insert the message to the list is nearly identical to adding an application message from ANT to the task so we will not show it here. This buffer is exclusively FIFO with a maximum size `ANT_OUTGOING_MESSAGE_BUFFER_SIZE`.

**AntDeQueueOutgoingMessage()**

```

/*!-----*/
static void AntDeQueueOutgoingMessage(void)

Removes the oldest entry of Ant_psDataOutgoingMsgList.

Requires:
- NONE

Promises:
- Ant_psDataOutgoingMsgList = Ant_psDataOutgoingMsgList->psNextMessage
  and the memory is freed

*/

```

This function complements the public function `AntQueueOutgoingMessage`. Once a message is posted to that buffer, the task does not have to worry about it as `ant.c` is the only task that ever removes from it after it knows the SSP module has finished sending.

`AntDeQueueOutgoingMessage` properly removes a message item from `psAntApplicationMsgList` message list.

```

AntOutgoingMessageListType *psMessageToKill;

if(Ant_psDataOutgoingMsgList != NULL)
{
    psMessageToKill = Ant_psDataOutgoingMsgList;
    Ant_psDataOutgoingMsgList = Ant_psDataOutgoingMsgList->psNextMessage;

    /* The doomed message is properly disconnected, so kill it */
    free(psMessageToKill);
}

```

**G\_u32AntFlags**

The final element to discuss before the task state machine is a Global flag register for ANT. Applications can observe ANT status here if the app needs to react to events that take place in ANT that do not have other mechanisms to share the information. This register also contains control flags that various parts of the program reference. The flags are shown below without their numerical bit values.

```

/* G_u32AntFlags */
/* Error / event flags */
#define _ANT_FLAGS_LENGTH_MISMATCH /* Set if counted Rx bytes != Length byte */
#define _ANT_FLAGS_CMD_ERROR /* A command received an error response */
#define _ANT_FLAGS_UNEXPECTED_EVENT /* Message parser saw an unexpected event */
#define _ANT_FLAGS_UNEXPECTED_MSG /* Message parser saw an unexpected message */

#define ANT_ERROR_FLAGS_MASK /* Mask out all error flags */
#define ANT_ERROR_FLAGS_COUNT /* Current number of error flags */

/* Status flags */
#define _ANT_FLAGS_ /* An ANT restart message was received */

/* Control flags */
#define _ANT_FLAGS_RX_IN_PROGRESS /* Set when an ANT frame reception starts */
#define _ANT_FLAGS_TX_IN_PROGRESS /* Set when an ANT frame transmission starts */
#define _ANT_FLAGS_TX_INTERRUPTED /* An attempt to transmit was interrupted */

```

```
/* end G_u32AntFlags */
```

It is assumed that this register is read-only to other applications even though nothing exists to enforce that. If this were a more public API, this status would be made through a function call to access a private member variable inside the ANT app. At the time of writing, only ant.c accesses this register.

#### 14.7 • ANT State Machine

The private functions in the ANT application end up doing most of the work in the ANT system. Only three public functions are needed to write an interface to ant.c and access the message lists. These are still somewhat low-level compared to what you might want a user task to do. If you work at this level, you need to have a solid understanding of how ant.c functions and you need to know a little bit about the ANT protocol.

The ANT application ends up being quite simple and makes sure that messages are being sent, received and processed. It also has to get ANT up and running to the point where it can be used in the system so initialization is very important. Once the ANT SM is initialized and operating, any application running will be able to use the ANT radio. The figure below shows the state diagram for ANT SM.

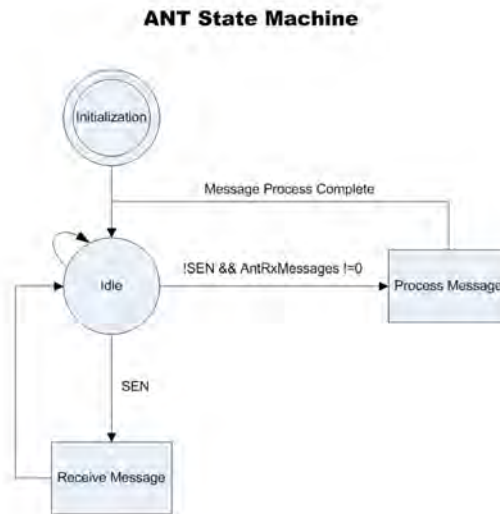


Figure 14-19 State diagram for ANT SM

##### 14.7.1 • Initializing the ANT SM

A call to AntInitialize() will start the ANT state machine and should be made outside of the main program loop like with any other application. The function call must occur after all of the lower level driver functions have been initialized since Ant requires MessageSender, SSP, and Debug functionality. This initialization function follows the system rules and reports its status with messages on the debug UART port so that developers can quickly see the status. AntInitialize will properly reset the synchronous serial interface to ANT and test communications. If successful, it will send the base configuration commands to set up the non-changing channel parameters of the system.

```

/*!-----
void AntInitialize(void)

Initialize the ANT system.

The ANT device is reset and communication checked through a version request.
The main channel parameters are then set up to default values.

The user can override this function by holding BUTTON3
during startup.

Requires:
- ANT_SPI peripheral is correctly configured
- Debug system is initialized so debug messages can be sent to UART

Promises:

If successful:
- G_stAntSetupData set to default ANT values
- G_u32ApplicationFlags _APPLICATION_FLAGS_ANT bit is set
- Ant_pfnStateMachine = AntSM_Idle

If failed or override:
- G_u32SystemFlags _SYSTEM_STARTUP_NO_ANT is set;
- G_u32ApplicationFlags _APPLICATION_FLAGS_ANT bit remains clear
- Ant_pfnStateMachine = AntSM_NoResponse
*/

```

The function starts by checking a disable feature that can be used to turn off ANT functionality in the SAM3U2. One of the things this does is keep the ANT RESET line in high-impedance mode so it can be managed by the J-Link and not driven by the SAM3U2. This is very helpful if the nRF51422 needs to be flashed or is being debugged directly through the J-Link. If ANT is to boot normally, the SAM3U2 is given control of the ANT RESET line and initialization begins.

```

void AntInitialize(void)
{
    /* Check for manual disabling of ANT */
    if( ANT_BOOT_DISABLE() )
    {
        G_u32SystemFlags |= _SYSTEM_STARTUP_NO_ANT;
        DebugPrintf(G_au8AntMessageNoAnt);

        /* Float all of the ANT interface lines so that the J-Link programmer
        or other firmware will not be impacted by the Host MCU */
        AT91C_BASE_PIOA->PIO_PER = ANT_PIOA_PINS;
        AT91C_BASE_PIOB->PIO_PER = ANT_PIOB_PINS;

        /* Disable all outputs (set to HiZ input) */
        AT91C_BASE_PIOA->PIO_ODR = ANT_PIOA_PINS;
        AT91C_BASE_PIOB->PIO_ODR = ANT_PIOB_PINS;

        Ant_pfnStateMachine = AntSM_NoResponse;
    } /* Otherwise try to start up ANT normally */
    else
    {
        /* Give PIO control of ANT_RESET line */
        AT91C_BASE_PIOB->PIO_OER = PB_21_ANT_RESET;

        /* Announce on the debug port that ANT setup is starting and initialize pointers */
        DebugPrintf(G_au8AntMessageInit);
    }
}

```

```
G_psAntApplicationMsgList = 0;
Ant_psOutgoingMsgList = 0;
```

The first data structure to get initialized is `G_asAntChannelConfiguration` which holds channel information for all available ANT channels. We know `AntChannelNumberType` is sequential so we can type cast the loop variable 'i' to write that member in the loop.

```
/* Initialize the G_asAntChannelConfiguration data struct */
for(u8 i = 0; i < ANT_NUM_CHANNELS; i++)
{
    G_asAntChannelConfiguration[i].AntChannel          = (AntChannelNumberType)i;
    G_asAntChannelConfiguration[i].AntChannelType      = ANT_CHANNEL_TYPE_DEFAULT;
    G_asAntChannelConfiguration[i].AntNetwork          = ANT_NETWORK_DEFAULT;
    G_asAntChannelConfiguration[i].AntDeviceIdLo       = ANT_DEVICE_ID_LO_DEFAULT;
    G_asAntChannelConfiguration[i].AntDeviceIdHi      = ANT_DEVICE_ID_HI_DEFAULT;
    G_asAntChannelConfiguration[i].AntDeviceType       = ANT_DEVICE_TYPE_DEFAULT;
    G_asAntChannelConfiguration[i].AntTransmissionType = ANT_TRANSMISSION_TYPE_DEFAULT;
    G_asAntChannelConfiguration[i].AntChannelPeriodLo  = ANT_CHANNEL_PERIOD_LO_DEFAULT;
    G_asAntChannelConfiguration[i].AntChannelPeriodHi  = ANT_CHANNEL_PERIOD_HI_DEFAULT;
    G_asAntChannelConfiguration[i].AntFrequency        = ANT_FREQUENCY_DEFAULT;
    G_asAntChannelConfiguration[i].AntTxPower          = ANT_TX_POWER_DEFAULT;
    G_asAntChannelConfiguration[i].AntFlags            = 0;

    for(u8 j = 0; j < ANT_NETWORK_NUMBER_BYTES; j++)
    {
        G_asAntChannelConfiguration[i].AntNetworkKey[j] = 0;
    }
}
```

Next the SSP peripheral is setup and requested.

```
/* Configure the SSP resource to be used for the application */
Ant_sSspConfig.SspPeripheral      = ANT_SPI;
Ant_sSspConfig.pCsGpioAddress    = ANT_SPI_CS_GPIO;
Ant_sSspConfig.u32CsPin         = ANT_SPI_CS_PIN;
Ant_sSspConfig.eBitOrder        = LSB_FIRST;
Ant_sSspConfig.eSspMode         = SPI_Slave_FLOW_CONTROL;
Ant_sSspConfig.fnSlaveTxFlowCallback = AntTxFlowControlCallback;
Ant_sSspConfig.fnSlaveRxFlowCallback = AntRxFlowControlCallback;
Ant_sSspConfig.pu8RxBufferAddress = Ant_au8AntRxBuffer;
Ant_sSspConfig.ppu8RxNextByte    = &Ant_pu8AntRxBufferNextChar;
Ant_sSspConfig.u16RxBufferSize  = ANT_RX_BUFFER_SIZE;

Ant_Ssp = SspRequest(&Ant_sSspConfig);
ANT_SSP_FLAGS = 0;
```

Lastly, `AntSyncSerialInitialize` is called to run the initialization sequence.

If `_APPLICATION_FLAGS_ANT` is set as a result, ANT initialization is successful. If not, the ANT system is simply pointed to a failed state. In both cases, the debug output alerts the user to the state of the ANT task.

```

/* Reset ANT, activate SPI interface and get a test message */
AntSyncSerialInitialize();

/* Report status out the debug port */
if(G_u32ApplicationFlags & _APPLICATION_FLAGS_ANT)
{
    DebugPrintf(G_au8AntMessageOk);
    DebugPrintf("ANT version: ");
    DebugPrintf(Ant_u8AntVersion);
    DebugLineFeed();

    G_u32AntFlags &= ~_ANT_FLAGS_RESTART;
    Ant_pfnStateMachine = AntSM_Idle;
}
else
{
    /* The ANT device is not responding -- it may be dead, or it may not yet
    be loaded with any firmware */
    DebugPrintf(G_au8AntMessageInitFail);

    Ant_pfnStateMachine = AntSM_NoResponse;
}
}

```

#### 14.8 • Implementing the ANT State Machine

Once `AntInitialize` has completed successfully, the state machine pointer is set to the Idle state and ready to run in the main loop. The ANT state machine has three active states:

##### **AntSM\_Idle()**

```

/*!-----
static void AntSM_Idle(void)

Idle state that will process new messages if any are present, monitors ANT
for incoming messages and sends broadcast messages that are waiting to be sent.
Incoming messages from ANT always get priority.
*/
static void AntSM_Idle(void)
{
    u32 u32MsgBitMask = 0x01;
    u8 u8MsgIndex = 0;
    static u8 au8AntFlagAlert[] = "ANT flags:\n\r";

    /* Error messages: must match order of G_u32AntFlags Error / event flags */
    static u8 au8AntFlagMessages[][20] =
    { /* "012345678901234567\n\r" */
        "Length mismatch\n\r",
        "Command error\n\r",
        "Unexpected event\n\r",
        "Unexpected message\n\r"
    };
};

```

The first thing the Idle state does is check the error flags in `G_u32AntFlags`. Currently the system just outputs a debug message and then clears these flags since the `ant.c` task already responds to these flags.



```

/* Check flags */
if(G_u32AntFlags & ANT_ERROR_FLAGS_MASK)
{
    /* At least one flag is set, so print header and parse out */
    DebugPrintf(au8AntFlagAlert);
    for(u8 i = 0; i < ANT_ERROR_FLAGS_COUNT; i++)
    {
        /* Check if current flag is set */
        if(G_u32AntFlags & u32MsgBitMask)
        {
            /* Print the error message */
            DebugPrintf(au8AntFlagMessages[u8MsgIndex]);
        }
        u32MsgBitMask <= 1;
        u8MsgIndex++;
    }

    /* Clear all the error flags now that they have been reported */
    G_u32AntFlags &= ~ANT_ERROR_FLAGS_MASK;
}

```

Next, any messages that might be waiting from ANT (that would have been received via interrupt) are processed.

```

/* Process messages received from ANT */
AntProcessMessage();

```

If the ANT processor wants to send a message to the SAM3U2, the SEN line will be asserted so this is checked next and the top priority in the task.

```

/* Handle messages coming in from ANT */
if( IS_SEN_ASSERTED() )
{
    Ant_pfnStateMachine = AntSM_ReceiveMessage;
}

```

Otherwise, the code checks to see if any messages have been queued to Ant\_psOutgoingMsgList. As long as no message is currently in progress and there is at least one message queued, the transmit sequence will be started.

```

/* Send a message if the system is ready and there is one to send */
else if( (Ant_u32CurrentTxMessageToken == 0 ) &&
        (Ant_psOutgoingMsgList != NULL) )
{
    /* Give the message to AntTx which will set Ant_u32CurrentTxMessageToken */
    if(AntTxMessage(Ant_psOutgoingMsgList->au8MessageData))
    {
        Ant_u32TxTimer = G_u32SystemTime1ms;
        Ant_pfnStateMachine = AntSM_TransmitMessage;
    }
    else
    {
        /* Transmit attempt failed. !!!! Do something? */
        DebugPrintf("\n\rANT transmit failed\n\r");
    }
}

```

**AntSM\_ReceiveMessage()**

```

/*!-----
static void AntSM_ReceiveMessage(void)

Completely receive an ANT message. Reception is very fast and should complete
in less than 600us for a 15-byte message. AntRxMessage could just be called from
Idle but giving it its own state minimizes the total time and allows for easier fu-
ture updates should they be required.
*/
static void AntSM_ReceiveMessage(void)
{
    Ant_DebugRxMessageCounter++;
    AntRxMessage();

    Ant_pfnStateMachine = AntSM_Idle;
} /* end AntSM_ReceiveMessage() */

```

This state blocks while it completely receives an ANT message. Reception is very fast and should complete in less than 600us even for a 15-byte message and therefore not violate system timing. At 10Hz messaging, the message period is 100ms so polling at 1ms allows plenty of margin to ensure no messages are missed and it gives plenty of time to process each message and react accordingly.

The state is separate from Idle to give the maximum time possible just to receive a message within the 1ms system loop design. This also allows ANTSM\_Idle flexibility to evolve without having to consider the possibility of a relatively long blocking function sharing time with Idle.

As we approach 1kHz messaging or Burst transfers we could begin to have problems and would have to rethink how messages in this system are processed. No doubt this would be a fundamental architecture change that is simply not intended to be supported by the EiE system.

**AntSM\_TransmitMessage()**

```

/*!-----
static void AntSM_TransmitMessage(void)

Wait for an ANT message to be transmitted. This state only occurs once the
handshaking transaction has been completed and transmit to ANT is verified and
underway.
*/
static void AntSM_TransmitMessage(void)
{

```

ANT messages are tracked in the system via a message token, Ant\_u32CurrentTxMessageToken. This is the message number assigned when the ANT message is queued to the SSP module. The system uses QueryMessageStatus with the ANT message token to wait for the message to TIMEOUT or COMPLETE and then takes appropriate action.

```

eCurrentMsgStatus = QueryMessageStatus (Ant_u32CurrentTxMessageToken);
switch(eCurrentMsgStatus)
{
    case TIMEOUT:
        DebugPrintf("\n\rTransmit message timeout\n\r");
        /* Fall through */

```

```

case COMPLETE:
    /* Kill the message and update flags */
    AntDeQueueOutgoingMessage();
    Ant_u32CurrentTxMessageToken = 0;
    G_u32AntFlags &= ~_ANT_FLAGS_TX_IN_PROGRESS;

    /* Wait for SEN to deassert so we know ANT is totally ready for the next
    transaction. Should be max 170us */
    while ( IS_SEN_ASSERTED() && (Ant_u32TxTimer < ANT_ACTIVITY_TIME_COUNT) )
    {
        Ant_u32TxTimer++;
    }

```

A message that times out is likely due to the ANT processor being somehow stuck which is typically a result of sending a message that is improperly encoded. As this is one of those “should never happen because you shouldn’t be sending bad messages” scenarios, the stack does not take any specific action but does try to clock SRDY until SEN deasserts so that ANT is alive.

```

/* If we timed out, then ANT is likely stuck */
if(Ant_u32RxTimer > ANT_ACTIVITY_TIME_COUNT)
{
    /* Try to unstick ANT !!!! Should have a timeout for this */
    while( IS_SEN_ASSERTED() )
    {
        AntSrdyPulse();
    }
}
Ant_pfnStateMachine = AntSM_Idle;
break;

default:
    /* Do nothing for now */
    break;
} /* end switch */

```

A fully robust system would alert the task who sent the message, and either repair ANT automatically (going as far as resetting ANT or the whole system) or expect the task to try to repair it.

## 14.9 • API Summary

An API could be written to handle every possible message that ANT could send or provide even higher-level messaging functions so the application can make bulk data transfers without having to form 8-byte messages. There are many capabilities that ANT itself offers such as burst mode transfers or the entire ANT FS (file system) functionality to address some of these needs.

The point is to emphasize that there are infinite ways to write a driver and construct an API for a system like ANT. With no hard and fast rules, it is up to the software designer to create a system that correctly balances the capabilities with the needs of the end user application while optimizing the code for the available processor resources including code space, RAM, and execution speed. This will also translate into net power consumption of the application that uses the stack. Creating a great API is a challenging task and will very often take a few versions to polish it up.

ANT is interesting to work with because the protocol is still fairly low-level. A user needs to understand the message structure and system timing, and even reference the command ID table to build messages correctly. Deciding what an API should abstract requires a good understanding of the use cases for the API.

Consider the act of sending data. At the highest level of abstraction, the API could provide a “SendData()” function that could take a pointer and size of the data to send. This would be the simplest interface from the client’s perspective. The API implementation would have to format the data into 8-byte groups for transmission and run a series of Acknowledged message transfers to send all the data and ensure it got through. If the system had connectivity to another device, then likely the data would eventually be transferred successfully. This could be reported back to the application in a similar way as how the Message task tracks token status.

What if a connection to the other device was not present? Due to the nature of the ANT protocol, would the user need to know if they were the Master or Slave? How would power be optimized? If the client application was a Slave, what should happen with the continual Broadcast synchronization messages? The questions are endless, and in that situation, it makes more sense to avoid trying to predict everything a client might want and instead offload a lot of specific details to the client assuming they have a general working knowledge of ANT.

The EiE API aims to provide a simple interface to basic ANT functionality while relying on a reasonable level of ANT knowledge by the user. It is coded in `ant_api.c` and `ant_api.h`. This API has grown a few times and there are still some functions that would be helpful to add. Though this chapter has been extremely heavy on code, there is still no better way to understand how to code up a system than to study this example with our explanation.

#### 14.9.1 • ANT Configuration and Status Message

The ANT API is divided into sections that roughly correspond to the two “modes” of operation that we described for ANT. Configuration and status messages are used almost exclusively on channels that are not operating in the system.

##### **AntAssignChannel()**

Before you can use an ANT channel, it must be assigned to set up all of the ANT channel requirements. It is up to the user to know what they want – this function just updates the messages needed to fully define a channel’s parameters. This is entirely a data storage operation though the function still returns a Boolean type to indicate success. The code starts by verifying that the channel being attempted for configuration is available. This is the only way that the function will return FALSE.

```
bool AntAssignChannel(AntAssignChannelInfoType* psAntSetupInfo_)
{
    /* Check to ensure the selected channel is available */
    if(AntRadioStatusChannel(psAntSetupInfo_>AntChannel) != ANT_UNCONFIGURED)
    {
        DebugPrintf("AntAssignChannel error: channel is not unconfigured\n\r");
        return FALSE;
    }
}
```

All of the message structs being used are globals so they are available to other functions if needed as there is a lot of data required to store these. The values that need to be updated from the ChannelInfo are loaded directly starting with the Library configuration message and the Network key. After the values in each message are updated, `G_asAntChannelConfiguration` and the message’s checksum must also be updated.

```
/* Setup the library config message (for extended data) - use defaults for now */
G_au8AntLibConfig[4] = AntCalculateTxChecksum(G_au8AntLibConfig);

/* Set Network key message */
G_au8AntSetNetworkKey[2] = psAntSetupInfo->AntNetwork;
for(u8 i = 0; i < ANT_NETWORK_NUMBER_BYTES; i++)
{
    G_au8AntSetNetworkKey[i + 3] = psAntSetupInfo->AntNetworkKey[i];
    G_asAntChannelConfiguration[psAntSetupInfo->AntChannel].AntNetworkKey[i] =
        psAntSetupInfo->AntNetworkKey[i];
}
G_au8AntSetNetworkKey[11] = AntCalculateTxChecksum(G_au8AntSetNetworkKey);
```

The channel ID, channel period, channel radio frequency, power and search timeout are all configured in the same way. There is no benefit of showing that code. Once all of the messages are updated, the function exits by switching the state to AssignChannel so that the messages can queue.

### **AntUnassignChannelNumber()**

This simple function attempts to unassign the channel specified. Channels must be in ANT\_CLOSED state for this to work. The channel state is checked, the UnassignChannel message is updated, and the message is queued. If everything works the function returns TRUE.

```
bool AntUnassignChannelNumber(AntChannelNumberType eChannel_)
{
    u8 au8AntUnassignChannel[] = {MSG_UNASSIGN_CHANNEL_SIZE,
        MSG_UNASSIGN_CHANNEL_ID, 0, CS};

    /* Check if the channel is closed */
    if(AntRadioStatusChannel(eChannel_) != ANT_CLOSED)
    {
        DebugPrintf("AntUnssignChannel error: channel not closed\n\r");
        return FALSE;
    }

    /* Update the channel number */
    au8AntUnassignChannel[2] = eChannel_;

    /* Update checksum and queue the unassign channel message */
    au8AntUnassignChannel[3] = AntCalculateTxChecksum(au8AntUnassignChannel);
    return( AntQueueOutgoingMessage(au8AntUnassignChannel) );
} /* end AntUnassignChannelNumber() */
```

### **AntOpenChannelNumber() / AntOpenScanningChannel() / AntCloseChannelNumber()**

A configured and assigned channel can be opened. If it's a scanning channel then channel 0 must be used so no channel parameter is passed. An open channel can be closed. These three functions work almost identically so we'll just detail one. The required channel state is checked and if ok the corresponding command message is updated and queued.

```
bool AntOpenChannelNumber(AntChannelNumberType eChannel_)
{
    u8 au8AntOpenChannel[] = {MSG_OPEN_CHANNEL_SIZE, MSG_OPEN_CHANNEL_ID, 0, CS};

    /* Check if the channel is ready */
```

```

if(AntRadioStatusChannel(eChannel_) != ANT_CONFIGURED)
{
    DebugPrintf("AntOpenChannel error: channel not ready\n\r");
    return FALSE;
}

/* Update the channel number in the message for a regular channel */
if(eChannel_ != ANT_CHANNEL_SCANNING)
{
    au8AntOpenChannel[2] = eChannel_;
}

/* Update the checksum value and queue the open channel message */
au8AntOpenChannel[3] = AntCalculateTxChecksum(au8AntOpenChannel);
G_asAntChannelConfiguration[eChannel_].AntFlags |= _ANT_FLAGS_CHANNEL_OPEN_PENDING;

return( AntQueueOutgoingMessage(au8AntOpenChannel) );
} /* end AntOpenChannelNumber() */

```

### AntRadioStatusChannel()

The current status of a channel is important for a task to know for many reasons. The ANT API passes clear status information about any channel to any task that inquires. This should be used by a task before requesting a channel, and also to control its own state machine based on the current status of the channel. A default ANT system is 250 times slower than the 1ms system loop for the EiE system, so there will be a lot of waiting while messages are queued up, delivered, and responded to by the ANT processor.

This function just translates the status flags that ant.c keeps track of based on the messages that the ANT processor sends. By parsing these flags, a simple easy-to-understand status can be returned to the task.

```

AntChannelStatusType AntRadioStatusChannel(AntChannelNumberType eChannel_)
{
    if(G_asAntChannelConfiguration[eChannel_].AntFlags & _ANT_FLAGS_CHANNEL_CONFIGURED)
    {
        if(G_asAntChannelConfiguration[eChannel_].AntFlags &
            _ANT_FLAGS_CHANNEL_CLOSE_PENDING)
        {
            return ANT_CLOSING;
        }
        else if(G_asAntChannelConfiguration[eChannel_].AntFlags & _ANT_FLAGS_CHANNEL_OPEN)
        {
            return ANT_OPEN;
        }
        else
        {
            return ANT_CLOSED;
        }
    }
    else
    {
        return ANT_UNCONFIGURED;
    }
} /* end AntRadioStatusChannel () */

```

### 14.9.2 • ANT Data Messages

The second group of functions is all about data. There are four data-related messages that offer the fundamentally useful functionality of an ANT wireless connection: moving data. The first two functions in the group provide the mechanism to queue data to be sent. The second pair of functions allows a task to read data received over the air.

#### **AntQueueBroadcastMessage() / AntQueueAcknowledgedMessage()**

The most basic ANT data message is the Broadcast message. This is the message that will be sent continually by ANT Master devices. For Slaves, this message will be sent in reply to a message received by a Master as long as it has been queued prior to the Master's message arriving.

An Acknowledged data message is queued identically, but of course, will have different results through the ANT system. Only the Broadcast data code is shown.

The API functions simply load the data provided by the user with the assumption that the supplied pointer references an 8-byte data array. This is where the correct usage and understanding of ANT is important. If the user does not understand ANT, doesn't read the function definition, and wants to send more or less than 8 bytes, their pointer may reference garbage data. Adding garbage data to the message will not hurt anything – it just won't make sense. One could argue there is a potential security issue there due to an unbounded pointer. If that is a concern, this function would have to be written differently to be more secure.

```
bool AntQueueBroadcastMessage(AntChannelNumberType eChannel_, u8 *pu8Data_)
{
    /* Update the dynamic message data */
    G_au8AntBroadcastDataMessage[2] = eChannel_;
    for(u8 i = 0; i < ANT_DATA_BYTES; i++)
    {
        G_au8AntBroadcastDataMessage[3 + i] = *(pu8Data_ + i);
    }

    G_au8AntBroadcastDataMessage[11] =
        AntCalculateTxChecksum(G_au8AntBroadcastDataMessage);

    return( AntQueueOutgoingMessage(G_au8AntBroadcastDataMessage) );
} /* end AntQueueBroadcastMessage */
```

Remember that if an Acknowledged data message is queued to a Master, the Master will send the Acknowledged message but then revert to sending Broadcast messages if a new Acknowledged message is not queued before the next message period. Slaves will respond with whatever type of message is queued. Neither a Master or Slave can send both an Acknowledged message and Broadcast message at the same time. However, both can send the simple ACK to a received Acknowledged message in the same period that a new data message is sent. User tasks must look for an ANT\_TICK that confirms the message was sent. The arrival of this event means that the user task should queue the next data message so it is ready for the next message period.

#### **AntReadAppMessageBuffer()**

The most important and – in our experience – most difficult function to understand is AntReadAppMessageBuffer(). There are infinite ways that data received over the ANT radio could be provided to a task. There are just as many tradeoffs to consider. Our choice to group these as ANT\_DATA and ANT\_TICK has been explained and hopefully you see the value in pre-sorting messages.

Remember, also, that these messages are queued in a buffer that `ant.c` manages. What we wanted to do with the API is provide a simple mechanism to get the oldest unread message from ANT back to the user task. The task does not know if the message is an `ANT_DATA` or `ANT_TICK` message, so that's one problem to solve. The design addresses this by using identical structures to capture information from either message type.

There is also the danger that a task could read a message that does not belong to it. For example, `Task1` could be sending and receiving on Channel 0, and `Task2` could be using Channel 3. Since tasks don't know where a message came from until they read it, there exists a possibility of "stealing" a message from another task. This, of course, depends on how exactly the messages are stored and processed once they are requested. As we said, there is a myriad of decisions to be made!

We chose to design what we thought was a happy medium that didn't require a huge amount of firmware to manage. The ANT API provides `AntReadAppMessageBuffer()` that causes the information from the oldest message in the `ant.c` message list to be loaded into four global variables. Once the data is loaded, the message is dequeued from the message list.

The four globals are:

- **G\_u32AntApiCurrentMessageTimeStamp:** the system time when the message that was just loaded actually arrived from the ANT processor. Since the Ant message buffer is FIFO, this value should never be lower than a previous value.
- **G\_eAntApiCurrentMessageClass:** `ANT_TICK` or `ANT_DATA`. This is the first variable that a task should read after calling `AntReadAppMessageBuffer`. This is the first decision point that a task should make in deciding how to process the message.
- **G\_a8AntApiCurrentMessageBytes:** the 8 bytes of data corresponding to the Message Class. If it is an `ANT_DATA` message, then these 8 bytes are the 8 data bytes received over the air from whatever device is connected. The meaning of the bytes is entirely up to the system that is set up. It could be a sensor reading, a text message, or any other proprietary information. If the message is an `ANT_TICK`, then the bytes should be parsed according to the definition in `ant_api.h`.
- **G\_sAntApiCurrentMessageExtData:** This is the extra data that may come with a message. Both `ANT_DATA` and `ANT_TICK` messages can read the channel number from this array. All other bytes are not defined for `ANT_TICK` messages. `ANT_DATA` messages will include all of the message sender's information including its Device ID, Device Type, Transmission Type, and RSSI.

It is essential to understand how this function works and what these variables hold after they are updated. Looking at the implementation of `AntReadAppMessageBuffer()` should help.

The function does nothing if there are no new messages, but if there is a message, the timestamp, message type and data are copied first.

```
bool AntReadAppMessageBuffer(void)
{
    u8 *pu8Parser;

    if(G_psAntApplicationMsgList != NULL)
    {
        /* Grab the single bytes */
        G_u32AntApiCurrentMessageTimeStamp = G_psAntApplicationMsgList->u32TimeStamp;
```



```

G_eAntApiCurrentMessageClass = G_psAntApplicationMsgList->eMessageType;

/* Copy over all the payload data */
pu8Parser = &(G_psAntApplicationMsgList->au8MessageData[0]);
for(u8 i = 0; i < ANT_APPLICATION_MESSAGE_BYTES; i++)
{
    G_au8AntApiCurrentMessageBytes[i] = *(pu8Parser + i);
}

```

Next, the extended data is copied. The Ant application has already formatted this appropriately for ANT\_TICK and ANT\_DATA.

```

/* Copy over the extended data */
G_sAntApiCurrentMessageExtData.u8Channel =
    G_psAntApplicationMsgList->sExtendedData.u8Channel;
G_sAntApiCurrentMessageExtData.u8Flags =
    G_psAntApplicationMsgList->sExtendedData.u8Flags;
G_sAntApiCurrentMessageExtData.u16DeviceID =
    G_psAntApplicationMsgList->sExtendedData.u16DeviceID;
G_sAntApiCurrentMessageExtData.u8DeviceType =
    G_psAntApplicationMsgList->sExtendedData.u8DeviceType;
G_sAntApiCurrentMessageExtData.u8TransType =
    G_psAntApplicationMsgList->sExtendedData.u8TransType;
G_sAntApiCurrentMessageExtData.s8RSSI =
    G_psAntApplicationMsgList->sExtendedData.s8RSSI;

```

Now all of the data has been captured in the four globals so the Ant application can remove the message from its list and the function returns.

```

/* Done, so message can be removed from the buffer */
AntDeQueueApplicationMessage();
return TRUE;
}

/* Otherwise return FALSE and do not touch the current data array */
return FALSE;

} /* end AntReadAppMessageBuffer() */

```



A great exercise would be to draw the whole system from the ANT processor all the way to the user task. Identify what data exists at each point in the system and what data structures store this data. Show how the API accesses the ant.c information.

We can acknowledge that a system that has multiple tasks using ANT would have to be designed to check the current globals to see if the message belongs to it. The system would have to allow one complete cycle through the main loop so that all tasks could get a look at the message before the next one was pulled out of the ant.c buffer. Admittedly we have not written an application that needs to do this, but the approach seems like it would work. Perhaps we would add a supervisor task to help manage messages.

### AntGetdBmAscii()

The final data function is just a tool to read out what we think is the most useful and entertaining piece of information that the nRF51422 ANT radio can provide: signal strength. Every message received by the ANT radio hardware is tagged with the signal strength observed at that time. This information is appended to the message data and available in the Extended message bytes.

The value is signed and can be used directly in firmware, but there are many applications

where displaying it in ASCII is useful. So this API function was designed to provide an easy conversion between the extended data value and an ASCII string.

The maximum transmit power in an ANT system is currently 4dBm, so it is impossible to receive an RSSI value that is greater than 0. The highest we have seen is around -40dBm. However, the function starts out by considering a positive value as it starts to build the ASCII string. If the value is negative, then the absolute value (two's complement) is calculated.

```
void AntGetdBmAscii(s8 s8RssiValue_, u8* pu8Result_)
{
    u8 u8AbsoluteValue;

    /* Handle the positive number */
    if(s8RssiValue_ >= 0)
    {
        /* Print '+' but only for numbers larger than 0 */
        *pu8Result_ = '+';
        if(s8RssiValue_ == 0)
        {
            *pu8Result_ = ' ';
        }

        u8AbsoluteValue = (u8)s8RssiValue_;
    }
    /* Handle the negative number */
    else
    {
        *pu8Result_ = '-';
        u8AbsoluteValue = (u8)(~s8RssiValue_ + 1);
    }
}
```

Receiver sensitivity of the ANT radio is down around -100dBm, but values that low are not really useful. To simplify this function, it was decided to limit to two digits on display, so -99dBm would be the lowest value returned. Once this is found, the digits are parsed out, converted to ASCII, and the final string is loaded.

```
/* Limit any display to two digits */
if(u8AbsoluteValue > 99)
{
    u8AbsoluteValue = 99;
}

/* Write the numeric value */
pu8Result_++;
*pu8Result_ = (u8AbsoluteValue / 10) + NUMBER_ASCII_TO_DEC;
pu8Result_++;
*pu8Result_ = (u8AbsoluteValue % 10) + NUMBER_ASCII_TO_DEC;
} /* end AntGetdBmAscii() */
```

#### 14.10 • Chapter Exercise

It is our hope that this API makes sense enough that it can be used to experiment with ANT. As with learning anything new, trying things out and solving problems that come up is the best way to ingrain a new concept in your mind.

The EiE webpage offers three modules on ANT that review the most important concepts and takes you through using ANTWare to make sure your development tools are working

properly. It then walks you through a simple Master-based module followed by a more complicated Slave-based module.

There are many ANT-based projects online for the EiE system. There are endless games that can be made to work over ANT and some very interesting and practical applications. Remember that ANT is a global standard, which means you can interface your development board to any ANT+ device on the market. Learning ANT+ is trivial compared to learning the base ANT system. ANT+ is basically just ensuring specific data formats for compatible devices.

ANT is also native in most Android installations now and there are more and more examples of applications where it can be used.

#### **14.11 • Conclusion**

This book has been a massive amount of information about bare-metal embedded systems. Hopefully, it has helped you build a strong foundation of knowledge. The simple fact is that we rely on this knowledge every day when we are designing and writing firmware for products in industry. While you can get by for a while using other people's code, APIs, and examples, without the knowledge you have gained here you are potentially building a house of cards on quicksand.

The EiE program will continue to expand, and the online resources will keep growing. Keep an eye on the [eie GitHub](#) account for firmware updates, new features and applications, and inevitable bug fixes.

All the best to you.

## Glossary

<b>AHB</b>	Advanced High speed Bus
<b>APB</b>	Advanced Peripheral Bus
<b>ASCII</b>	American Standard for Information Interchange
<b>Binary</b>	A base-2 number system
<b>Bit</b>	The smallest unit that can be represented in a digital system. A bit is either 1 or 0.
<b>Bit-Bash</b>	A manual, brute force implementation of an algorithm, protocol, or process that could likely be done more simply using a hardware peripheral (a.k.a. Bit Bang).
<b>BJT</b>	Bipolar Junction Transistor
<b>BLE</b>	Bluetooth Low Energy
<b>Byte</b>	A group of 8 bits
<b>CDMA</b>	Code Division Multiple Access
<b>CMOS</b>	Complementary Metal Oxide Semiconductor
<b>CMSIS</b>	Cortex Microcontroller Software Interface Standard
<b>CTS</b>	Clear to Send

<b>dB</b>	Decibel
<b>dBm</b>	Decibel-milliwatts (mW)
<b>Decimal</b>	A base-10 number system
<b>DMA</b>	Direct Memory Access
<b>EEPROM</b>	Electrically Erasable Programmable Read Only Memory that is byte wise readable and writable.
<b>FET</b>	Short for MOSFET
<b>FIFO</b>	First in, first out
<b>FILO</b>	First in, last out
<b>Flash Memory</b>	Electrically erasable read only memory that is organized and erased by page. Individual bytes are writable.
<b>FSTN</b>	Filtered Super Twisted Nematic
<b>GIE</b>	Global Interrupt Enable
<b>GSM</b>	Global System for Mobile communication
<b>GPIO</b>	General Purpose Input Output
<b>GUI</b>	Graphical User Interface

<b>Hexidecimal</b>	A base-16 number system typically written with 0x prefix (e.g. 0x20)
<b>HSPA</b>	High Speed Packet Access
<b>I<sup>2</sup>C</b>	Inter-IC communication (a.k.a. IIC or I2C)
<b>IAR</b>	Integrated Development Software vendor for various microprocessors and microcontrollers
<b>IC</b>	Integrated Circuit
<b>IDE</b>	Integrated Development Environment. An IDE typically has an editor, compiler, and debugger.
<b>IIC</b>	Inter-IC communication (a.k.a. I <sup>2</sup> C or I2C)
<b>Instruction Set</b>	The logical description of the hardware that is designed into a microprocessor that gives it the functions that it can perform. Common instructions are ADD, MULTIPLY, and MOVE.
<b>IoT</b>	Internet of Things
<b>ISR</b>	Interrupt Service Routine
<b>JIT</b>	Just in time
<b>LCD</b>	Liquid Crystal Display
<b>LSb</b>	Least Significant byte
<b>LSB</b>	Least Significant Bit

<b>LTE</b>	Long Term Evolution
<b>MCU</b>	Microcontroller Unit
<b>MISO</b>	Master In Slave Out or a delicious sauce
<b>MOSFET</b>	Metal Oxide Semiconductor Field Effect Transistor
<b>MOSI</b>	Master Out Slave In
<b>MPU</b>	Microprocessor Unit
<b>MSB</b>	Most Significant Bit
<b>MSb</b>	Most Significant Byte
<b>Nibble</b>	A group of 4 bits
<b>NMOS</b>	N-channel MOSFET
<b>Octal</b>	A base-8 number system
<b>OpCode</b>	Operation Code. The complete word consisting of an instruction and its arguments that are stored in each line of flash which the processor then executes.
<b>PAN</b>	Personal Area Network
<b>PDC</b>	Peripheral DMA Controller

<b>PMOS</b>	P-channel MOSFET
<b>POST</b>	Power on self test
<b>PWM</b>	Pulse Width Modulation
<b>RAM</b>	Random Access Memory. Volatile storage.
<b>RF</b>	Radio Frequency
<b>RTS</b>	Ready to Send
<b>ROM</b>	Read Only Memory. Non volatile storage.
<b>RTOS</b>	Real Time Operating System
<b>SOC</b>	System on Chip
<b>SPI</b>	Serial Peripheral Interface
<b>TC</b>	Timer Counter
<b>TDMA</b>	Time Division Multiple Access
<b>TFT</b>	Thin film Transistor
<b>Toggle</b>	Reversing the logic state of a signal. E.g. Low to High, Zero to One, or On to Off.



<b>TWI</b>	Two-wire interface (a.k.a. I <sup>2</sup> C)
<b>UART</b>	Universal Asynchronous Receiver Transmitter
<b>UMTS</b>	Universal Mobile Telecommunications System
<b>USART</b>	Universal Synchronous Asynchronous Receiver Transmitter

## Index

### Symbols

2.4GHz Transceiver 55

2-wire interface 405

32kB Kickstart 63

48-bit timer 235

### A

Acknowledged Data 479

Active interrupts 197

ADC API 468

ADC Driver 464

ADC Hardware 460

ADC Initialization 466

ADC Interrupt 467

ADC Peripheral 460

ADC Registers

Analog Control 462

Channel Data 464

Channel Enable 464

Control 462

Disable 464

Extended Mode 463

Interrupt Enable 464

Last Converted Data 464

Mask 464

Mode 463

Status 463, 464

ADC State Machine 468

Additional Interrupt Modes Enable 177

Address bytes 439

Advanced Peripheral Bus 25

Aliasing 452

Alpha-numeric displays 438

analog supply input 461

Analog to Digital Conversion 451

ANT 82

ANT Channel ID 479

ANT communication 164

ANT Data 536

ANT\_DATA 518

ANT Functions

AntAbortMessage() 500

AntAssignChannel() 533

AntCalculateTxChecksum() 522

AntDeQueueApplicationMessage() 522

AntDeQueueOutgoingMessage() 525

AntExpectResponse() 507

AntGetdBmAscii() 538

AntInitialize() 526

AntOpenChannelNumber() 534

AntOpenScanningChannel() 534

AntParseExtendedData() 515

AntQueueAcknowledgedMessage() 536

AntQueueBroadcastMessage() 536

AntQueueExtendedApplicationMessage()  
520

AntQueueOutgoingMessage() 523

AntRadioStatusChannel() 535

AntReadAppMessageBuffer() 536

AntRxMessage() 495

AntSM\_TransmitMessage() 531

AntSyncSerialInitialize() 504

AntTickExtended() 519

AntTxMessage() 500

AntUnassignChannelNumber() 534

G\_u32AntFlags 525

ANT-Host communication 476

ANT-Host messaging 494

ANT Interface 482

ANT message Frame 482

ANT Message Protocol 475

ANT Physical Layer 476

ANT radio protocol 315

ANT Radio System 473

AntSM\_Idle() 529

AntSM\_ReceiveMessage() 531

ANT Stack 475  
 ANT State Machine 526  
 ANT Sub-System 488  
 ANT\_TICK 491, 518  
 ANTware II 82, 487  
 API description 115  
 API functions 115, 182, 210, 223, 416  
 API-level firmware 117  
 Application Program Status Register 92  
 Arduino 51, 396  
 ARM 29  
 ARM cores 91  
 ASCII 316  
 ASCII conversions 402  
 ASCII development board 51, 396  
 ASCII LCD 433, 434, 438  
 Assembler 26  
 assembly language 91  
 Assembly Language Syntax 94  
 asynchronous 323  
 Atmel ATSAM3U2C 65  
 Atmel Cortex 277  
 Audible frequency 452  
 Audio API Functions 258  
 Audio Bits 268  
 Audio function initialization 256  
 auto bauding 312  
 Automatic generation 117

## B

Bandwidth 452  
 baud rate 314  
 Baud Rate Generator 321, 322  
 Beagleboard 146  
 binary 32  
 bit-bash 365  
 Bit Bashing 262  
 Bit descriptions 134  
 bit-wise definitions 134

Blade board 397  
 Blade connector 460  
 Blade Daughter Board 396  
 Blade Firmware 398  
 BLE 82  
 Blinking 188  
 Block-box perspective 131  
 Block Control Register 239  
 Block Diagram 51  
 Bluetooth Low Energy 473  
 Board Audio Driver 256  
 Board Support Package 132  
 Boolean flag 214  
 Boolean value 223  
 Boolean variable 275  
 bootloader 27  
 Braces { } 127  
 breakpoint 230, 485  
 Broadcast messages 480  
 buck converter 49  
 Burst Data 480  
 Button API 222  
 Button Driver 208  
 Button held 209  
 Button history 209  
 Button Operation 209  
 button status 219  
 Button Typedefs 211

## C

callback pointer 243  
 CANBUS 273  
 C/C++ Compiler 66  
 C code source files 95  
 Channel Control Register 237  
 Channel Event 486  
 Channel ID 478  
 Channel Mode Register 237  
 Channel Parameters 478

- Channel Period 478
- Channel Response message 484
- Channel Type 478
- Character RAM Addresses 441
- checksum 69, 315
- Chip Select 377
- Clearing an interrupt 199
- Clipping 455
- clock behavior 364
- Clock Control 235
- clock drift 313
- clock frequency 30
- Clock Generator 139
- Clocking 312
- clock setup 141
- clock signal 30, 312
- clock stretching 407
- Cloning 84
- CMOS 38
- CMSIS 134, 201
- CMSIS-CORE 23
- Coding Conventions 125
- collector-emitter 46
- Communication peripherals 207
- Communications peripherals 274
- Compilers 109
- concurrent mode 82
- Configuration File 118
- Configuration messages 485
- context preservation 198
- Continuous message stream 480
- Control byte 440
- Control Data 437
- Core Registers 91
- Cortex-M0 23, 55, 475
- Cortex-M3 22, 25, 91
- Cortex-M4 23
- Cortex Microcontroller 132
- Counter Interrupts 207
- Counter Value 238
- C-primer 115
- CPU registers 79, 80, 107
- C-runtime library 137
- crystal oscillator 31
- C-syntax 117
- CYCLECOUNTER 79
- D**
- Data Errors 315
- data lines 313
- Data Link 476
- Data Link layer 309
- Data Processing instructions 100
- Data Receive 340
- Data Structures 115, 490
- Data Transmission 273
- Data Transmit 336
- DDRAM 441
- dead time 251
- debouncing 208, 211
- Debug API 346
- Debugger 34, 69
- Debugger Driver 106
- Debugger setup 70
- Debugger toolbars 78
- Debug Programmer Access 352
- DebugSM\_CheckCmd 358
- DebugSM\_Idle 354
- Debug Task 345
- Dereference 102
- Development Board 50
- Development Tools 81
- Device Number 479
- Device Type 479
- Diagnostics 66
- Differential Measurement 458
- Direct Memory Access 276
- Disassembly window 75

Displaying characters 317

DMAC 277

DMA Controller

ADC 278

I2C / Two-wire Interface 278

PWM 278

Serial Peripheral Interface 278

UART 278

USART 278

DMA transfer function 301

Dot-matrix LCDs 434

Doxygen 117

Doxygen documentation 118

Doxygen tags 122, 129

Driver Implementation 184

Duty cycle 247

Dynamic Memory Allocation 342

Dynastream 474

## E

edge detection 209

Edge Select 177

edge-triggered interrupts 407

EEPROM 26

EiE Audio Hardware 252

EiE firmware 84

EiE Messaging Task 289

EiE SPI Driver 373

EiE UART Driver 326

Embedded C 115

Embedded designers 117, 191

Embedded Peripheral 169

Empty project 64

ENDTX interrupt code 425

ENOB 456

enum 204

Error 456

Exception entry 201

Exception model 199

Exception priorities 200

External Hardware 162

## F

Falling Edge or Low Level 177

fault protection 251

FET 42

FIFO 300, 470

FIFO buffer 301

Filter configuration 176

Firewire 309

firmware 19

Flag register 33

FreeRTOS 146

full-duplex transmission 367

Function calls 115

function declarations 115, 129

Function parameters 126

Functions

AddNewMessageStatus () 303

atoi() 319

bool Adc12StartConversion() 469

ButtonStartDebounce() 216

DebugInitialize() 350

DebugPrintf() 346

DebugPrintNumber() 347

DebugRxCallback() 351

DebugScanf() 351

DebugSM\_Error() 360

DeQueueMessage() 301

\_disable\_fault\_irq() 196

\_disable\_irq() 196

\_enable\_fault\_irq() 196

\_enable\_irq() 196

GpioSetup() 174, 205

GpioSetup(void) 177

I2CRepeatedStart() 410

I2CStart() 410

I2CStop() 410

InterruptSetup() 205  
IsButtonHeld() 222  
IsButtonPressed() 222  
IsTimeUp() 220, 444  
LcdClearChars() 448  
LcdCommand() 447  
LcdInitialize() 444  
LcdMessage() 447  
LedInitialize() 181, 185  
LedOff() 187, 267, 287  
LedOn() 186, 267, 287  
LedRunActiveState() 181  
LedSM\_Idle() 264  
LedToggle() 187, 188  
main() 288  
memcpy() 299  
MessagingInitialize() 295  
NVIC\_DisableIRQ( (IRQn\_Type)x) 204  
NVIC\_EnableIRQ( (IRQn\_Type)x) 204  
PWMAudioOff() 261  
PWMAudioOn() 261  
PWMAudioSetFrequency() 258  
PWMSetupAudio() 256  
PWMSetVolume() 262  
QueryMessageStatus() 305  
QueueMessage() 296  
ReadByte() 410  
SystemSleep() 229  
SysTickSetup() 228  
TimerInitialize() 239, 243  
TimerSet() 241  
TimerStart() 242  
TimerStop() 242  
TwiWriteData() 444  
UpdateMessageStatus() 304  
UserApp1Initialize() 401  
UserApp1RunActiveState() 288  
UserApp1SM 402  
void Adc12AssignCallback( ) 468

WasButtonPressed() 222  
WriteByte() 410  
Fundamentals 32

## G

gas detector 144  
general interrupt 204  
GitHub 84  
GitHub cloud 82  
GitHub Desktop 82  
Glitch Input Filter Enable 175  
global interrupt enable 196  
Global variables 128  
GPIO 104, 137, 161, 192, 367  
GPIO Initialization 143  
GPIO Interrupts 207  
GPIO setup 178  
GPS tracker 146  
GUI 82, 119

## H

half-duplex design 395  
handshaking 477  
Hard Faults 286  
HB test point 157  
HD44780 standard 436  
Headers section 74  
Heap 520  
heap space 344  
hexadecimal 26, 32  
high-level functions 132  
Hi-Z inputs 163  
Host Microcontroller 436  
Hungarian Notation 126

## I

I<sup>2</sup>C 273, 405  
I<sup>2</sup>C repeaters 406  
I<sup>2</sup>C Signaling 408  
IAR compiler 148

- IAR EWARM 61
- IAR files 72
- IAR information screen 62
- IAR installation 61, 62
- IAR Integrated 61
- IAR Simulator 75
- IIC 273
- Infinite Loop 144
- INIT values 204
- Instruction Mnemonic 94
- Instructions 93
- Intel Hex 26
- Inter-Integrated Circuit 405
- Interrupt configuration 213
- Interrupt Enable 175
- Interrupts 193, 194, 195
- Interrupts and C 203
- interrupt service routine 193
- Interrupt Service Routine 196
- interrupt source 194, 204
- Interrupts priorities 197
- Interrupts set flags 199
- Intrinsic functions 201
- IPR registers 204
- IRDA 273
- IRQ flag 213
- IRQn\_Type enum 205
- ISO model 475
- J**
- J-Link 26, 56
- J-LINK pinouts 50
- JTAG 26
- JTAG pins 163
- L**
- LCD Application 446
- LCD Character RAM 441
- LCD Command Set 442
- LCD Controllers 435
- LCD Hardware 433
- LCD Initialization 443
- LCD Interface 436
- LCD Pixels Characteristics 434
- LCD vendor 437
- Least Significant Bit 32
- LED brightness 248
- LED definition 212
- LED Driver 104, 161, 181
- LED functionality 182
- LED PWM cycle 263
- LED PWM Design 263
- LED PWM driver 263
- Library configuration 65
- Light Emitting Diodes 39
- LINBUS 273
- Linear Regulators 47
- Linked Lists 284
- Linker configuration 68
- Linker files 67
- List 66
- Load Instructions 101
- load register 101
- load-store architecture 91
- local callback 244
- Logical expressions 115
- Logic Block Diagram 170
- low impedance 459
- LPT parallel port 309
- LSB 314
- M**
- Main assembler example 97
- Main Clock 142
- main oscillator 226
- Map File 190
- maskable 194
- Master and Slave addresses 406
- Master branch 86

Master-In-Slave-Out 364  
Master Receive 375  
Master Transmit 375  
MCK clock 225  
MCK scaling 250  
MC peripheral 164  
MCU-IC interface 440  
Memory allocation 115  
message counter 298  
Message Handling 483  
message slots 292  
Message Task Data Structures 290  
Messaging Public Functions 305  
Messaging State Machine 306  
microamp current 226  
microcontroller 15, 19, 145  
Microcontroller Programs 25  
MicroC/OS 146  
microprocessors 19  
MISRA 125  
Missing codes 457  
MODBUS 273  
Mode 119  
MODULE SUMMARY 190  
MOSFET 42  
Most Significant Bit 32  
MOV instruction 95  
MOVPLS instruction 95  
MOVS instructions 98  
MQX 146  
MSB 314  
MSB flipping 394  
MSP-430 61  
Multi-driver Enable 175  
Multiple User Tasks 270  
Multi-Slave configurations 405  
Musical notation 269

**N**

NACK and Errors 431  
nested parenthesis 128  
Nested Vectored Interrupt Controller 170,  
195, 201  
nesting 197  
Network 476  
Network key 533  
Network Number 478  
network processor 476  
New Project 63  
NMOS 42  
Node 477  
Noise 457  
noise-intolerant circuits 50  
non-maskable 195  
non-volatile flash 22  
nRFgo Studio 82  
null modem 311  
Number Systems 32  
NVIC flag 204  
Nyquist Frequency 453

**O**

Object-Oriented 214  
octal 32  
ODSR bits aligned 108  
Ohm's Law 36  
open-drain drivers 406  
Operand 2 functions 96  
Operands 94  
Operating Systems 145  
Operator Precedence 130  
Operators 128  
oscilloscope 158  
OTP 26  
Output converter 67  
Output Data Status Register 108  
Output Enable 175



- Output Enable Register 171
- Output Status Register 107
- Output Write Enable 177
- OVERHEAD 109
- P**
- Pairing 481
- parasitic capacitances 46
- passthrough mode 356
- Passthrough Mode 353
- payloads 489
- PDC 412
- PDC registers 279, 281
- periodic signals 247
- Peripheral AB Select 176
- Peripheral initialization 410
- Peripheral Interrupts 207
- personal area networks 473
- Physical layer 309
- Piezoelectric 251
- PIMO 311
- Pin Allocation 163
- Pin Data Status 175
- pin-mapping 164
- PIO 135
- PIOA 135, 165
- PIOB 165
- PIOB definitions 104
- PIO block diagram 170
- PIOB peripheral 104
- PIOB registers 106
- PIO Controller 169
- PIO Enable 175
- PIO Internal Hardware 169
- PIO Interrupts 212
- PIO peripheral interrupts 213
- PIO peripherals 165
- PIO register 186
- PIO Registers 174
- PMOS 42
- Pointers 115
- POMI 311
- Port A 168
- PORTA 182, 219
- PORTB 182, 219
- Power Management Controller 141, 170
- Power On Self-Test 403
- Power Supplies 46
- Precision 456
- Preprocessor 66
- Present Data Status Register 111
- Priorities 203
- processor logic 93
- Processor RAM allocation 343
- Program Counter 92
- Program Status Register 92
- Program Structures 144
- Public network 478
- Pull-up Enable 176
- Pulse Width Modulation 233, 247
- Push origin 89
- PWM
  - Channel Counter Registers 256
  - Channel Duty Cycle 255
  - Channel Mode Registers 255
  - Channel Period 255
  - Clear 254
  - Clock Register 253
  - Deadtime Registers 254
  - Disable 253
  - Enable 253
  - Event Line Registers 254
  - Fault Mode 254
  - Interrupt Enable 254
  - Mask 254
  - Output Override 254
  - Protection Value 254
  - Status Registers 253

- Synchronization Registers 254
- Write Protect Control Register 254
- Write Protect Status Register 254
- PWM and Audio 251
- PWM duty cycles 265
- PWM peripheral 249
- PWM pins 163
- PWM Registers 253
- Q**
- QDEC Interrupt Enable 239
- QNX 146
- Quantization 451
- R**
- RAM 20, 52
- RAM locations 101
- Raspberry PI 146
- RC oscillator 30, 137, 142
- Reading Character Input 349
- Reference Voltages 457
- Register window 97, 101
- resister name 136
- Resolution 454
- Resource Conflicts 275
- RF Frequency 478
- round-robin scheduler 151
- RS-232 15, 273
- RS-232 Overview 309
- RS-232 protocol 81
- RS-232 standard 310
- RS-422 273
- RS-485 273
- RTOS 147
- S**
- SAM3U 22, 31
- SAM3U2 24, 25, 52, 162, 170
- SAM3U abbreviations 412
- Sampling 451
- sampling clock 312
- SATA 309
- scheduler 146
- SD card SPI 164
- semaphore 271, 275
- serial clock 312, 364
- Serial communication 309
- Serial Drivers 491
- Serial Peripheral Interface 309
- serial port 310
- serial protocols 309
- Signal Conditioning 458
- Signaling 313
- Single Measurement 458
- Slave Channel ID 481
- Slave Receive 376
- Slave Scanning 487
- Slave Transmit 376
- Sleep function 193
- Sleep timer 225
- slow clock 137
- SMART devices 474
- Smartphones 473
- SM-based system 149
- SOC 82
- Special Commands 122
- Special Comment Blocks 120
- special function pins 163
- SPI 273, 315
- SPI communication 477
- SPI configurations 366
- SPI data communication 364
- SPI Data Structures 378
- SPI Driver Functions
  - SspAssertCS() 383
  - SspDeassertCS() 383
  - SspGenericHandler() 386
  - SspQueryReceiveStatus() 385
  - SspReadByte() 384

- SspReadData() 384
- SspRelease() 382
- SspRequest() 381
- void SSP0\_IRQHandler(void) 380
- void SSP1\_IRQHandler(void) 380
- void SSP2\_IRQHandler(void) 380
- void SspInitialize(void) 380
- void SspManualMode(void) 380
- void SspRunActiveState(void) 380
- SPI Hardware 368
- SPI Mode Behaviour 373
- SPI protocol 363
- SPI Registers
  - Baud Rate Generator 373
  - Channel Status Register 373
  - Chip Select Registers 371
  - Control Register 370
  - Interrupt Disable 371
  - Interrupt Enable 371
  - Mask Registers 371
  - Receive Data Register 370
  - Receive Holding Register 370
  - Status Register, 373
  - Status Registers 373
  - Transmit Data Register 371
  - Transmit Holding Register 371
  - USART Mode Register 370
  - Write Protection Mode 373
- SPI Signaling 363
- SPI Slave devices 364
- SSP peripheral 491
- Ssp State Machine 392
- Stack declaration 191
- stack frame 198
- START condition 416
- State Declarations 130
- State diagram 349, 426
- State Machine 147
- ST Discovery board 51
- STOP condition 416
- Store Instructions 101
- store register 101
- Storing characters 317
- struct's pointer 136
- structures 115
- subset of Exceptions 194
- Super Loop 147
- Switches 37
- Switching power supplies 49, 249
- Switch statements 127
- Symbol Definitions 127
- Symbolic Memory windows 102
- Synchronous Serial Protocol 369
- syntax 115
- System Tick
  - Calibration Register 227
  - Control and Status Register 227
  - Current Value Register 227
  - Reload Value Register 227
- SysTick breakpoint 231
- SysTick interrupt 194
- T**
  - Tabs and Indenting 130
  - Tag-Connect 28
  - TC Register 238
  - Tera Term 81
  - Terminal Control Codes 361
  - Test Driven Development 386
  - Timer API 240
  - Timer Counter 233, 249
  - timer/counter pins 163
  - Timer Driver 239
  - Timer Peripheral 233
  - Timing 203
  - Toggling the LED 156
  - transformers 47
  - Transistors 42

- Transmission Type 479
- Transmit Data Types 479
- Transport layers 476
- TWI 273, 405
- TWI Data Structures 417
- TWI Driver 416
- TWI Driver Functions
  - TwiInitialize() 418
  - TwiManualMode() 419
  - TwiReadData() 422
  - TwiRunActiveState() 419
  - TwiWriteData() 420
- TWI Hardware 411
- TWI interrupt handler 419
- TWI peripheral registers 423
- TWI queue 416
- TWI Receive 428
- TWI Registers
  - Clock Waveform Generator 414
  - Control Register 414
  - Internal Address Register 414
  - Interrupt Disable 415
  - Interrupt Enable 415
  - Mask Registers 415
  - Master Mode Register 414
  - Receive Holding Register 416
  - Slave Mode Register 414
  - Status Register 415
  - Transmit Holding Register 416
- TWI State Machine 423
- TWI Transmit 423
- Type Definitions 125
- U**
- UART 28, 309
- UART Driver Design 336
- UART Driver Functions
  - UartInitialize() 329
  - UartRelease() 331
  - UartRequest() 330
  - UartWriteByte() 334
  - UartWriteData() 333
- UART Interrupts 335
- UART Peripheral 320
- UART registers
  - Baud Rate Generator Register 325
  - Channel Status Register 325
  - Control Register 325
  - Disable 325
  - Interrupt Enable 325
  - Mask Register 325
  - Mode Register 325
  - Receiver Holding Register 325
  - Receiver Time-out Register 326
  - Status Register 326
  - Transmit Holding Register 325
  - Transmitter Timeguard Register 326
  - Write Protect Mode 326
- UART Registers 324
- UART Task Data Structures 327
- unexpected behavior 199
- Unicode 316
- unprogrammed MCU 137
- UnreadChar pointer 402
- unsigned char 126
- unsigned long 126
- unsigned short 126
- UPLL circuit 140
- USART 320
- USART peripheral 321
- USART registers 326
- USB 15, 52, 273
- USB dongle drivers 62
- V**
- Variable types 115
- Vector Table 196, 200, 203
- Version control 83

voltage regulator 48

## **W**

warning flag 298

Watchdog Control Register 138

Watchdog Mode Register 138

Watchdog timer 137

watermarking 298

While loop 76

White Space 128

Windows start menu 83

wireless technology 473

Workspace 77

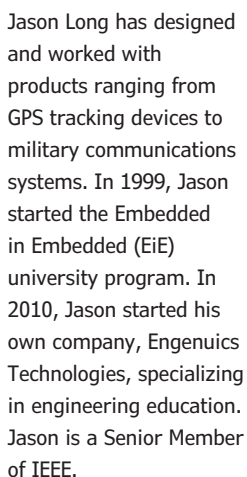
## **Z**

Zigbee consortium 473





# ARM Cortex-M Embedded Design from 0 to 1



9 781907 920738

[www.elektor.com](http://www.elektor.com)

Hobbyists can mash together amazing functional systems using platforms like Arduino or Raspberry Pi, but it is imperative that engineers and product designers understand the foundational knowledge of embedded design. There are very few resources available that describe the thinking, strategies, and processes to take an idea through hardware design and low-level driver development, and successfully build a complete embedded system. Many engineers end up learning the hard way, or never really learn at all.

ARM processors are essentially ubiquitous in embedded systems. Design engineers building novel devices must understand the fundamentals of these systems and be able to break down large, complicated ideas into manageable pieces. Successful product development means traversing a huge amount of documentation to understand how to accomplish what you need, then put everything together to create a robust system that will reliably operate and be maintainable for years to come.

This book is a case study in embedded design including discussion of the hardware, processor initialization, low-level driver development, and application interface design for a product. Though we describe this through a specific application of a Cortex-M3 development board, our mission is to help the reader build foundational skills critical to being an excellent product developer. The completed development board is available to maximize the impact of this book, and the working platform that you create can then be used as a base for further development and learning.

The Embedded in Embedded program is about teaching fundamental skill sets to help engineers build a solid foundation of knowledge that can be applied in any design environment. With nearly 20 years of experience in the industry, the author communicates the critical skill development that is demanded by companies and essential to successful design. This book is as much about building a great design process, critical thinking, and even social considerations important to developers as it is about technical hardware and firmware design.



LEARN > DESIGN > SHARE