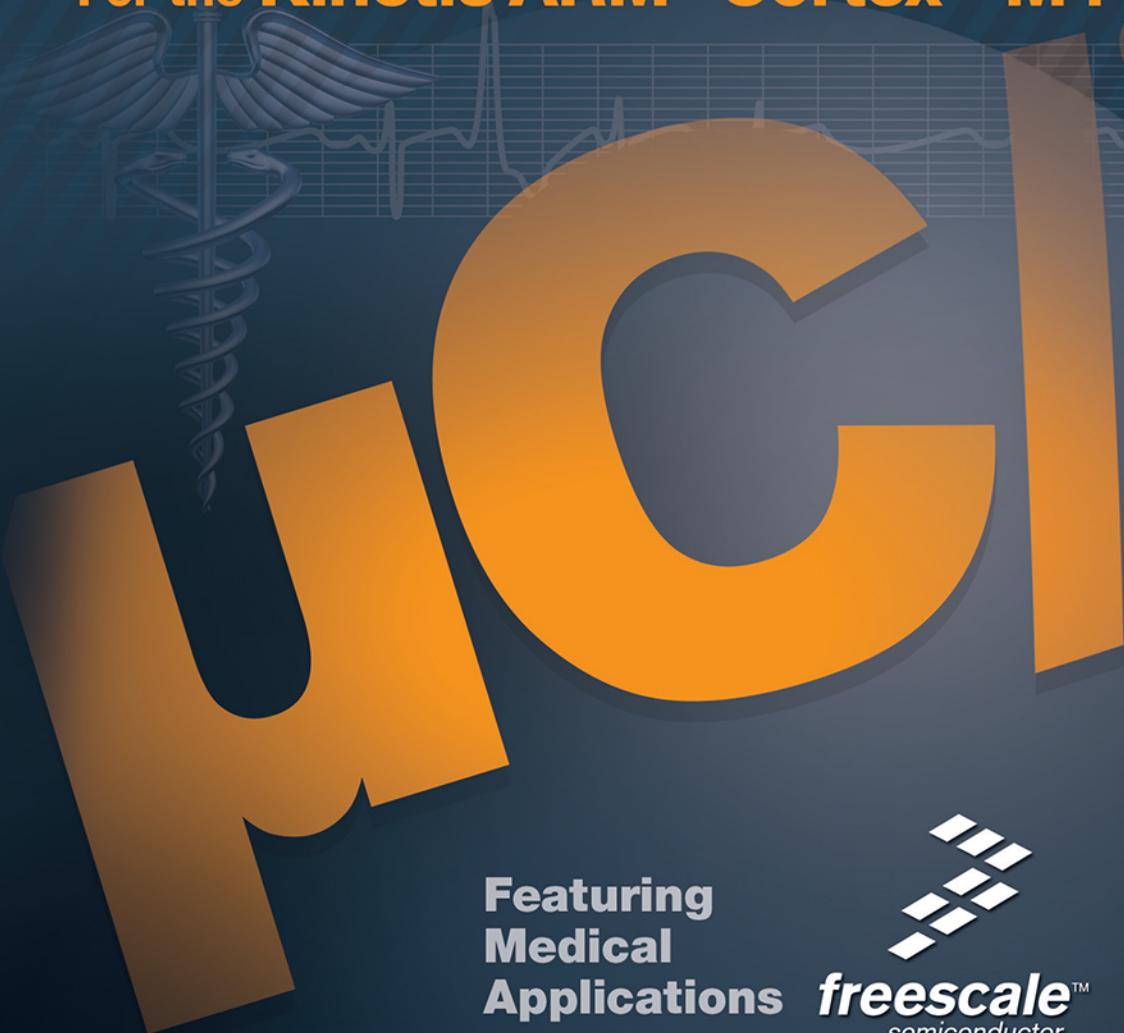


μC/OS-II®

The Real-Time Kernel

For the Kinetis ARM® Cortex™-M4



Featuring
Medical
Applications **freescale**™
semiconductor



Micrium®

Jean J. Labrosse
Juan P. Benavides
José Fernández-Villaseñor, M.D.



and the Freescale

Kinetis ARM® Cortex™-M4

Jean J. Labrosse

Juan P. Benavides

and

José Fernandez-Villaseñor, M.D.

Micri^μum
Press

Weston, FL 33326

Micrium Press
1290 Weston Road, Suite 306
Weston, FL 33326
USA
www.micrium.com

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

Copyright © 2015 by Micrium Press except where noted otherwise. Published by Micrium Press. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher; with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

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

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

978-0-9823375-2-3
100-uCOS-II-Freescale-Kinetis-001



Table of Contents

μC/OS-II and the Freescale Kinetis ARM® Cortex™-M4

Foreword	9
Chapter 1	Introduction
1-1	Foreground/Background Systems
1-2	Real-Time Kernels
1-3	RTOS (Real-Time Operating System)
1-4	μC/OS-II
1-5	Conventions
1-6	Chapter Contents
11	
12	
13	
15	
15	
17	
18	
Chapter 2	Directories and Files
2-1	Application Code
2-2	Board Support Package (BSP)
2-3	μC/OS-II, CPU Independent Source Code
2-4	μC/OS-II, CPU Specific Source Code
2-5	Summary
23	
25	
26	
27	
29	
30	
Chapter 3	Getting Started with μC/OS-II
3-1	Single Task Application
3-2	Multiple Application Tasks with Kernel Objects
33	
34	
41	

Table of Contents

Chapter 4	Introduction to Medical Applications	49
Chapter 5	Certification of Medical Systems	55
5-1	“Off-the-Shelf” Software	56
5-2	Medical Device Software – Recalls	57
5-3	Safety Critical	58
5-4	Traceability	60
5-5	Cost of Safety Critical Software	61
5-6	Medical Device Market Regulatory Environment	63
5-6-1	The Regulators	63
5-6-2	Industry and Standards Bodies	63
5-6-3	The Regulatory Environment in the United States	64
5-6-4	FDA 510(k)	65
5-6-5	FDA Premarket Approval	65
5-6-6	FDA Development Guidance	65
5-6-7	Device Class	66
5-6-8	Level of Concern (FDA/CDRH)	67
5-7	Regulatory Environment in the European Union	69
5-7-1	EU Device Classes	69
5-8	Medical Standards	70
5-8-1	IEC 62304	70
5-8-2	ISO 14971	72
5-8-3	ISO 13485	72
5-8-4	Medical Software in the Future	72
5-9	Common Safety-Critical Development Standards	73
5-10	Standards Bodies and Worldwide Standards Organizations	74
5-11	FDA Guidance and Documents	76
5-12	References	78
Chapter 6	Freescale’s Tower System Controller TWR-K53N512	81
6-1	TWR-K53N512 Pin Usage	84
6-2	TWR-K53N512 Jumper Settings	87
6-3	The Freescale TOWER SYSTEM KIT for the K53	90
0-1	Tower System Module TWR-SER	93
6-4	TWR-SER Jumper Settings	96
6-5	Expanding the Capabilities	98

Chapter 7	Setup	101
7-1	Downloading µC/OS-II Projects for this Book	101
7-1-1	\Examples	103
7-1-2	\Software	104
7-2	Downloading µC/Probe	106
7-3	Downloading the IAR Embedded Workbench for ARM	107
7-4	Setting up the Hardware	107
7-5	Downloading the TWR-K53N512 documentation	110
Chapter 8	ECG / Heart Rate Monitor	111
8-1	The Heart	111
8-2	Biological Electrical Potentials	118
8-3	ECG Leads	121
8-4	Cardiovascular Diseases (CVD)	126
8-5	ECG Design	127
8-6	Running the Example Project	134
8-7	How the Code Works	141
8-7-1	Biomedical Signal Analysis	145
8-8	Summary	147
Chapter 9	Blood Glucose Meter	149
9-1	Glucose	149
9-2	Diabetes Mellitus	155
9-3	Blood Glucose Sensor	157
9-4	Blood Glucose Meter Design	162
9-5	Running the Example Project	166
9-6	How the Code Works	171
9-7	Summary	177
Chapter 10	Pulse Oximeter	179
10-1	Respiration	180
10-2	Pulse Oximetry	185
10-3	Pulse Oximeter Design	190
10-4	Running the Example Project	195
10-5	How the Code Works	200
10-5-1	Biomedical Signal Analysis	205
10-5-2	Data Processing State Machine	207

Table of Contents

10-6	Summary	211
Chapter 11	Blood Pressure Monitor	213
11-1	Blood Pressure	214
11-2	The Baroreceptor Reflex	216
11-3	Renin-Angiotensin-Aldosterone System (RAAS)	218
11-4	Hypertension	222
11-5	Indirect Measurement of Arterial Blood Pressure	224
11-6	Design of a Blood Pressure Monitor	226
11-7	Running the Example Project	237
11-8	How the Code Works	242
11-8-1	Biomedical Signal Analysis	246
11-8-2	Data Processing State Machine	249
11-9	Summary	252
Appendix A	μC/OS-II Port for the Cortex-M4	253
A-1	os_cpu.h	254
A-2	os_cpu_c.c	258
A-2-1	os_cpu_c.c – OSTaskIdleHook()	259
A-2-2	os_cpu_c.c – OSInitHookBegin() and OSInitHookEnd()	259
A-2-3	os_cpu_c.c – OSStatTaskHook()	260
A-2-4	os_cpu_c.c – OSTaskCreateHook()	260
A-2-5	os_cpu_c.c – OSTaskDelHook()	260
A-2-6	os_cpu_c.c – OSTaskStkInit()	261
A-2-7	os_cpu_c.c – OSTaskSwHook()	265
A-2-8	os_cpu_c.c – OSTimeTickHook()	266
A-2-9	os_cpu_c.c – OS_CPU_SysTickHandler()	267
A-2-10	os_cpu_c.c – OS_CPU_SysTickInit()	268
A-3	os_cpu_a.asm	269
A-3-1	os_cpu_a.asm – OSStartHighRdy()	269
A-3-2	os_cpu_a.asm – OSCTxSw() and OSIntCtxSw()	270
A-3-3	os_cpu_a.asm – OS_CPU_PendSVHandler()	271
Appendix B	Micrium's μC/Probe	275
B-1	System Overview	275
B-2	μC/Probe on the Kinetis	278
B-3	Downloading μC/Probe	279

Appendix C	IAR Systems IAR Embedded Workbench for ARM	281
C-1	IAR Embedded Workbench for ARM – Highlights	282
C-2	Modular and Extensible IDE	284
C-3	Highly Optimizing C/C++ Compiler	286
C-4	Device Support	287
C-5	State-of-the-Art C-SPY® Debugger	288
C-6	C-SPY Debugger and Target System Support	289
C-7	IAR Assembler	289
C-8	IAR J-LINK Linker	289
C-9	IAR Library and Library Tools	290
C-10	Comprehensive Documentation	290
C-11	First Class Technical Support	290
Appendix D	Bibliography	291
Appendix E	Licensing µC/OS-II	293
Index	295

Table of Contents

Foreword

If you are reading these words I know one thing for sure: you are facing a challenge. That challenge falls within certain boundaries defined by your interest in this book. You need to learn how to use an RTOS, or you're planning to create a medical device for that rapidly growing market. You will face decisions on processor, platform, development tools, software design for life-critical systems, human machine interface, and more.

The good news is, you are in the right place for several reasons. For one thing, µC/OS has a long and storied history in the industry dating back to 1992. For another, you are smart enough to realize that writing everything from scratch is just plain dumb. With complex systems, especially those with life-critical or mission-critical applications, quality and reliability are absolutely paramount. Like any engineer, you want to use proven components. So here you are, reading and learning.

At Freescale we take an innovative approach to these problems, because we face them too. You need reliable components: processors, boards, OS, software stacks. You need them to all work together perfectly. Yet the simple fact is that your specific personal needs are not identical to everyone else. So how do we help you personally, while addressing a general market?

We take a solution approach, and this book fits perfectly into that strategy. The first component in that strategy is, of course, the processor. We've recently introduced the K50 family of microcontrollers, expanding upon our popular Kinetis ARM-Cortex M4 portfolio. This family of devices is focused on delivering the system level hardware to support a wide variety of measurement and communication requirements often found in medical devices. These are sophisticated and highly capable devices.

You need that processor on a board. What you need is not what "the other guy" needs. What peripherals should we include? What if you decide to change processors for some reason? We designed an innovative modular reference design platform called the Tower System – you can swap controller modules and various peripheral modules in and out of the system almost transparently. If you need a different peripheral, odds are excellent you

can just buy a module and plug it into the tower, and not buy an entirely new piece of hardware and start over. Several of our partners are jumping into this open modular design, creating a huge community of “Tower Geeks” developing products on this highly customizable reference design.

The solution is not complete without software, and that’s really where this book makes the greatest sense. There will be a lot going on in a modern medical device: continuous monitoring of an integrated measurement engine, processing of analog signals for the application and control functionality, human machine interface updates, and secure communication among devices and/or across networks. Getting all this to work together seamlessly and robustly is no easy task. Leveraging an RTOS during the system design will not only ease development, but also allow for optimal utilization of the variety of on-chip peripherals. This book will get you well started down that road.

On top of that, after introducing you to the fundamental principles of RTOS design and use, which you can apply to other RTOS designs, you have several real world examples of medical systems to build and experiment with: a heart rate monitor, a blood glucose meter, a pulse oximeter, and a blood pressure monitor.

There you have the main components of a real solution: a processor, a board with peripherals and drivers, an operating system, and the software required to put it to use to do something more than blink an LED. You still add the “secret sauce” that turns it into your product, but the Freescale solution puts you a long way up the learning curve. This book from Jean Labrosse is a big part of that.

The world of medical devices and applications continues to expand, presenting new opportunities and challenges for embedded system design. Freescale continues to innovate in this space to give the embedded designer the tools and the enablement to bring highly-capable and robust products to market in a timely and resource-efficient manner. I wish you the best of luck and trust you will find success as you tame your challenges. Thank you for choosing Freescale to help you along the way to making the world a better place.



Reza Kazerounian, Ph. D.
Senior Vice President and General Manager
Microcontroller Solutions
Freescale Semiconductor



Chapter

1

Introduction

Real-time systems are systems whereby the correctness of the computed values and their timeliness are at the forefront. There are two types of real-time systems, hard and soft real time.

What differentiates hard and soft real-time systems is their tolerance to missing deadlines and the consequences associated with those misses. Correctly computed values after a deadline has passed are often useless.

For hard real-time systems, missing deadlines is not an option. In fact, in many cases, missing a deadline often results in catastrophe, which may involve human lives. For soft real-time systems, however, missing deadlines is generally not as critical.

Real-time applications cover a wide range, but many real-time systems are embedded. An embedded system is a computer built into a system and not acknowledged by the user as being a computer. Embedded systems are also typically dedicated systems. In other words, systems that are designed to perform a dedicated function. The following list shows just a few examples of embedded systems:

Aerospace <ul style="list-style-type: none">■ Flight management systems■ Jet engine controls■ Weapons systems	Communications <ul style="list-style-type: none">■ Routers■ Switches■ Cell phones Computer peripherals <ul style="list-style-type: none">■ Printers■ Scanners	Office automation <ul style="list-style-type: none">■ FAX machines / copiers Process control <ul style="list-style-type: none">■ Chemical plants■ Factory automation■ Food processing Robots Video <ul style="list-style-type: none">■ Broadcasting equipment■ HD Televisions And many more
Audio <ul style="list-style-type: none">■ MP3 players■ Amplifiers and tuners	Domestic <ul style="list-style-type: none">■ Air conditioning units■ Thermostats■ White goods	

Real-time systems are typically more complicated to design, debug, and deploy than non-real-time systems.

1-1 FOREGROUND/BACKGROUND SYSTEMS

Small systems of low complexity are typically designed as foreground/background systems or super-loops. An application consists of an infinite loop (F1-1(1)) that calls modules (i.e., tasks) to perform the desired operations (background). Interrupt Service Routines (ISRs) shown in F1-1(3) handle asynchronous events (foreground). Foreground is also called interrupt level; background is called task level.

Critical operations that should be performed at the task level must unfortunately be handled by the ISRs to ensure that they are dealt with in a timely fashion. This causes ISRs to take longer than they should. Also, information for a background module that an ISR makes available is not processed until the background routine gets its turn to execute, which is called the task-level response. The worst-case task-level response time depends on how long a background loop takes to execute and, since the execution time of typical code is not constant, the time for successive passes through a portion of the loop is nondeterministic. Furthermore, if a code change is made, the timing of the loop is affected.

Most high-volume and low-cost microcontroller-based applications (e.g., microwave ovens, telephones, toys, etc.) are designed as foreground/background systems.

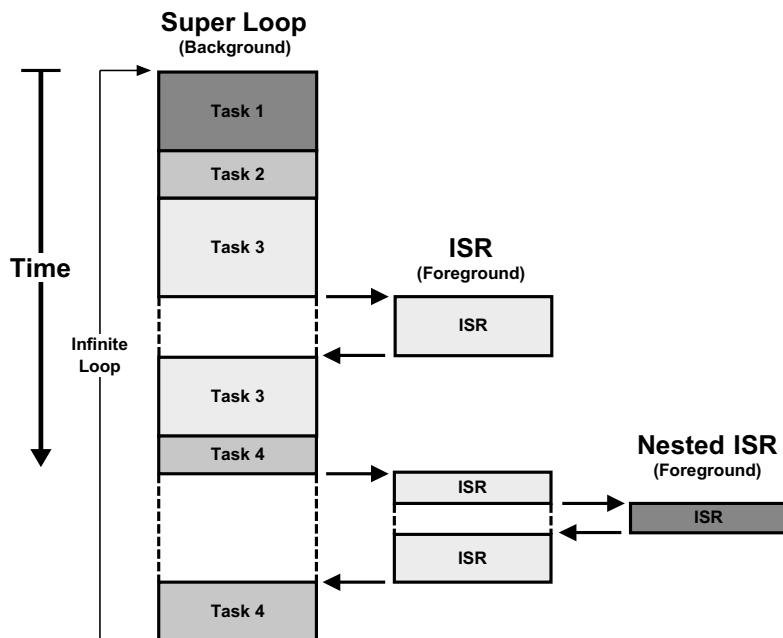


Figure 1-1 Foreground/Background (SuperLoops) systems

1-2 REAL-TIME KERNELS

A real-time kernel is software that manages the time and resources of a microprocessor, microcontroller or Digital Signal Processor (DSP).

The design process of a real-time application involves splitting the work into tasks, each responsible for a portion of the job. A task (also called a thread) is a simple program that thinks it has the Central Processing Unit (CPU) completely to itself. On a single CPU, only one task executes at any given time. A task is also typically implemented as an infinite loop.

The kernel is responsible for the management of tasks. This is called multitasking. Multitasking is the process of scheduling and switching the CPU between several tasks. The CPU switches its attention between several sequential tasks. Multitasking provides the illusion of having multiple CPUs and maximizes the use of the CPU. Multitasking also helps in the creation of modular applications. One of the most important aspects of multitasking is that it allows the application programmer to manage the complexity inherent in real-time applications. Application programs are easier to design and maintain when multitasking is used.

μ C/OS-II is a preemptive kernel, which means that μ C/OS-II always runs the most important task that is ready-to-run as shown in Figure 1-2.

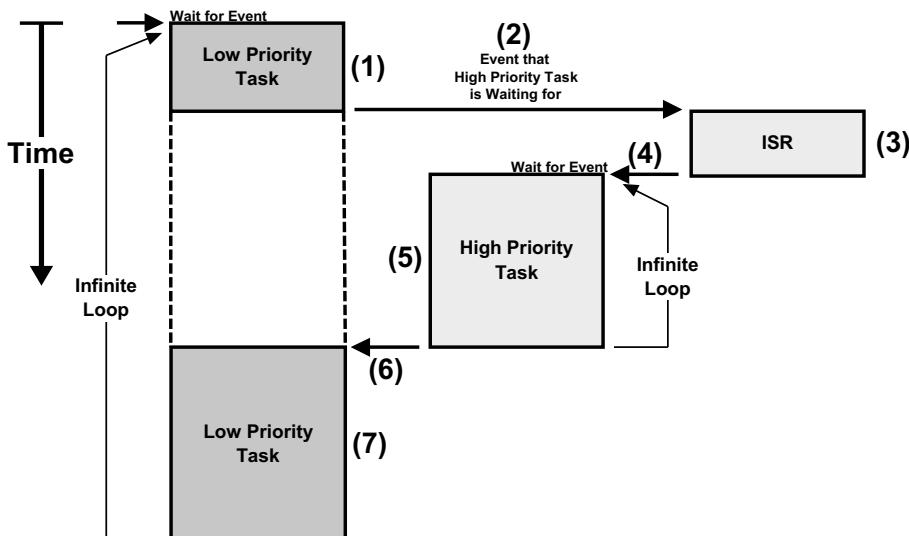


Figure 1-2 μ C/OS-II is a preemptive kernel

-
- F1-2(1) A low-priority task is executing.
 - F1-2(2) An interrupt occurs, and the CPU vectors to the ISR responsible for servicing the interrupting device.
 - F1-2(3) The ISR services the interrupt device, but actually does very little work. The ISR will typically signal or send a message to a higher-priority task that will be responsible for most of the processing of the interrupting device. For example, if the interrupt comes from an Ethernet controller, the ISR simply signals a task, which will process the received packet.
 - F1-2(4) When the ISR finishes, µC/OS-II notices that a more important task has been made ready-to-run by the ISR and will not return to the interrupted task, but instead context switch to the more important task.
 - F1-2(5) The higher-priority task executes and performs the necessary processing in response to the interrupt device.
 - F1-2(6) When the higher-priority task completes its work, it loops back to the beginning of the task code and makes a µC/OS-II function call to wait for the next interrupt from the device.
 - F1-2(7) The low-priority task resumes exactly at the point where it was interrupted, not knowing what happened.

Kernels such as µC/OS-II are also responsible for managing communication between tasks, and managing system resources (memory and I/O devices).

A kernel adds overhead to a system because the services provided by the kernel require time to execute. The amount of overhead depends on how often these services are invoked. In a well-designed application, a kernel uses between 2% and 4% of a CPU's time. And, since µC/OS-II is software that is added to an application, it requires extra ROM (code space) and RAM (data space).

Low-end single-chip microcontrollers are generally not able to run a real-time kernel such as µC/OS-II since they have access to very little RAM. µC/OS-II requires between 1 Kbyte and 4 Kbytes of RAM, plus each task requires its own stack space. It is possible for µC/OS-II to work on processors having as little as 4 Kbytes of RAM.

Finally, µC/OS-II allows for better use of the CPU by providing approximately 90 indispensable services. After designing a system using a real-time kernel such as µC/OS-II, you will not return to designing a foreground/background system.

1-3 RTOS (REAL-TIME OPERATING SYSTEM)

A Real Time Operating System generally contains a real-time kernel and other higher-level services such as file management, protocol stacks, a Graphical User Interface (GUI), and other components. Most additional services revolve around I/O devices.

Micrium offers a complete suite of RTOS components including: µC/FS (an Embedded File System), µC/TCP-IP (a TCP/IP stack), µC/GUI (a Graphical User Interface), µC/USB (a USB device and host stack), and more. Most of these components are designed to work standalone. Except for µC/TCP-IP, a real-time kernel is not required to use the components in an application. In fact, users can pick and choose only the components required for the application. Contact Micrium (www.micrium.com) for additional details and pricing.

1-4 µC/OS-II

µC/OS-II is a scalable, ROMable, preemptive real-time kernel that manages multiple tasks.

Here is a list of features provided by µC/OS-II:

Source Code: µC/OS-II is provided in ANSI-C source form. The source code for µC/OS-II is arguably the cleanest and most consistent kernel code available. Clean source is part of the corporate culture at Micrium. Although many commercial kernel vendors provide source code for their products, unless the code follows strict coding standards and is accompanied by complete documentation with examples to show how the code works, these products may be cumbersome and difficult to harness.

Intuitive Application Programming Interface (API): µC/OS-II is intuitive. Once familiar with the consistent coding conventions used, it is simple to predict the functions to call for the services required, and even predict which arguments are needed.

Preemptive multitasking: µC/OS-II is a preemptive multi-tasking kernel and therefore, µC/OS-II always runs the most important ready-to-run task.

Deterministic: Interrupt response with µC/OS-II is deterministic. Also, execution times of most services provided by µC/OS-II are deterministic.

Scalable: The footprint (both code and data) can be adjusted based on the requirements of the application. Adding and removing features (i.e., services) is performed at compile time through approximately 40 #defines (see `os_cfg.h`). µC/OS-II also performs a number of run-time checks on arguments passed to µC/OS-II services. Specifically, µC/OS-II verifies that the user is not passing `NULL` pointers, not calling task level services from ISRs, that arguments are within allowable range, and options specified are valid, etc. These checks can be disabled (at compile time) to further reduce the code footprint and improve performance. The fact that µC/OS-II is scalable allows it to be used in a wide range of applications and projects.

Portable: µC/OS-II has been ported to more than 45 CPU architectures.

ROMable: µC/OS-II was designed especially for embedded systems and can be ROMed along with the application code.

Mutual Exclusion Semaphores (Mutexes): Mutexes are provided for resource management. Mutexes are special types of semaphores that have built-in priority inheritance, which eliminate unbounded priority inversions.

Software timers: You can define any number of “one-shot” and/or “periodic” timers. Timers are countdown counters that perform a user-definable action upon counting down to 0. Each timer can have its own action and, if a timer is periodic, the timer is automatically reloaded and the action is executed every time the countdown reaches zero.

Error checking: µC/OS-II verifies that `NULL` pointers are not passed, that the user is not calling task-level services from ISRs, that arguments are within allowable range, that options specified are valid, that a pointer to the proper object is passed as part of the arguments to services that manipulate the desired object, and more. Most of µC/OS-II API functions return an error code concerning the outcome of the function call.

Can easily be optimized: µC/OS-II was designed so that it could easily be optimized based on the CPU architecture. Most data types used in µC/OS-II can be changed to make better use of the CPU’s natural word size.

Deadlock prevention: All of the µC/OS-II “pend” services include timeouts, which help avoid deadlocks.

Tick handling at task level: The clock tick manager in µC/OS-II is accomplished by a task that receives a trigger from an ISR. Handling delays and timeouts by a task greatly reduces interrupt latency.

User definable hooks: µC/OS-II allows the port and application programmer to define “hook” functions, which are called by µC/OS-II. A hook is simply a defined function that allows the user to extend the functionality of µC/OS-II. One such hook is called during a context switch, another when a task is created, yet another when a task is deleted, etc.

Built-in support for Kernel Awareness debuggers: This feature allows kernel awareness debuggers to examine and display µC/OS-II variables and data structures in a user-friendly way. The kernel awareness support in µC/OS-II can be used by µC/Probe to display this information at run-time.

Object names: Each µC/OS-II kernel object can have a name associated with it. This makes it easy to recognize what the object is assigned to. You can thus assign an ASCII name to a task, a semaphore, a mutex, a mailbox and a message queue. The object name can have any length, but must be NUL terminated.

1-5 CONVENTIONS

There are a number of conventions in this book.

First, you will notice that when a specific element in a figure is referenced, the element has a number next to it in parenthesis. A description of this element follows the figure and in this case, the letter “F” followed by the figure number, and then the number in parenthesis. For example, F3-4(2) indicates that this description refers to Figure 3-4 and the element (2) in that figure. This convention also applies to listings (starts with an “L”) and tables (starts with a “T”).

Second, you will notice that sections and listings are started where it makes sense. Specifically, do not be surprised to see the bottom half of a page empty. New sections begin on a new page, and listings are found on a single page, instead of breaking listings on two pages.

Third, code quality is something I've been avidly promoting throughout my whole career. At Micrium, we pride ourselves in having the cleanest code in the industry. Examples of this are seen in this book. I created and published a coding standard in 1992 that was published in the original µC/OS book. This standard has evolved over the years, but the spirit of the standard has been maintained throughout. The Micrium coding standard is available for download from the Micrium website, www.micrium.com

One of the conventions used is that all functions, variables, macros and #define constants are prefixed by “OS” (which stands for Operating System) followed by the acronym of the module (e.g., Sem), and then the operation performed by the function. For example OSSemPost () indicates that the function belongs to the OS (µC/OS-II), that it is part of the Semaphore services, and specifically that the function performs a Post (i.e., signal) operation. This allows all related functions to be grouped together in the reference manual, and makes those services intuitive to use.

You should notice that signaling or sending a message to a task is called posting, and waiting for a signal or a message is called pending. In other words, an ISR or a task signals or sends a message to another task by using OS???Post (), where ??? is the type of service: Sem, Flag, Mutex, Mbox. Similarly, a task can wait for a signal or a message by calling OS???Pend ().

1-6 CHAPTER CONTENTS

Figure 1-3 shows the layout and flow of the book. This diagram should be useful in understanding the relationship between chapters and appendices.

The first column on the left indicates chapters that should be read in order to understand µC/OS-II's structure as well as the examples. The columns to the right hand side consist of miscellaneous appendices that further expand on the information presented in the first columns on the left.

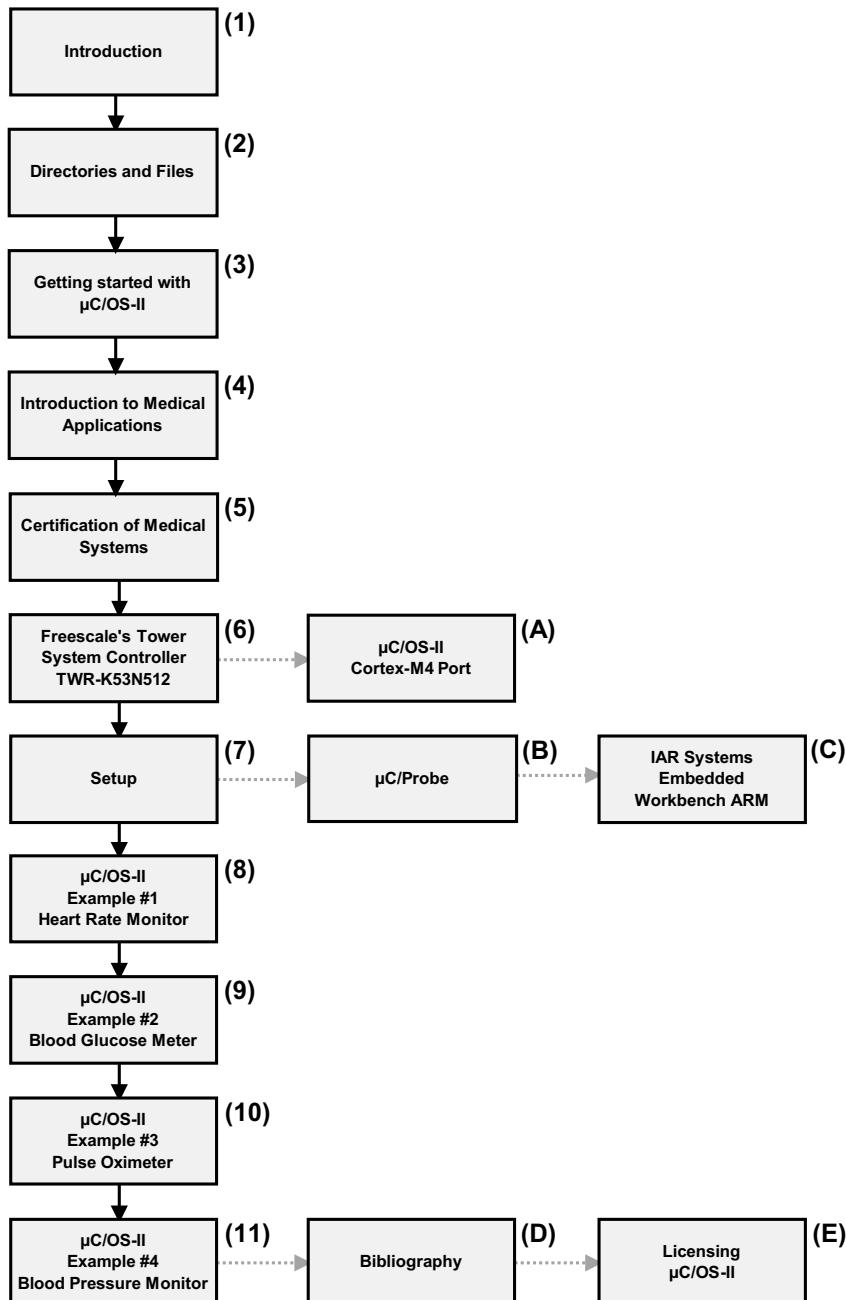


Figure 1-3 μC/OS-II Book Layout

- F1-3(1) **Chapter 1, Introduction.** This chapter.
- F1-3(2) **Chapter 2, Directories and Files.** This chapter explains the directory structure and files needed to build a µC/OS-II-based application. Here, you will learn about the files that are needed, where they should be placed, which module does what, and more.
- F1-3(3) **Chapter 3, Getting Started with µC/OS-II.** In this chapter, you will learn how to properly initialize and start a µC/OS-II-based application.
- F1-3(4) **Chapter 4, Introduction to Medical Applications.** This chapter presents the general context and main motivation for writing this book. It also presents a short description of the medical applications featured in the book.
- F1-3(5) **Chapter 5, Certification of Medical Systems.** This chapter covers general concepts that are applicable to all embedded medical systems, including the commercial development process and certification.
- F1-3(6) **Chapter 6, The Freescale Tower System and the TWR-K53N512.** This chapter provides a description of the TWR-K53N512 evaluation board that will be used for all the examples.
- F1-3(7) **Chapter 7, Setup.** This chapter explains how to set up the test environment to run µC/OS-II examples. It describes how to download the 32K Kickstart version of the IAR Systems Embedded Workbench for ARM tool chain, how to obtain example code that accompanies the book, and how to connect the TWR-K53N512 tower board to a PC.
- F1-3(8) **Chapter 8, Example #1: Heart Rate Monitor.** This chapter explains how to get µC/OS-II up and running. The project features a single channel ECG monitor that calculates the heart rate. You will also see how easy it is to use µC/Probe to display the heart rate.
- F1-3(9) **Chapter 9, Example #2: Blood Glucose Meter.** This chapter shows how to read a blood glucose test strip and display the blood glucose level using µC/Probe.

-
- F1-3(10) **Chapter 10, Example #3: Pulse Oximeter.** This chapter shows how to drive a pulse oximetry probe's LEDs and calculate the oxygen saturation and heart rate.
- F1-3(11) **Chapter 11, Example #4: Blood Pressure Monitor.** This chapter shows how to use µC/OS-II to drive an air pump and electrical valve in order to measure the blood pressure and display the value in µC/Probe.
- F1-3(12) **Appendix A, µC/OS-II Port for the Cortex-M4.** This appendix explains how µC/OS-II was adapted to the Cortex-M4 CPU. The Cortex-M4 contains interesting features specifically designed for real-time kernels, and µC/OS-II makes good use of these.
- F1-3(13) **Appendix B, µC/Probe.** This appendix provides a brief description of Micrium's award-winning µC/Probe, which easily allows users to display and change target variables at run time.
- F1-3(14) **Appendix C, IAR Systems Embedded Workbench ARM.** This appendix provides a brief description of IAR Systems Embedded Workbench for the ARM architecture (EWARM).
- F1-3(15) **Appendix D, Bibliography.**
- F1-3(16) **Appendix E, Licensing µC/OS-II.**

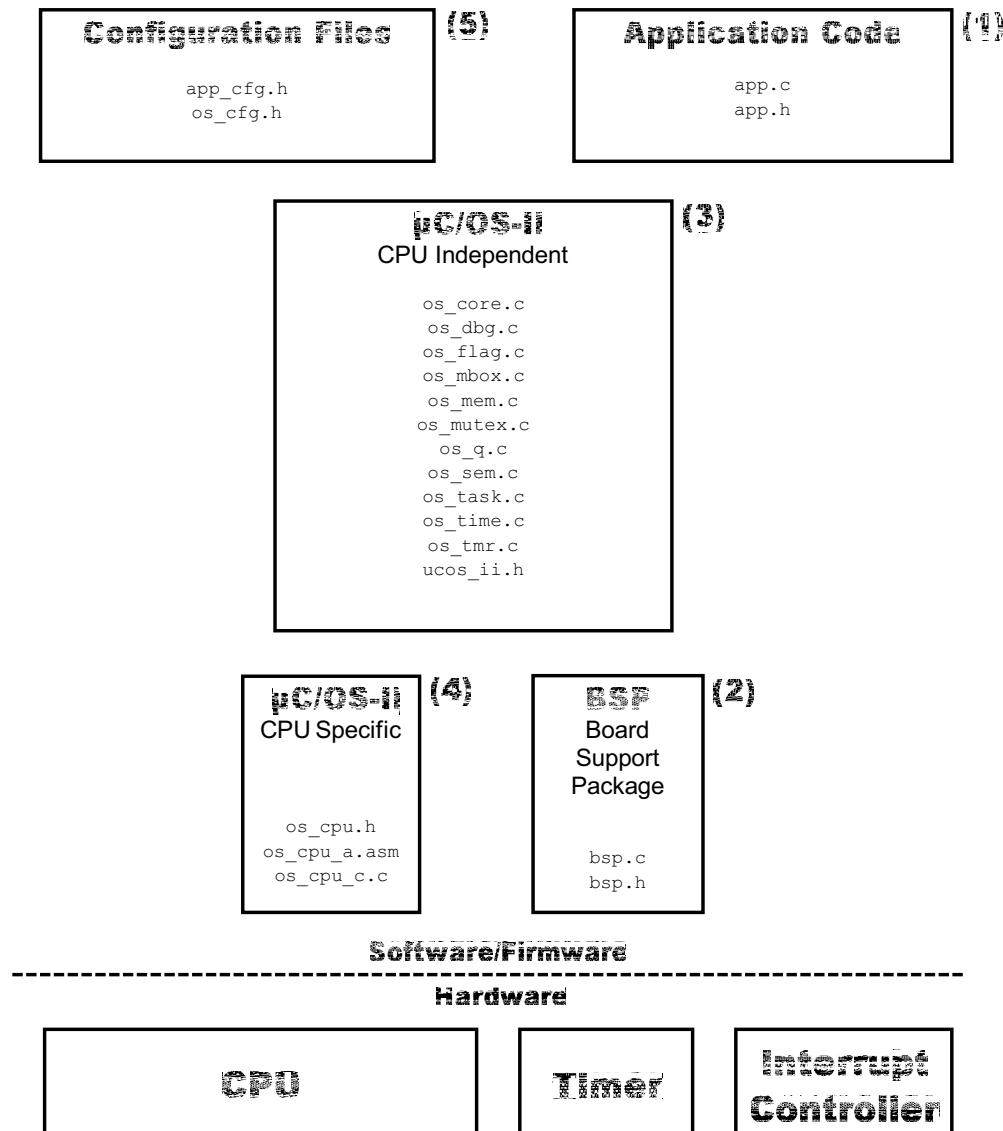
Chapter 2

Directories and Files

μ C/OS-II is fairly easy to use once it is understood exactly which source files are needed to make up a μ C/OS-II-based application. This chapter will discuss the modules available for μ C/OS-II and how everything fits together.

Figure 2-1 shows the μ C/OS-II architecture and its relationship with hardware. Of course, in addition to the timer and interrupt controller, hardware would most likely contain such other devices as Universal Asynchronous Receiver Transmitters (UARTs), Analog to Digital Converters (ADCs), Ethernet controller(s) and more.

This chapter assumes development on a Windows®-based platform and makes references to typical Windows-type directory structures (also called Folder). However, since μ C/OS-II is provided in source form, it can also be used on Unix, Linux or other development platforms.

Figure 2-1 **μC/OS-II Architecture**

- F2-1(1) The application code consists of project or product files. For convenience, these are simply called `app.c` and `app.h`, however an application can contain any number of files that do not have to be called `app.*`. The application code is typically where one would find the `main()`.

- F2-1(2) The Board Support Package (BSP) code needed by µC/OS-II is typically quite simple and generally, µC/OS-II only requires that you initialize a periodic interrupt source which is used for time delays and timeouts. This functionality can be placed in a file called `bsp.c` along with its corresponding header file, `bsp.h`.
- Semiconductor manufacturers often provide library functions in source form for accessing the peripherals on their CPU or MCU. These libraries are also part of the BSP.
- F2-1(3) This is the µC/OS-II processor-independent code. This code is written in highly portable ANSI C.
- F2-1(4) This is the µC/OS-II code that is adapted to a specific CPU architecture and is called a port.
- F2-1(5) Configuration files are used to define µC/OS-II features (`os_cfg.h`) to include in the application, specify the size of certain variables and data structures expected by µC/OS-II, such as idle task stack size and tick rate among others.

2-1 APPLICATION CODE

When Micrium provides example projects, they are placed in a directory structure shown below. Of course, a directory structure that suits a particular project/product can also be used.

```
\Micrium
  \Examples
    \<manufacturer>
      \<board_name>
        \<project name>
          \<compiler>
            \*.*
```

\Software

\Micrium

This is where we place all software components and projects provided by Micrium. This directory generally starts from the root directory of the computer.

\Software

This sub-directory contains all Micrium software components, such as the USB stack, TCP/IP stack, File System, etc.

\Examples

This sub-directory contains all projects related to evaluation boards supported by Micrium.

\<manufacturer>

This is the name of the manufacturer of the evaluation board. The “<” and “>” are not part of the actual name.

\<board name>

This is the name of the evaluation board. A board from Micrium will typically be called uC-Eval-xxxx where “xxxx” represents the CPU or MCU used on the board. The “<” and “>” are not part of the actual name.

\<project name>

The name of the project that will be demonstrated. For example, a simple µC/OS-II project might have a project name of “OS2-Ex1”. The “-Ex1” represents a project containing only µC/OS-II.

\<compiler>

This is the name of the compiler or compiler manufacturer used to build the code for the evaluation board. The “<” and “>” are not part of the actual name.

.

These are the project source files. Main files can optionally be called app*.*. This directory also contains configuration files such as os_cfg.h.

2-2 BOARD SUPPORT PACKAGE (BSP)

The Board Support Package (BSP) is generally found with the evaluation or target board as it is specific to that board. In fact, when well written, the BSP should be used for multiple projects.

```
\Micrium
  \Examples
    \<manufacturer>
```

```
\<board name>
  \BSP
    \*.*
```

\Micrium

Contains all software components and projects provided by Micrium.

\Examples

This sub-directory contains all projects related to evaluation boards.

\<manufacturer>

The name of the manufacturer of the evaluation board. The “<” and “>” are not part of the actual name.

\<board name>

The name of the evaluation board. A board from Micrium will typically be called `uC-Eval-xxxx` where “xxxx” is the name of the CPU or MCU used on the evaluation board. The “<” and “>” are not part of the actual name.

\BSP

This directory is always called BSP.

.

The source files of the BSP. Typically all of the file names start with BSP. It is therefore normal to find `bsp.c` and `bsp.h` in this directory. BSP code should contain such functions as LED control functions, initialization of timers, interface to Ethernet controllers and more.

2-3 μC/OS-II, CPU INDEPENDENT SOURCE CODE

The files in these directories are μC/OS-II processor independent files provided in source form. See Appendix E, “Licensing μC/OS-II” on page 293.

```
\Micrium
  \Software
    \uCOS-II
      \Cfg
        \Template
          \os_cfg.h
    \Source
```

```
\os_core.c  
\os_dbg.c  
\os_flag.c  
\os_mbox.c  
\os_mem.c  
\os_mutex.c  
\os_q.c  
\os_sem.c  
\os_task.c  
\os_time.c  
\os_tmr.c  
\ucos_iic  
\ucos_iic.h
```

\Micrium

Contains all software components and projects provided by Micrium.

\Software

This sub-directory contains all software components and projects.

\uCOS-II

This is the main µC/OS-II directory.

\Cfg\Template

This directory contains examples of configuration files to copy to the project directory. You will then modify these files to suit the needs of the application.

os_cfg.h specifies which features of µC/OS-II are available for an application. The file is typically copied into an application directory and edited based on which features are required from µC/OS-II. For more information consult the µC/OS-II User's Manual.

\Source

The directory containing the CPU-independent source code for µC/OS-II. All files in this directory should be included in the build. Features that are not required will be compiled out based on the value of #define constants in **os_cfg.h**.

os_core.c contains core functionality for µC/OS-II such as `OSInit()` to initialize µC/OS-II, `OSSched()` for the task level scheduler, `OSIntExit()` for the interrupt level scheduler, pend list (or wait list) management, ready list management, and more.

os_dbg.c contains declarations of constant variables used by a kernel aware debugger or μ C/Probe.

os_flag.c contains the code for event flag management.

os_mbox.c contains code for the μ C/OS-II mailbox.

os_mem.c contains code for the μ C/OS-II fixed-size memory manager.

os_mutex.c contains code to manage mutual exclusion semaphores.

os_q.c contains code to manage message queues.

os_sem.c contains code to manage semaphores used for resource management and/or synchronization.

os_task.c contains code for managing tasks using OSTaskCreate(), OSTaskDel(), OSTaskChangePrio(), and many more.

os_time.c contains code to allow a task to delay itself until some time expires.

os_tmr.c contains code to manage software timers.

ucos_i.h contains the main μ C/OS-II header file, which declares constants, macros, μ C/OS-II global variables (for use by μ C/OS-II only), function prototypes, and more.

2-4 **μ C/OS-II, CPU SPECIFIC SOURCE CODE**

The μ C/OS-II port developer provides these files.

```
\Micrium
  \Software
    \uCOS-II
      \Ports
        \<architecture>
          \<compiler>
            \os_cpu.h
            \os_cpu_a.asm
            \os_cpu_c.c
```

\Micrium

Contains all software components and projects provided by Micrium.

\Software

This sub-directory contains all software components and projects.

\uCOS-II

The main µC/OS-II directory.

\Ports

The location of port files for the CPU architecture(s) to be used.

\<architecture>

This is the name of the CPU architecture that µC/OS-II was ported to. The “<” and “>” are not part of the actual name.

\<compiler>

The name of the compiler or compiler manufacturer used to build code for the port. The “<” and “>” are not part of the actual name.

The files in this directory contain the µC/OS-II port.

os_cpu.h contains a macro declaration for `OS_TASK_SW()`, as well as the function prototypes for at least the following functions: `OSCtxSw()`, `OSIntCtxSw()` and `OSStartHighRdy()`.

os_cpu_a.asm contains the assembly language functions to implement at least the following functions: `OSCtxSw()`, `OSIntCtxSw()` and `OSStartHighRdy()`.

os_cpu_c.c contains the C code for the port specific hook functions and code to initialize the stack frame for a task when the task is created.

2-5 SUMMARY

Below is a summary of all directories and files involved in a µC/OS-II-based project. The “`<-Cfg`” on the far right indicates that these files are typically copied into the application (i.e., project) directory and edited based on the project requirements.

```
\Micrium
  \Examples
    \<manufacturer>
      \<board name>
        \<project name>
          \app.c
          \app.h
          \other
          \<compiler>

\Software
  \CPU
    \<manufacturer>
      \<architecture>
        \*.*

\uCOS-II
  \Cfg\Template
    \os_cfg.h <-Cfg
  \Source
    \os_core.c
    \os_flag.c
    \os_mbox.c
    \os_mem.c
    \os_mutex.c
    \os_q.c
    \os_sem.c
    \os_task.c
    \os_time.c
    \os_tmr.c
    \ucos_ii.h

  \Ports
    \<architecture>
      \<compiler>
        \os_cpu.h
        \os_cpu_a.asm
        \os_cpu_c.c
```


Chapter 3

Getting Started with µC/OS-II

µC/OS-II provides services to application code in the form of a set of functions that perform specific operations. µC/OS-II offers services to manage tasks, semaphores, mail boxes, mutual exclusion semaphores and more. As far as the application is concerned, it calls the µC/OS-II functions as if they were any other functions. In other words, the application now has access to a library of approximately 70 new functions.

In this chapter, the reader will appreciate how easy it is to start using µC/OS-II. Refer to the µC/OS-II user's manual available at <http://micrium.com/downloadcenter/> for the full description of several of the µC/OS-II services presented in this chapter.

It is assumed that the project setup (files and directories) is as described in the previous chapter, and that a C compiler exists for the target processor that is in use. However, this chapter makes no assumptions about the tools or the processor that is used.

3-1 SINGLE TASK APPLICATION

Listing 3-1 shows the top portion of a simple application file called `app.c`.

```
/*
***** INCLUDE FILES *****
*/
#include <app_cfg.h>                                (1)
#include <bsp.h>
#include <ucos_ii.h>
/*
***** LOCAL GLOBAL VARIABLES *****
*/
static OS_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];    (2)
/*
***** FUNCTION PROTOTYPES *****
*/
static void AppTaskStart (void *p_arg);                  (3)
```

Listing 3-1 `app.c (1st Part)`

- L3-1(1) As with any C programs, you need to include the necessary headers to build the application.

app_cfg.h is a header file that configures the application. For our example, `app_cfg.h` contains `#define` constants to establish task priorities, stack sizes, and other application specifics.

bsp.h is the header file for the Board Support Package (BSP), which defines `#defines` and function prototypes, such as `BSP_Init()`, `BSP_LED_On()`, `OS_TS_GET()` and more. The actual BSP might contain multiple header files. `bsp.h` is shown generically here.

ucos_i.h is the main header file for µC/OS-II, and includes the following header files:

`app_cfg.h`
`os_cfg.h`
`os_cpu.h`

- L3-1(2) Each task created requires its own stack. A stack must be declared using the `OS_STK` data type as shown. The stack can be allocated statically as shown here, or dynamically from the heap using `malloc()`. It should not be necessary to free the stack space, because the task should never be destroyed, and thus, the stack would always be used.
- L3-1(3) This is the function prototype of the task that we will create.

Most C applications start at `main()` as shown in Listing 3-2.

```

void main (void)
{
    INT8U err;

    BSP_IntDisAll();                                (1)

    OSInit();                                       (2)

    err = OSTaskCreateExt((void (*)(void *))AppTaskStart,
                          (void *)0,                               (3)
                          (OS_STK *)&AppTaskStartStk[APP_CFG_TASK_START_STK_SIZE - 1], (4)
                          (INT8U )APP_TASK_START_PRIO,               (5)
                          (INT16U )APP_TASK_START_PRIO,              (6)
                          (OS_STK *)&AppTaskStartStk[0],             (7)
                          (INT32U )APP_CFG_TASK_START_STK_SIZE,     (8)
                          (void *)0,                               (9)
                          (INT16U )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR));   (10)
                                                        /* Check the return error code. */

    OSTaskNameSet(APP_CFG_TASK_START_PRIO, "Start Task", &err);

    OSStart();                                      (12)

    while(DEF_ON){                                 /* Should Never Get Here. */
        };
}

```

Listing 3-2 **app.c (2nd Part)**

- L3-2(1) The startup code for the compiler will bring the CPU to `main()`. `main()` then starts by calling a BSP function that disables all interrupts. On most processors, interrupts are disabled at startup until explicitly enabled by application code. However, it is safer to turn off all peripheral interrupts during startup.
- L3-2(2) `OSInit()` is called to initialize μC/OS-II. `OSInit()` initializes internal variables and data structures, and also creates between two (2) and four (4) internal tasks. At a minimum, μC/OS-II creates the idle task (`OS_IdleTask()`), which executes when no other task is ready-to-run. μC/OS-II also creates the tick task, which is responsible for keeping track of time.

It is important to note that `OSInit()` must be called before any other µC/OS-II function.

- L3-2(3) You create a task by calling `OSTaskCreateExt()`. Most of µC/OS-II's functions return an error code via a pointer to an `INT8U` variable, `err` in this case. If `OSTaskCreateExt()` was successful, `err` will be set to `OS_ERR_NONE`. If `OSTaskCreateExt()` encounters a problem during initialization, it will return immediately upon detecting the problem and set `err` accordingly. If this occurs, look up the error code value in `ucos_i.h`.

`OSTaskCreateExt()` requires 9 arguments. The first argument is a pointer to the actual task's code. The µC/OS-II User's Manual available at <http://micrium.com/downloadcenter/> provides additional information about tasks. A typical µC/OS-II task is implemented as an infinite loop as shown:

```
void MyTask (void *p_arg)
{
    /* Do something with "p_arg".
    while (1) {
        /* Task body */
    }
}
```

The task receives an argument when it first starts. As far as the task is concerned, it looks like any other C function that can be called by the code. However, your code *must not* call `MyTask()`. The call is actually performed through µC/OS-II.

- L3-2(4) The second argument of `OSTaskCreateExt()` is the actual argument that the task receives when it first begins. In other words, the "`p_arg`" of `MyTask()`. In the example a `NULL` pointer is passed, and thus "`p_arg`" for `AppTaskStart()` will be a `NULL` pointer.

The argument passed to the task can actually be any pointer. For example, the user may pass a pointer to a data structure containing parameters for the task.

- L3-2(5) The third argument to `OSTaskCreateExt()` is a pointer to the task's top of stack. If the configuration constant `OS_STK_GROWTH` is set to 1, the stack is assumed to grow downward (i.e. from high memory to low memory). '`ptos`' will thus

point to the highest (valid) memory location of the stack. If `OS_STK_GROWTH` is set to 0, '`ptos`' will point to the lowest memory location of the stack and the stack will grow with increasing memory locations. '`ptos`' must point to a valid 'free' data item.

- L3-2(6) The next argument to `OSTaskCreateExt()` is the priority of the task. The priority establishes the relative importance of this task with respect to the other tasks in the application. A low-priority number indicates a high priority (or more important task).
- L3-2(7) The fifth argument is the task's ID, which can be a number between 0 and 65535.
- L3-2(8) The sixth argument is a pointer to the task's bottom of stack. If the configuration constant `OS_STK_GROWTH` is set to 1, the stack is assumed to grow downward (i.e. from high memory to low memory). '`pbos`' will thus point to the lowest (valid) memory location of the stack. If `OS_STK_GROWTH` is set to 0, '`pbos`' will point to the highest memory location of the stack and the stack will grow with increasing memory locations. '`pbos`' must point to a valid 'free' data item.
- L3-2(9) The seventh argument to `OSTaskCreateExt()` specifies the size of the task's stack in number of `OS_STK` elements (not bytes). For example, if you want to allocate 1 Kbyte of stack space for a task and the `OS_STK` is a 32-bit word, then you need to pass 256.
- L3-2(10) The eight argument is a pointer to a user supplied memory location which is used as a TCB extension. For example, this user memory can hold the contents of floating-point registers during a context switch, the time each task takes to execute, the number of times the task has been switched-in, etc.
- L3-2(11) The last argument of `OSTaskCreateExt()` contains additional information (or options) about the behavior of the task. The lower 8-bits are reserved by `μC/OS-II` while the upper 8 bits can be application specific. Current choices are:
- `OS_TASK_OPT_STK_CHK` Stack checking to be allowed for the task
 - `OS_TASK_OPT_STK_CLR` Clear the stack when the task is created

- OS_TASK_OPT_SAVE_FP If the CPU has floating-point registers, save them during a context switch.
- L3-2(12) The next step in main() to create the task is to call `OSTaskNameSet()` to set the name of the task.
- L3-2(13) The final step in `main()` is to call `OSStart()`, which starts the multitasking process. Specifically, µC/OS-II will select the highest-priority task that was created before calling `OSStart()`.

A few important points are worth noting. For one thing, you can create as many tasks as you want before calling `OSStart()`. However, it is recommended to only create one task as shown in the example because, having a single application task allows µC/OS-II to determine the relative speed of the CPU. This allows µC/OS-II to determine the percentage of CPU usage at run-time. Also, if the application needs other kernel objects such as semaphores and message queues then it is recommended that these be created prior to calling `OSStart()`. Finally, notice that interrupts are not enabled. This will be discussed next by examining the contents of `AppTaskStart()`, which is shown in Listing 3-3.

```
static void AppTaskStart (void *p_arg) (1)
{
    p_arg = p_arg;
    BSP_Init(); (2)
    CPU_Init(); (3)
    OS_BSP_TickInit(); (4)
    BSP_LED_Off(0); (5)
    while (1) { (6)
        BSP_LED_Toggle(0); (7)
        OSTimeDlyHMSM((INT16U) 0, (INT16U) 0, (INT16U) 0, (INT32U)100); (8)
    }
}
```

Listing 3-3 app.c (3rd Part)

- L3-3(1) As previously mentioned, a task looks like any other C function. The argument “`p_arg`” is passed to `AppTaskStart()` by `OSTaskCreate()`, as discussed in the previous listing description.

- L3-3(2) `BSP_Init()` is a Board Support Package (BSP) function that is responsible for initializing the hardware on an evaluation or target board. The evaluation board might have General Purpose Input Output (GPIO) lines that might need to be configured, relays, sensors and more. This functionality is found in a file called `bsp.c`.
- L3-3(3) `CPU_Init()` initializes the µC/CPU services. µC/CPU provides services to measure interrupt latency, obtain time stamps, and provides emulation of the count leading zeros instruction if the processor used does not have that instruction, and more.
- L3-3(4) `OS_CPU_SysTickInit()` sets up the µC/OS-II tick interrupt. For this, the function needs to initialize one of the hardware timers to interrupt the CPU at a rate of: `OS_TICKS_PER_SEC`, which is defined in `os_cfg.h`.
- L3-3(5) `BSP_LED_Off()` is a function that will turn off all LEDs. `BSP_LED_Off()` is written such that a zero argument means all the LEDs.
- L3-3(6) Most µC/OS-II tasks will need to be written as an infinite loop.
- L3-3(7) This BSP function toggles the state of the specified LED. Again, a zero indicates that all the LEDs should be toggled on the evaluation board. You simply change the zero to 1 and this will cause LED #1 to toggle. Exactly which LED is LED #1? That depends on the BSP developer. Specifically, access to LEDs are encapsulated through the functions: `BSP_LED_On()`, `BSP_LED_Off()` and `BSP_LED_Toggle()`. Also, for sake of portability, we prefer to assign LEDs logical values (1, 2, 3, etc.) instead of specifying which port and which bit on each port.
- L3-3(8) Finally, each task in the application must call one of the µC/OS-II functions that will cause the task to “wait for an event.” The task can wait for time to expire (by calling `OSTimeDly()`, or `OSTimeDlyHMSM()`), or wait for a signal or a message from an ISR or another task. In the code shown, we used `OSTimeDlyHMSM()` which allows a task to be suspended until the specified number of hours, minutes, seconds and milliseconds have expired. In this case, 100 ms. the µC/OS-II User’s Manual available at <http://micrium.com/downloadcenter/> provides additional information about time delays.

3-2 MULTIPLE APPLICATION TASKS WITH KERNEL OBJECTS

The code of Listing 3-4 through Listing 3-8 shows a more complete example and contains three tasks: a mutual exclusion, semaphore, and a message queue.

```
/*
***** INCLUDE FILES *****
*/
#include <app_cfg.h>
#include <bsp.h>
#include <ucos_ii.h>
/*
***** LOCAL GLOBAL VARIABLES *****
*/
static OS_EVENT *AppMutex; (1)
static OS_EVENT *AppQ;
static void *AppQ_Tbl[10];
static OS_STK AppTaskStartStk[APP_CFG_TASK_START_STK_SIZE]; (2)
static OS_STK AppTask1_Stk[128]; (3)
static OS_STK AppTask2_Stk[128];
/*
***** FUNCTION PROTOTYPES *****
*/
static void AppTaskStart (void *p_arg); (4)
static void AppTask1 (void *p_arg);
static void AppTask2 (void *p_arg);
```

Listing 3-4 app.c (1st Part)

- L3-4(1) A mutual exclusion semaphore (a.k.a. a mutex) is a kernel object (a data structure) that is used to protect a shared resource from being accessed by more than one task. A task that wants to access the shared resource must obtain the mutex before it is allowed to proceed. The owner of the resource relinquishes the mutex when it has finished accessing the resource. This process is demonstrated in this example.

-
- L3-4(2) A message queue is a kernel object through which Interrupt Service Routines (ISRs) and/or tasks send messages to other tasks. The sender “formulates” a message and sends it to the message queue. The task(s) wanting to receive these messages wait on the message queue for messages to arrive. If there are already messages in the message queue, the receiver immediately retrieves those messages. If there are no messages waiting in the message queue, then the receiver will be placed in a wait list associated with the message queue. This process will be demonstrated in this example.
 - L3-4(3) A stack is allocated for each task.
 - L3-4(4) The prototype of the tasks are declared.

Listing 3-5 shows the C entry point, i.e. `main()`.

```

void main (void)
{
    INT8U err;

    CPU_Init();
    BSP_IntDisAll();
    OSInit();
    /* Check for 'err' */

    AppMutex = OSMutexCreate(9, &err);                                (1)
    OSEventNameSet(AppMutex, "My Mutex", &err);
    /* Check for 'err' */

    AppTaskQ_DAQ = OSQCreate(&AppTaskQ_DAQ_Tbl[0], APP_MSG_MAX);      (3)
    OSEventNameSet(AppTaskQ_DAQ, "DAQ Task Q", &err);
    /* Check for 'err' */

    err = OSTaskCreateExt((void (*)(void *)) AppTaskStart,
                          (void *) 0u,
                          (OS_STK *) &AppTaskStartStk[APP_CFG_TASK_START_STK_SIZE - 1],
                          (INT8U ) APP_CFG_TASK_START_PRIO,
                          (INT16U ) APP_CFG_TASK_START_PRIO,
                          (OS_STK *) &AppTaskStartStk[0u],
                          (INT32U ) APP_CFG_TASK_START_STK_SIZE,
                          (void *) 0u,
                          (INT16U ) (OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR));
    /* Check for 'err' */

    OSTaskNameSet(APP_CFG_TASK_START_PRIO, "Startup Task", &err);
    /* Check for 'err' */

    OSStart();
}

```

Listing 3-5 app.c (2nd Part)

- L3-5(1) `OSMutexCreate()` is used to create and initialize a mutex. The first argument is the priority inheritance priority (PIP) that is used when a high priority task attempts to acquire the mutex that is owned by a low priority task. In this case, the priority of the low priority task is raised to 9 until the resource is released. The µC/OS-II User's Manual available at <http://micrium.com/downloadcenter/> provides additional information about mutual exclusion semaphores.

You can assign an ASCII name to the mutex, which is useful when debugging.

- L3-5(2) Before you create a message queue, you need to declare a message storage area as an array of pointers to voids. Then you create the message queue by calling `OSQCreate()` and specify the base address and size of the of the message storage area. The `μC/OS-II` User's Manual available for download at <http://micrium.com/downloadcenter/> provides additional information about message queues.

You can assign an ASCII name to the message queue which can also be useful during debugging.

- L3-5(3) The first application task is created.

Listing 3-6 shows how to create other tasks once multitasking as started.

```

static void AppTaskStart (void *p_arg)
{
    INT8U err;

    p_arg = p_arg;
    BSP_Init();
    CPU_Init();
    OS_BSP_TickInit();

    err = OSTaskCreateExt((void (*)(void *)) AppTask1,
                          (void *) 0u,
                          (OS_STK *)&AppTask1_Stk[127],
                          (INT8U) 11,
                          (INT16U) 11,
                          (OS_STK *)&AppTask1_Stk[0u],
                          (INT32U) 128,
                          (void *) 0u,
                          (INT16U) (OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR));    (1)

    /* Check for 'err' */

    OSTaskNameSet(11, "Task 1", &err);
    /* Check for 'err' */

    err = OSTaskCreateExt((void (*)(void *)) AppTask2,
                          (void *) 0u,
                          (OS_STK *)&AppTask2_Stk[127],
                          (INT8U) 12,
                          (INT16U) 12,
                          (OS_STK *)&AppTask2_Stk[0u],
                          (INT32U) 128,
                          (void *) 0u,
                          (INT16U) (OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR));    (2)

    /* Check for 'err' */

    OSTaskNameSet(12, "Task 2", &err);
    /* Check for 'err' */

    BSP_LED_Off(0);

    while (1) {
        BSP_LED_Toggle(0);
        OSTimeDlyHMSM(0, 0, 0, 100);
    }
}

```

Listing 3-6 app.c (3rd Part)

- L3-6(1) Task #1 is created by calling `OSTaskCreateExt()`. If this task happens to have a higher priority than the task that creates it, μC/OS-II will immediately start Task #1. If the created task has a lower priority, `OSTaskCreateEx()` will return to `AppTaskStart()` and continue execution.
- L3-6(2) Task #2 is created and if it has a higher priority than `AppTaskStart()`, μC/OS-II will immediately switch to that task.

```

static void AppTask1 (void *p_arg)
{
    INT8U err;

    while (1) {
        OSTimeDly(1);                                (1)

        OSQPost(AppQ, 1);                            (2)

        OSMutexPend(AppMutex, 0, &err);             (3)

        /* Access shared resource */

        OSMutexPost(AppMutex);                      (4)

    }
}

```

Listing 3-7 app.c (4th Part)

- L3-7(1) The task starts by waiting for one tick to expire before it does anything useful. If the μC/OS-II tick rate is configured for 1000 Hz, the task will be suspended for 1 millisecond.
- L3-7(2) The task then sends a message to another task using the message queue `AppQ`. In this case, the example sends a fixed message of value “1,” but the message could have consisted of the address of a buffer, the address of a function, or whatever would need to be sent.

- L3-7(3) The task then waits on the mutual exclusion semaphore since it needs to access a shared resource with another task. If the resource is already owned by another task, AppTask1() will wait forever for the mutex to be released by its current owner. The forever wait is specified by passing 0 as the second argument of the call.
- L3-7(4) When OSMutexPend() returns, the task owns the resource and can therefore access the shared resource. The shared resource may be a variable, an array, a data structure, an I/O device, etc. You should note that we didn't actually show the access to the shared resource. This is not relevant at this point.
- L3-7(5) When the task is done with the shared resource, it must call OSMutexPost() to release the mutex.

```

static void AppTask2 (void *p_arg)
{
    INT8U err;
    void *p_msg;

    while (1) {
        p_msg = OSQPend(AppQ, 0, &err); (1)

        /* Process message received */
        } (2)
    }
}

```

Listing 3-8 app.c (5th Part)

- L3-8(1) Task #2 starts by waiting for messages to be sent through the message queue AppQ. The task waits forever for a message to be received because the second argument specifies an infinite timeout.

When the message is received p_msg will contain the message (i.e., a pointer to “something”). In our case, AppTask2() will always receive a message value of ‘1’. Both the sender and receiver must agree as to the meaning of the message.

- L3-8(2) Here you would add your own code to process the received message.

Chapter 4

Introduction to Medical Applications

A large percentage of the U.S. population, the baby boomers, is approaching retirement age. They are more health conscious than any previous generation, tend to continue to work longer and have substantial disposable income. They are used to demanding products and features that improve the quality of life, and most are extremely comfortable in the personal use of digital devices. In fact, a repetitive stress called “Blackberry Thumb” is no stranger to this demographic.

While the conditions on the demand side appear ripe to support a large and growing market, what about the supply side?

For healthcare providers, things do not at first blush look as rosy. Dramatic increases in the cost of healthcare means continued cuts in services. Further reductions in the average length of hospital stays, and even in the ability to offer expensive testing, are expected.

Furthermore, other factors beyond the scope of this book are driving healthcare personnel shortages. No matter the place, whether it is a large medical center in a developed country or a rural community hospital in a third world country, being a patient involves a lot of waiting around. Taking aside the time spent dealing with the bureaucracy of scheduling appointments, filling out the forms and paying the bills, most of the time is spent waiting for the limited number of healthcare personnel that can take care of you.

Unless the economics of healthcare change dramatically, the trend towards outpatient care will drive an increase in remote care options. Battery, sensor, processing and wireless technology advances will support the move from hands-on and in-person testing to remote test and monitoring. And, in some remote regions of the world, it may be the only way to provide reasonable care.

Home medical devices will play a major role as remote care catches on. And, as the evolution takes place, embedded software accuracy and accountability become even more critical in nature.

One of the main obstacles standing in the way of the development of home medical devices is the cost associated with the rigorous processes and practices mandated in safety-critical development that the medical devices manufacturers have to face in order to meet the certification needs for their products.

This chapter describes with more detail the medical device marketplace and the conditions and factors behind its rapid growth. It also puts emphasis into the regulatory environment, especially in the U.S. and the European Union, and makes emphasis on the embedded software that provides safety-critical assurance against device failure, and the processes and regulatory criteria involved in the software's development, verification and validation.

Micrium's µC/OS-II and several of its RTOS modules have been deployed in many medical designs and µC/OS-II has 100% of the required documentation needed to comply with the FDA 510(k)/Pre-market Approval (PMA), and also complies with IEC 62304, IEC 60601 and ISO 14971.

This book demonstrates how the Freescale's Tower System and Micrium's µC/OS-II can be used as the cornerstone to build a home medical device that relies on a combination of proven hardware and software platforms. µC/OS-II is a certifiable real-time kernel that has gone through a very rigorous validation process. Freescale's ultra-low-power Kinetis MCUs are the core of the TWR-K53N512 that integrates precise analog components such as operational amplifiers, transimpedance amplifiers, high resolution ADCs and DACs specifically designed for low-power portable medical devices used in telemedicine.

The examples in this book include home medical devices such as a *Heart Rate Monitor*, *Blood Glucose Meter*, *Pulse Oximeter* and a *Blood Pressure Monitor*.

All four examples are based on the following key elements as shown in Figure 4-1:

- Freescale's Tower System controller TWR-K53N512 (Kinetis ARM® Cortex™-M4)
- Freescale's Medical Analog-Front-End modules: MED-EKG, MED-GLU, MED-SPO2 and MED-BPM
- IAR Systems Embedded Workbench for ARM (EWARM)

-
- Segger J-Link for ARM Processors
 - Micrium's µC/OS-II
 - Micrium's µC/Probe

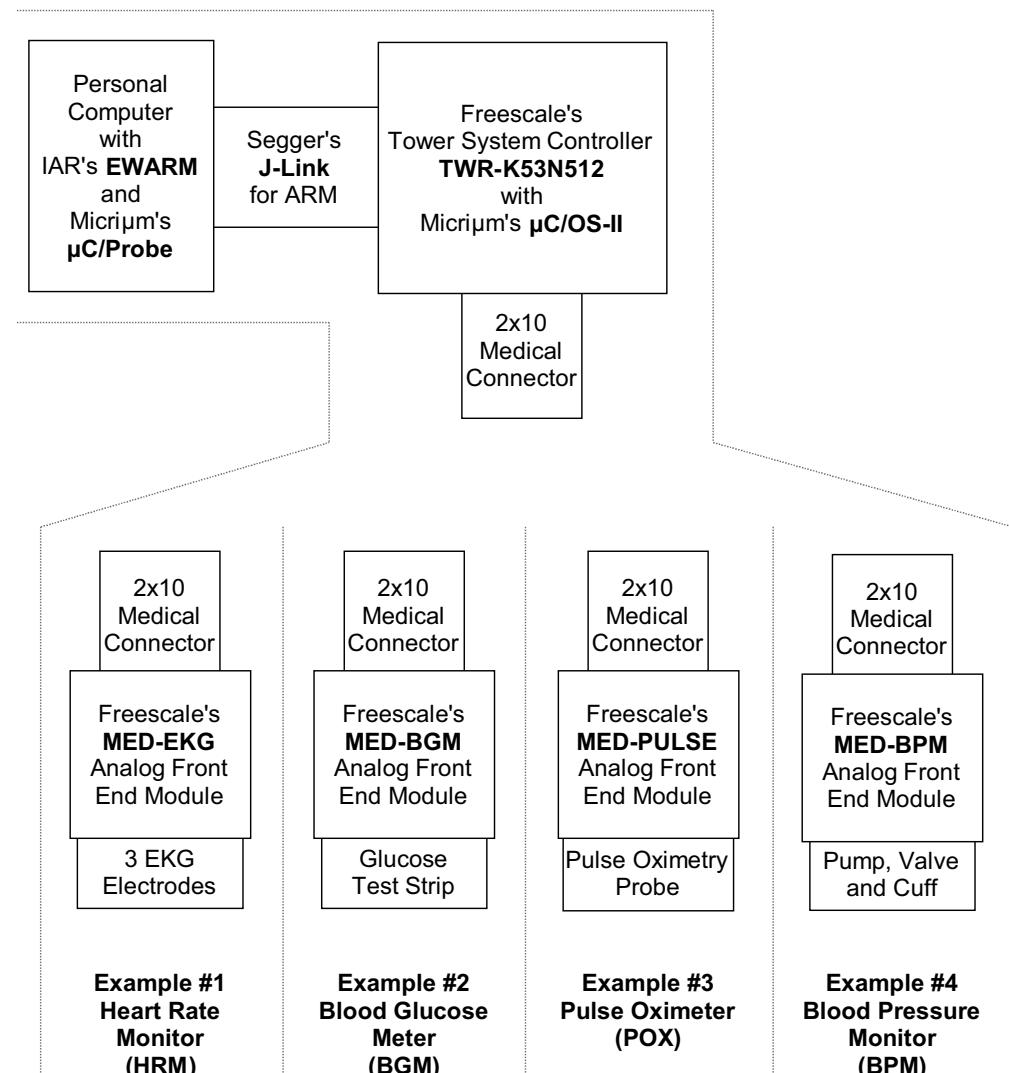
The TWR-K53N512 is available from Freescale and is a Controller Module compatible with their Tower System that features the Kinetis K53 low-power microcontroller based on the ARM® Cortex™ M4 architecture. Chapter 6, “Freescale’s Tower System Controller TWR-K53N512” on page 81 will describe the main details of this controller as it will be used in all four examples.

In order to build the example code, you must download the IAR Systems Embedded Workbench for ARM from the IAR website.

You can download an evaluation version of µC/Probe from the Micrium website in order to not only monitor the application variables at run time but also to prototype the front panel of each medical device. The source code of the examples is available for free from the same website.

Each example requires an Analog-Front-End (AFE) module that connects to the TWR-K53N512 controller via a 2x10 medical connector if you want to run the examples live with the signals coming from a human being. The AFEs contain all the circuits necessary to interface with standard medical supplies and the signal conditioning circuitry.

In the absence of these AFEs, all the examples feature a separate simulation task that generates the necessary analog signals. This type of built-in signal generator is typically used for calibration and testing in the medical industry.

Figure 4-1 **Medical Application Examples Overview**

This book and Freescale's TWR-K53N512 make an excellent platform for hardware engineers interested in testing their own medical signal conditioning circuits as they do not have to worry about the software details. Freescale offers prototyping boards compatible with their tower system for such purpose. Software engineers interested in testing their own medical signal analysis algorithms will find this book and platform very useful too as they do not have to worry about the hardware necessary to acquire the signals. Engineers and scientists in general and those not familiar with Biomedical Engineering will find this book

very interesting as they will learn some of the inner works of certainly the most advanced control system in the planet called the human body. An illustrated explanation of some of the main regulation mechanisms including the glucose level and blood pressure in the body will serve not only as inspiration for the design of any control system but also as an introduction to each example.

The medical application examples in this book are provided solely as a reference to help engineers use Micrium and Freescale Semiconductor products.

There are no express or implied copyright licenses granted hereunder to design or fabricate any medical devices based on the information in this book.

Micrium and Freescale Semiconductor make no warranty, representation or guarantee regarding the suitability of these examples for any particular purpose, nor does Micrium and Freescale Semiconductor assume any liability arising out of the application or use of any example design, and specifically disclaims any and all liability, including without limitation consequential or incidental damages.

The following medical application examples are not designed, intended, or authorized for use as components in systems intended to support or sustain life, or for any other application in which the failure of the system could create a situation where personal injury or death may occur.

You should always consult your physician or other healthcare professionals for specific advice regarding any medical or health condition.

Chapter

5

Certification of Medical Systems

For the purpose of this chapter, a medical device is defined as any electronic device that uses software to perform its intended purpose within the health-care industry; the actual product might even be “just” software.

A 2007 report titled “Future Trends in Medical Device Technologies” by Wm. A. Herman and Gilbert B. Devey [1] summarized the work of a cross functional work-group consisting of participants from the US Food and Drugs Administration (FDA), medical industry, think-tanks and academia. The group was asked to predict the direction the medical industry would take over the next 10 years. It is no surprise that many predictions were not only based on their areas of personal expertise, but also on technology convergence already seen in such commercial sectors as cellular communication, gaming, and IT.

The work-group placed a high degree of confidence that the 10-year mark would see home and tele healthcare firmly entrenched based on the assumption that a supporting host of sensor, monitors and remote devices will be sufficiently advanced to support these roles.

Other predictions include:

Devices providing full time monitoring, advanced detection, therapy, and active intervention will replace expensive tests. While many of those examples would likely be implemented in something beyond the normal embedded real-time devices, the survey included devices based on smart sensors, monitors, glucose-monitoring devices and drug delivery systems as well.

The training of doctors will benefit from Internet-based medical databases and services. Virtual reality, robotics, and computer-aided diagnostics were all mentioned as technologies that will continue to grow over the next decade.

Software needed to make these devices operable can be seen in common smartphones and such other electronics as remote control, Bluetooth, Zigbee, and other wireless protocols. Fault tolerant file system or embedded databases would be the basis for delivery of powerful chemotherapy drugs, or even sending GPS coordinates to family members in an emergency.

Let's look closer at this subject of software to be able to see its impact on medical devices, and the important choices that must be made by designers.

5-1 “OFF-THE-SHELF” SOFTWARE

Designers with experience in embedded device software know that software begets yet more software. Each line of code that enables a new “visible” feature requires 10 to 100 lines of code to support it. Let's take, for example, a new feature addition to an existing defibrillator design. The goal is to record all information that is captured from the patient during a cardiac event, the defibrillator's intervention, and post-intervention data and store it for real-time and future use.

This is not an unreasonable goal. All of this information is currently available such as information the defibrillator gathers and uses to determine the optimal intensity, frequency and duration of the electric shock it administers. However, in the past it used a simple memory file system to capture and analyze the cardiac data. Now it must convert the data to a common file system such as FAT32. Amazingly, adding what seems to be a trivial feature enhancement requires tens of thousands of lines of code.

Upgrades will not stop with a file system. Soon, a touch screen and 2-D and 3-D graphics will also be expected. In the not-too-distant future, the system should be tied into the health-care IT system using wireless communication. Now, with that level of system exposure, patient privacy and data concerns require even more software be added to protect the software that was just added. We'll look again at this concept of cost per line of code.

For even the largest companies staffing this kind of expertise is not only expensive but it can undermine the company's value-add focus. Smaller companies and startups have no choice but to turn to third party “off-the-shelf” (OTS) software.

For many consumer products such as cell phones, third party software is easily used. That is not true in medical applications. OTS software is not easy to navigate and the not-invented-here (NIH) syndrome is strong in this industry, especially given the potential for litigation. While medical device manufacturers do use third-party software, it is not always developed with their specific needs in mind. Perhaps that is a contributor to the rapid increase in recalls, as discussed in the next section.

5-2 MEDICAL DEVICE SOFTWARE – RECALLS

According to an FDA analysis of 3140 medical devices conducted between 1992 and 1998, 7.7% of them were recalled due to faulty software. Of those software related recalls, 79% were attributable to bugs introduced after the product's initial release. The FDA reports that within the subsequent years ending in 2005, nearly one in three of all medical devices containing software were recalled [2].

Just three years later, in an April 29th, 2008 presentation titled “CDRH Software Update,” John F Murray Jr., an FDA software compliance expert, reported that faulty software accounted for 18% of all recalled devices. In a period of 10 years, the rate of software related recalls increased a whopping 133%.

This suggests that if the growth rate of recalls remains static, in 10 years, faulty software will account for over 40% of recalls within this industry.

The growing rate of recalls is a clear indication that the current regulatory process is inadequate. The fact that two thirds [3] of bugs found in the recalled medical device were introduced post release may be one reason that as of March 2010 [1], all but the most minor of medical devices sold within the EU must now comply with a more rigorous medical software standard known as IEC 62304.

U.S. manufacturers, along with international and domestic stakeholders participated in the creation of the international standard even though it is not adopted by the FDA. Today, IEC 62304 is the most cost effective approach in meeting the certification needs of the global market.

Before looking at the EU and U.S. regulatory climate, let's begin to look at just what safety critical means in a medical context.

5-3 SAFETY CRITICAL

Most medical software falls into a special sub-category called safety-critical. The operating environment, operators, patients, and electro-mechanical portion of the medical device comprise a safety-critical system. The failure or improper operation of such system may allow or cause:

- Injury or loss of life to a human or animal
- Environmental damage
- Damage or loss of capital equipment

The primary focus of medical software pertains to the safety of patients, operators and staff. Typical safety-critical software life cycle tasks include:

- Planning
- Requirements
- Design
- Coding and Integration
- Testing and Verification
- Configuration Management (CM)
- Quality Assurance
- Post Release Maintenance

Many of the activities that take place within software development can be likened to hardware redundancies that make up critical hardware systems. The redundant practices i.e. code reviews; traceability, code coverage, etc. remove single point failures and, while costly, are less expensive than fixing failures after the fact.

For manufacturers who have product families or products that share code, not only can code be reused but also, all of the shared artifacts (artifacts in this case include everything concerned with certification) can as well. In fact, because of the additional processes and practices mandated in safety-critical development the relative savings gained through code reuse easily exceeds that of commercial software development.

There are several areas where the pain-points of consumer electronics match those of medical device development. Scheduling, time-to-market pressures, and sufficient staff to develop and maintain software such as embedded, real-time operating system (RTOS) kernels, networking stacks, and file systems, are all examples. In many situations, it is more cost effective to leave the development and maintenance to the experts and license software as needed.

When properly vetted, the use of commercial software can be a wise choice that speeds development, improves quality (or at least does not decrease it), reduces overall development costs and reduces the stress placed on development teams allowing them to stay focused on achieving the core goals of the project rather than developing commodity software.

Off-the-shelf software intended for general-purpose devices is not the same as software that has been developed, verified and validated for use in safety-critical devices. A medical device manufacturer using OTS software generally gives up software life cycle control, but still bears the responsibility for the continued safe and effective performance of the medical device.

The manufacturer has two choices when considering the use of OTS software in their design. One is to purchase software that despite the OTS label, is designed, verified, validated and comes with the same documentation that is expected by the FDA or other certification agency. Micrium's uC/OS and several of its RTOS components fit this category. It has been deployed in many medical designs, and has 100% of the required documentation needed to comply with the FDA 510(k)/Pre-market Approval (PMA), and also complies with IEC 62304, IEC 60601, and ISO 14971.

The second choice involves using "Software Of Unknown Provenance" (SOUP) [4]. Using this option is at first appealing, as it appears to be much less expensive than properly supported software, and in many cases, it offers great features. There are also inherent negatives. The primary negative is the code is not yet properly verified, validated and documented. The result is often a more expensive route. The References page has several links to FDA documents that provide some guidance for SOUP. One particular valuable link

is: Guidance for Industry, FDA Reviewers and Compliance on Off-the-Shelf Software Use in Medical Devices, Office of Device Evaluation, Center for Devices and Radiological Health, Food and Drug Administration, September 1999.

<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm073779.pdf>

5-4 TRACEABILITY

The concept of traceability is used and implemented in many ways. Each requirement has an explicit attribute of traceability to its source/origin. It may evolve from a general requirement, result from a conversation with a user, result from adoption of a standard, or adhering to a new regulation. However, each requirement is present for a reason, and that reason is said to be an attribute contained within the requirement. A natural extension of this idea is to add a traceable attribute that points to the implementation and validation of the requirement, essentially documenting the path from a requirement origin thru implementation and its validation.

When traceability is not pursued with vigor, it can cause added project expense and missed schedules.

When used properly, and its role expanded to encompass and include the full gamut of software development activities including requirements specification document creation, software architecture design, detailed design, verification, validation, test and quality assurance (QA), technical publications, maintenance and other post-release activities, and software reuse, etc, it provides insight into every aspect of the project life cycle.

The concept of traceability as originally intended for validation has rapidly expanded. Some companies consider it to be a core component of the Scope of Work (SOW). It provides a way to manage and estimate the cost of changes to project requirements. Traceability is used instead of prototyping to prove understanding and to communicate to clients the nature of the design.

5-5 COST OF SAFETY CRITICAL SOFTWARE

Most developers acquainted with safety-critical software will tell you that it is anything but cheap.

Several figures are thrown out, but a rule of thumb is that commercial-grade software runs from \$15 to \$30 per line of code and that safety-critical software generally costs five to ten times that amount. The range of \$75 to \$300 per source line of software is not unrealistic. However, given that almost 20% of all recalled devices are due to software faults, adds a significant amount to that total [3] given the time to recall, fix and redistribute the solution.

Other factors to consider include:

Many software developers still follow an 80/20 rule whereby 80% of the cost is in code and debug, while 20% is attributable to design.

Results from a 2002 study indicate that 1/3 of the cost associated with faulty software could be eliminated with proactive processes [5].

Up to \$59 billion (2002) of waste is attributable to faulty software, and over half of that amount is borne by users [5].

- What is the impact of when a fault is introduced vs. when it is found?
- What is the total cost of a recall?
- What is the cost of civil litigation?
- When is a bug introduced vs. when the bug is found?

Safety-critical software standards represent a rational compromise of social and market forces. On one side is stakeholder safety, and on the other, positive economic return for investors.

There is no such thing as bug free software, as well as there is a point of investment in quality at which there will be no positive return on the investment in a safety critical device. The quality of software can be determined by comparing its characteristics (features, capabilities, behavior) with the set of requirements that govern its creation. If its

characteristics satisfy the set of requirements completely, high quality is achieved. On the other hand, if its characteristics do not meet the set of requirements, low quality results. Success in documenting software requirements is a crucial factor in the successful validation of the resulting software code [6].

Requirements are often described in terms of a hierarchy of requirements i.e. high-level, and low-level.

An example of this could be system requirements that are composed of sets of hardware requirements, and software requirements. The software requirements can be broken down into even more sets of software that represent RTOS requirements, and application software requirements. This process continues until it can not be decomposed any further yet still have the attribute that it is actionable and it is measurable. Verification and validation each play an essential role in any discussion of software quality.

The terms verification and validation are often used interchangeably. Some even go as far as to use verification, validation and testing as if they mean the same thing. It is important to understand the difference of the two terms as they represent fundamental concepts in the safety-critical development processes. According to the FDA:

Verification addresses the question: “Are we making the software correctly?”

“The objective of software verification is to perform, create and document with objective evidence that the design outputs of each phase of the software development life cycle meets all of the specified requirements for that phase. Software verification looks for consistency, completeness, and correctness of the software and its supporting documentation, as it is being developed, and provides support for a subsequent conclusion that software is validated. Software testing is one of many verification activities intended to confirm that software development output meets its input requirements. Other verification activities include various static and dynamic analyses, code and document inspections, walk-through, and other techniques.”

In comparison, validation addresses the question: “Are we making the correct software?”

The FDA considers software validation to be “confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled.”

5-6 MEDICAL DEVICE MARKET REGULATORY ENVIRONMENT

There are two categories of rules and regulations within the regulatory environment for medical devices: technical and non-technical. Development standards and the product cycle fall within the first category, and non-technical governance falls into the second category. The players within the medical market may be active in two or more of the following three groups, but generally take a primary role in one:

- Manufacturers – all stakeholders playing a development, manufacturing, and sales role
- Regulators – Government or Non-governmental groups responsible for compliance
- Industry and Standards Bodies – International and regional groups with goals of protection and promotion of their industry and clientele

5-6-1 THE REGULATORS

The laws, regulatory burden and constraints that apply to a domestic manufacturer of medical equipment require a substantial time and expense commitment. Manufacturers that market and sell their products internationally face a vast, convoluted, and complex morass of global laws, regulations and statutes.

Each nation has its own agency that controls its medical market. In the U.S., Congress confers power to the FDA. In the European Union, the European Parliament and the Council of the EU, empower the Competent Authorities of member states with the responsibility. Such countries as Japan, Canada, Mexico, and Great Britain have a similar approach to regulation.

5-6-2 INDUSTRY AND STANDARDS BODIES

The voluntary association of members and stakeholders from within the medical market create medical standards. While federated employees also participate in the standards groups, standards represent a consensus on best practices that will benefit the industry as a whole. The goals are focused on improving the overall industry.

There are three primary international standardization organizations for medical devices i.e. the International Organization for Standardization (ISO), the International Electrotechnical Commission (IEC), and the International Telecommunication Union (ITU). Generally, ITU

covers telecommunications, IEC covers electrical and electronic engineering, and ISO covers the remainder.

The Institute of Electrical and Electronics Engineers (IEEE), and Association for the Advancement of Medical Instrumentation (AAMI) also play a role in standards and guidance. The Global Harmonization Task Force (GHTF) is dedicated to harmonizing the different technical standards to achieve uniformity worldwide.

The next section looks at the world's largest regulatory environments, the U.S. and the EU.

5-6-3 THE REGULATORY ENVIRONMENT IN THE UNITED STATES

The U.S. FDA is responsible for interpreting Congressional statutes provided for in Title 21 of the Code of Federal Regulations (CFR). Title 21 regulates food, drugs and medical devices within the U.S., and defines the regulatory authority of the U.S. Food and Drug Administration (FDA), the Drug Enforcement Administration (DEA), and the Office of National Drug Control Policy. The regulations that apply strictly to the FDA are contained in 21 CFR Parts 1 to 1499 [7]. The Center for Devices and Radiological Health (CDRH), a branch of the FDA, is responsible for the premarket approval of all medical devices, and oversees the manufacturing, performance and safety of these devices.

Either the 510(k) or the Pre-market Approval (PMA) process governs FDA/CDRH compliant devices, depending on their classification. Determining which application process to use is not complex, but it does seem to be somewhat arbitrary. Therefore, it is recommended that prior to making any assumptions about the device type, class or application process that the following determinations are made for a medical device:

- The degree of development rigor placed on the manufacturer is dependent on a number of factors including the intended use of the device and predicate device classification play a large role in establishing the class of the device. The FDA allows a great deal of latitude in establishing predicated use. Attempts should be made to either reduce or eliminate the degree of regulatory controls placed on the device in question.
- Work closely with the FDA or a qualified consultant throughout the process to avoid mistakes that require restarting the process.

5-6-4 FDA 510(K)

While one often hears the term “Certification” and “FDA 510(k)/PMA” used in the same sentence, in reality the 510(k) is not really a standard. It is the section of the Federal Food, Drug and Cosmetic Act (FD&CA) that defines how medical devices requiring FDA review qualify to be sold in the US market. The FDA 510(k) process is used to obtain marketing clearance for a device that is substantially equivalent in safety and effectiveness to another lawfully marketed device, or to a standard recognized by the FDA when used for the same intended purpose [8].

5-6-5 FDA PREMARKET APPROVAL

Pre-market Approval (PMA) is the most stringent type of application required by the FDA and applies to “Class III” devices such as life-support devices, devices with the potential to do great injury or new devices, which have an unknown safety and hazard potential. To gain approval, the manufacturer must present adequate scientific evidence to assure that the device is safe and effective for its intended uses [8].

5-6-6 FDA DEVELOPMENT GUIDANCE

Once the application and device are established, the FDA provides the following documentation that provides guidance as to how the software should be developed, documented and controlled [9]:

- Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices
- General Principles of Software Validation
- Guidance for Off-the-Shelf Software Use in Medical Devices
- Cybersecurity for Networked Medical Devices Containing Off-the-Shelf (OTS) Software

The IEC 62304 Standard incorporates equivalent or superior software life-cycle processes compared with those above, and it is recognized in more markets than the FDA's clearance.

5-6-7 DEVICE CLASS

The FDA defines medical devices as: Class I, Class II, or Class III. The classifications are assigned based on the level of potential hazard associated with a device. The probability that harm will manifest also influences the Class of the device.

- Class I devices represent the lowest degree of hazard and is thereby subject to the lowest amount of regulatory controls. Many Class I devices are exempt and do not have to apply for clearance. If not found exempt, then a 510(k) is required.

Low-hazard devices are devices such as thermometers, blood pressure monitors, and certain laboratory equipment.

- Class II has a higher likelihood of hazard and is therefore subject to greater number of regulatory controls. There are also a number of exempt Class II devices. If not found exempt, then a 510(k) is required.
- Class III is associated with the greatest hazard and level of regulatory controls and is required to submit a PMA. Some Class III devices that have a manufacturer with a proven record of safety, use best practices, etc., may be allowed to use the 510(k) option.

Life-support and critical monitoring equipment are generally considered to be high-hazard devices. If they fail or are faulty, the probability for an adverse patient outcome is high. Devices that have the potential of doing significant harm to the operator also qualify as a Class III device.

Examples include anesthesia equipment that has the potential to harm more than the patient. During an operation, it can release a flammable gas and pure O₂ into an enclosed space. Another class of device that is usually considered a class III device is a therapeutic device that uses energetic energy as part of the therapy. Neutron or electron treatment devices can do large degrees of harm if they fail, or used improperly.

5-6-8 LEVEL OF CONCERN (FDA/CDRH)

Not to be confused with the Medical Device Classification (Class I, II, or III), the FDA has defined additional protocols for devices that use or contain software. In this case, the system as a whole is considered, i.e. consisting of operators, patient, environment, hardware and software (including OTS software) but the Level of Concern (LOC) protocol is a measure of the hazard contributed solely by the software.

The LOC for software falls into one of three categories:

- Major - The software LOC is Major if the software could contribute, either directly or indirectly result in death or serious injury to the patient or operator.
- Moderate - The software LOC is Moderate if the software could contribute, either directly or indirectly result in minor injury to the patient or operator.
- Minor - The software LOC is Minor if failures or latent design flaws are unlikely to cause any injury to the patient or operator.

The FDA provides Table 2-01 [10] to assist in determining the degree of Verification and Validation that is required in order to support the clearance of the device.

Software Documentation	Minor Concern	Moderate Concern	Major Concern
Level of Concern	A statement indicating the Level of Concern and a description of the rationale for that level.		
Software Description	A summary overview of the features and software operating environment.		
Device Hazard Analysis	Tabular description of identified hardware and software hazards, including severity assessment and mitigations.		
Software Requirements Specification (SRS)	Summary of functional requirements from SRS.	The complete Software Requirements Specification (SRS) document.	
Architectural Design Chart	No documentation is necessary in the submission.	Detailed depiction of functional units and software modules. May include state diagrams as well as flow charts.	

Chapter 5

Software Documentation	Minor Concern	Moderate Concern	Major Concern
Software Design Specification (SDS)	No documentation is necessary in the submission.	Software Design Specification (SDS) document.	
Traceability Analysis	Traceability among requirements, specifications, identified hazards and mitigations, and verification and validation testing.		
Software Development Environment Description	No documentation is necessary in the submission.	Summary of software life cycle development plan, including a summary of the configuration management and maintenance activities.	Summary of software life cycle development plan. Annotated list of control documents generated during development process. Include the configuration management and maintenance plan documents.
Verification and Validation Document (V&V)	Software functional test plan, pass / fail criteria, and results.	Description of V&V activities at the unit, integration, and system level. System level test protocol, including pass/ fail criteria, and tests results.	Description of V&V activities at the unit, integration, and system level. Unit, integration and system level test protocols, including pass/ fail criteria, test report, summary, and tests results.
Revision Level History	Revision history log, including release version number and date		
Unresolved Anomalies	No documentation is necessary in the submission.	List of remaining software anomalies, annotated with an explanation of the impact on safety or effectiveness, including operator usage and human factors.	

Table 5-1 Degrees of Verification and Validation

5-7 REGULATORY ENVIRONMENT IN THE EUROPEAN UNION

Medical devices in the European Union and the standards that govern them have been harmonized and codified under a set of comprehensive Medical Device Directives. The primary directives are:

- Directive 90/385/EEC for implantable medical devices
- Directive 2007/47/EEC medical devices
- Directive 98/79/EC in vitro diagnostic medical devices

2007/47/EC is a set of regulations comparable to the ones discussed in the Regulatory Environment in the U.S.

5-7-1 EU DEVICE CLASSES

The EU uses a ranking system similar to the FDA's except it allows for four categories, ranging from low risk to high risk.

- Class I (including Is & Im)
- Class IIa
- Class IIb
- Class III

The difference between a Class IIa, and Class IIb device are beyond the scope of this book.

5-8 MEDICAL STANDARDS

5-8-1 IEC 62304

IEC 62304 Medical device software - Software life-cycle processes, was created by a joint working group of the ISO and IEC team members and released in 2006. As of March 2010, it is mandatory for medical products carrying the CE mark and sold in the EU. With recognition by the FDA as a consensus standard, IEC 62304 is the de facto standard for manufacturers selling to the international market.

IEC 62304 recognizes that the software life cycle involves not just development, but that software is also released and used in the field. It requires up front planning and addresses real-life product issues such as software maintenance, problem resolution, and change management. IEC 62304 makes the assumption that the software life cycle exists within a quality and risk management system. With regard to the risk management process, ISO 14971 is assumed and is a normative standard. ISO 14971 is dominant in the risk management category, while ISO 13485 is the quality management system of choice.

IEC 62304 defines the software lifecycle as a framework of essential processes, including:

- Software development process
- Software maintenance process
- Software hazard management process
- Software configuration management process
- Software problem resolution process
- Risk management process - ISO 14971
- Quality management system (Suggested ISO 13485)

IEC 62304 divides each process into a set of activities, and each set is subdivided into a set of tasks. In addition to identifying the set of processes, activities and tasks that are necessary and sufficient to undertake the project, IEC 62304 also provides a scale, which manufacturers are to use to evaluate the hazard associated with the software as implemented within the device.

Similar to the “Level of Concern” used by the FDA, these software safety classes are assigned based on severity as follows:

- Class A: No injury or damage to health is possible
- Class B: Non-serious injury is possible
- Class C: Death or serious injury is possible

Once the Class of the software is established, IEC 62304 provides a list of the processes, activities and tasks that must be performed in order to develop software that will meet the needs of the application. Table 5-2 below is a sample of the direction provided by the standard. As expected, Process 5, Activity 5.1 requires that the planning task is performed for all classes of software. But for activity 5.1.5, “Software development standards, methods and tools planning” this level of planning is only required for Class C.

Clauses and Sub-clauses	Class A	Class B	Class C
4.0 General Requirements			
Clause 4 All Requirements	X	X	X
<hr/>			
5.0 Software Development Process			
5.1 Software development Planning (activity)	X	X	X
5.1.1 Software Development Plan (task)	X	X	X
5.1.2 Keep Software Development plan ...	X	X	X
5.1.3 Software Development plan reference ...	X	X	X
5.1.4 Software Development standards, ...			X
5.1.5 Software integration and ...		X	X
...
Clause 8 All requirements	X	X	X
<hr/>			
Clause 9 All requirements	X	X	X

Table 5-2 Software safety classes

IEC is not perfect and there is room for improvement. Observations consistently find that IEC 62304 is easier to understand and has less ambiguity, it is provided as a single document and incorporates mechanisms for handling off the shelf software as an integral part of the process, and it provides a process for partitioning software into separate classes to reduce validation costs without sacrificing the integrity of the device.

5-8-2 ISO 14971

The latest release, formally known as ISO 14971:2007 “Medical Devices — Risk Management — Application of Risk Management to Medical Devices,” is recognized by the FDA and specified by the EU’s Medical Device Directive (MDD) and is the normative risk management process used by IEC 62304. Unless otherwise stated, the assumption is that manufacturers use ISO 14971 as their defined and documented process that satisfies risk analysis, evaluation, and control. Similar to IEC 62304, ISO 14971 applies to the device life cycle, not just development [10], [11]. It is best treated as an absolute requirement.

5-8-3 ISO 13485

ISO 13485 “Medical devices -- Quality management systems -- Requirements for regulatory purposes,” is seen as required infrastructure for medical device manufacturers. While other standards such as ISO 9001 are also acceptable, ISO 13495 is tailored to meet the specific regulatory and quality requirements of broad needs of the medical device industry and certain sectors such as implantable devices, and sterile medical devices [10], [11]. It is best treated as an absolute requirement.

5-8-4 MEDICAL SOFTWARE IN THE FUTURE

The working group tasked with predicting the future had a much easier job than either the FDA or the industry as a whole will in the years to come. While the medical industry is not missing ethics, it is optimistic rather than pragmatic. That optimism is exhibited by the “Tell me it is Safe” FDA process compared with a “Prove it” process used in avionics.

While the institutional goals of the FDA are worthy, it is clear that the process to approve device for market represents a clash of two cultures. What other organization in the world drives a safety critical process by detailing the documents that must be submitted rather than the steps needed to ensure that the device is safe?

Why is that important? It is not the paperwork that is critical. The paperwork is a by-product of doing the right thing at the right time. If documentation alone drives the process, safety will never truly catch up.

The convergence of technologies similar to in cell phones and other consumer electronics such as wireless connectivity, 2D and 3D graphics, multimedia, etc will increase convenience, expand capabilities, and drive portability. Many of these technologies will in turn drive the need for even more software such as security and privacy software that is needed because of the wireless and other connectivity. All of this will result in device software code bases that will grow by a magnitude or more, and for many companies most of this software will have to come from outside.

The problem for manufacturers will not be the availability of software, but the availability of software that is qualified or able to be qualified for use in medical devices. Companies such as Micrium, and Validated Software provide software and the evidence of compliance with both FDA and IEC 62304 standards.

5-9 COMMON SAFETY-CRITICAL DEVELOPMENT STANDARDS

Globally, there are many regional, and industry specific standards. Two robust standards specifically involved in software development are IEC 61508 and DO-178B. Given that standards apply to not only software, but all aspect of design, each industry and its related equipment poses dramatically different hazards. Therefore the methods used to evaluate and mitigate these hazards promote the use of diverse standards governing device creation. Specific standards that address not only the medical industry are:

- IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems. IEC-61508-3 specifically addresses software.
- UK Defense Standard 00-56 Issue 2, Safety Management Requirements for Defense Systems
- US Requirements and Technical Concepts for Aviation (RTCA) DO-178B Software Considerations in Airborne Systems and Equipment Certification
- US RTCA DO-278B Guidelines for Communications, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance

Chapter 5

- US RTCA DO-254 North American Avionics Hardware
- EUROCAE ED-12B European Airborne Flight Safety Systems. Technically equivalent to DO- 178B
- IEC 62304 - Medical Device Software - Software Life Cycle Processes
- IEC 61513, Nuclear power plants – Instrumentation and control for systems important to safety General requirements for systems, based on EN 61508
- IEC 61511 Functional safety - safety instrumented systems for the process industry sector
- IEC 62061, Safety of machinery - Functional safety of safety- related electrical, electronic and programmable electronic control systems, based on EN 61508
- EN 50128, Railway applications - Communications, signaling and processing systems. Software for railway control and protection systems
- EN 50129, Railway Industry Specific
- NASA Safety Critical Guidelines

5-10 STANDARDS BODIES AND WORLDWIDE STANDARDS ORGANIZATIONS

- American National Standards Institute (ANSI) - www.ansi.org
- American Society for Testing and Materials (ASTM) - www.astm.org
- Association for the Advancement of Medical Instrumentation (AAMI) - www.aami.org
- Australian Therapeutic Goods Administration - www.tga.gov.au
- British Standards Institution (BSI) - www.bsi-global.com
- Canadian Standards Association (CSA) - www.csa.ca

Standards Bodies and Worldwide Standards Organizations

- European Committee for Electrotechnical Standardization (CENLELEC) - www.cenelec.eu
- European Committee for Standardization (CEN) - www.cen.eu
- European Telecommunications Standards Institute (ETSI) - www.etsi.org
- Finnish Standards Association - www.sfs.fi
- French Association for Standardization (AFNOR) - www.afnor.fr
- German Standards Institute (DIN) - www.din.de
- Global Harmonization Task Force (GHTF) - www.ghtf.org
- IEEE-SA Standards Association (IEEE-SA) – www.standards.ieee.org
- International Electrotechnical Commission (IEC) – www.iec.org
- International Standards Organization (ISO) - www.iso.org
- Medicines and Healthcare Products Regulatory Agency (UK) - www.mhra.gov.uk
- National Institute for Standards and Technology (NIST) - www.nist.gov
- Radio Technical Commission for Aeronautics (RTCA) - <http://www.rtca.org>
- Standards Association of Australia - www.standards.org.au
- Standards Council Canada (SCC) - www.scc.ca
- Swedish Standards Institute (SIS) - www.sis.se
- Swiss Association for Standardization (SNV) - www.snv.ch

5-11 FDA GUIDANCE AND DOCUMENTS

The FDA/CDRH website contains an easily searchable wealth of information. Although access to some specifications such as IEC 62304 is fee-based, it is possible to meet FDA requirements by using strictly public material. A worthwhile place to begin the process is at: How to Market Your Device:

<http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/HowtoMarketYourDevice>

Additional helpful documents to download include:

- Design Control Guidance for Medical Device Manufacturers, CDRH, FDA, March 1997

<http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm070627.htm>

- Do It by Design, An Introduction to Human Factors in Medical Devices, CDRH, FDA, March 1997

<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm095061.pdf>

- Electronic Records; Electronic Signatures Final Rule, 62, Federal Register 13430, March, 1997

<http://www.fda.gov/downloads/RegulatoryInformation/Guidances/ucm125125.pdf>

- General Principles of Software Validation; Final Guidance for Industry and FDA Staff Document, Center for Devices and Radiological Health, FDA, January 2002

<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085371.pdf>

- Glossary of Computerized System and Software Development Terminology, Division of Field Investigations, Office of Regional Operations, Office of Regulatory Affairs, FDA, August 1995

<http://www.fda.gov/iceci/inspections/inspectionguides/ucm074875.htm>

- Guidance for the Content of Pre-market Submissions for Software Contained in Medical Devices, Office of Device Evaluation, CDRH, Food and Drug Administration, May 11, 2005
<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm089593.pdf>
- Guidance for Industry, FDA Reviewers and Compliance on Off-the-Shelf Software Use in Medical Devices, Office of Device Evaluation, Center for Devices and Radiological Health, Food and Drug Administration, September 1999
<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm073779.pdf>
- Guidance for Industry Process Validation: General Principles and Practices, Center for Biologics Evaluation and Research (CBER), Food and Drug Administration January 2011
<http://www.fda.gov/downloads/Drugs/GuidanceComplianceRegulatoryInformation/Guidances/UCM070336.pdf>
- Medical Devices; Current Good Manufacturing Practice (CGMP) Final Rule; Quality System Regulation, 61 Federal Register 52602, October 7 1996
<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/PostmarketRequirements/QualitySystemsRegulations/MedicalDeviceQualitySystemsManual/UCM122806.pdf>
- Reviewer Guidance for a Pre-Market Notification Submission for Blood Establishment Computer Software, Center for Biologics Evaluation and Research, Food and Drug Administration, January 1997
<http://www.fda.gov/downloads/BiologicsBloodVaccines/GuidanceComplianceRegulatoryInformation/OtherRecommendationsforManufacturers/MemorandumtoBloodEstablishments/UCM062208.pdf>

- Blood Establishment Computer Software: Understanding What to Include in a 510 (K) Submission, Transcript J Murray, Nov 2009, pg. 180
<http://www.fda.gov/downloads/BiologicsBloodVaccines/NewsEvents/WorkshopsMeetingsConferences/UCM198711.pdf>
- Guidance for Industry - Cybersecurity for Networked Medical Devices Containing Off-the-Shelf (OTS) Software, January 2005
<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm077823.pdf>
- Implementation of Risk Management Principles and Activities within a Quality System, Management, July 2005
<http://www.ghtf.org/documents/sg3/sg3n15r82005.pdf>
- Quality Management Systems - Process Validation Guidance, FDA Draft Guidance, January 2004
http://www.ghtf.org/documents/sg3/sg3_fd_n99-10_edition2.pdf

5-12 REFERENCES

- [1] W. Herman, G. Devey, "Future Trends in Medical Device Technologies", 2008.
- [2] U.S. Food and Drug Administration (FDA), "General Principles of Software Validation; Final Guidance for Industry and FDA Staff", 2002.
- [3] J. Murray Jr., "CDRH Software Update", 2008.
- [4] U.S. Food and Drug Administration (FDA), "Glossary of Computerized System and Software Development Terminology", 2005.
- [5] RTI - Health, Social, and Economic Research, G. Tassey , Ph.D. (NIST), "The Economic Impacts of Inadequate Infrastructure for Software Testing - Final Report", 2002.

References

-
- [6] B. Dolan, Interview: “The iPhone medical app denied 510(k)”, Mobile Health News, 2010.
 - [7] U.S. Food and Drug Administration (FDA), “Guidance for Industry, FDA Reviewers and Compliance on: Off-The-Shelf Software Use in Medical Devices”, 1999.
 - [8] U.S. Food and Drug Administration (FDA), “How to market your device”, 2011
 - [9] U.S. Food and Drug Administration (FDA), “Device classification”, 2011
 - [10] Swiss International Electrotechnical Commission, “International standard 62304: Medical device software – Software life cycle processes”, 2006.
 - [11] European Commission, Health & Consumers Directorate-General, “Implementation of Directive 2007/47/EC Amending Directives 90/385/EEC, 93/42/EEC AND 98/8/EC”, 2009.

Chapter 6

Freescale's Tower System Controller TWR-K53N512

The TWR-K53N512 is a Tower Controller Module compatible with the Freescale Tower System. It can function as a stand-alone, low-cost platform for the evaluation of the Kinetis K53 family of microcontroller (MCU) devices.

The TWR-K53N512 features the Kinetis K53 low-power MCU based on the ARM® Cortex™-M4 architecture. As we will describe in more detail in the next chapter, the MCU comes with USB 2.0 full-speed OTG, 10/100 Mbps Ethernet MAC, a flexible, low-power segment LCD controller and analog front end capabilities specifically designed for medical devices. The K53N512 includes 256Kbytes of program flash storage and an additional 256Kbytes of FlexMemory non-volatile storage that can be used as additional program flash memory, data flash, or variable size/endurance EEPROM.

The TWR-K53N512 is the controller we used for each of the four medical application examples featured in this book. As shown in Figure 6-1 the board provides interface to the medical expansion connector that allows you to expand the capabilities of the system by connecting the following Analog Front End (AFE) plugins from Freescale:

- MED-EKG: Electrocardiograph
- MED-GLU: Blood glucose meter
- MED-SPO2: Pulse oximeter
- MED-BPM: Blood pressure monitor

We will describe in more detail each of these AFE plug-ins in their corresponding example chapters 5, 6, 7 and 8.

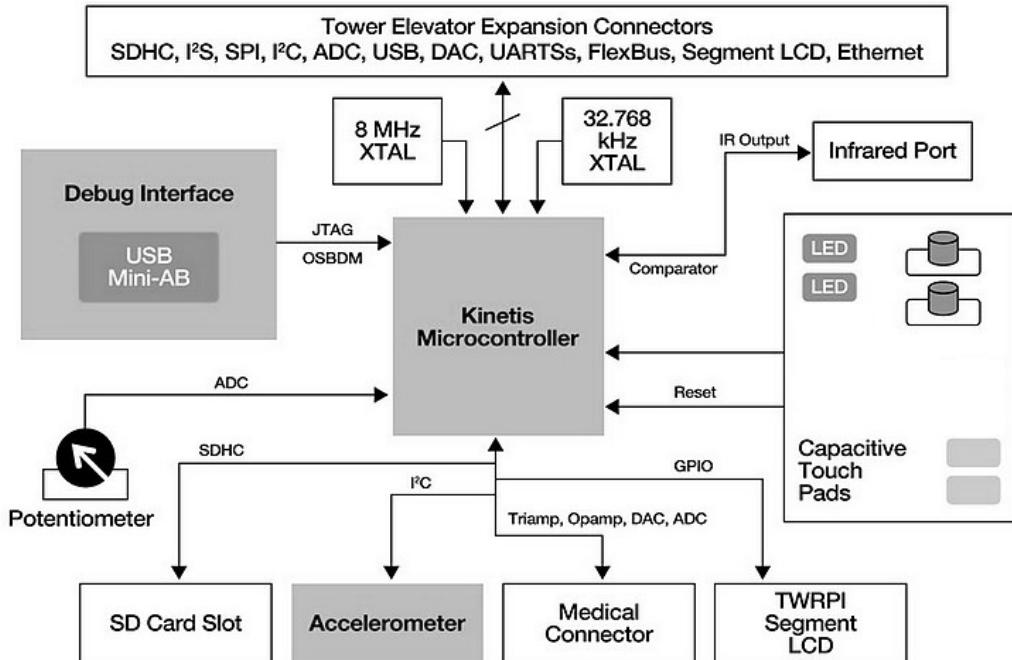


Figure 6-1 TWR-K53N512 Block Diagram

The following list summarizes the features of the TWR-K53N512 and Figure 6-2 and Figure 6-3 show the TWR-K53N512 with some of the key features called out:

- Tower compatible microcontroller module
- MK53N512VMD100: K53N512 in a 144 MAPBGA with 100MHz operation
- Touch Tower Plug-in Socket
- General purpose Tower Plug-In (TWRPI) socket
- On-board JTAG debug circuit (OSJTAG) with virtual serial port
- 2x10 Medical expansion connector
- Three axis accelerometer (MMA7660)
- Two (2) user-controllable LEDs
- Two (2) capacitive touch pads
- Two (2) user mechanical push buttons

Freescale's Tower System Controller TWR-K53N512

- Potentiometer
- Battery Holder for 20mm lithium battery (e.g. 2032, 2025)
- SD Card slot

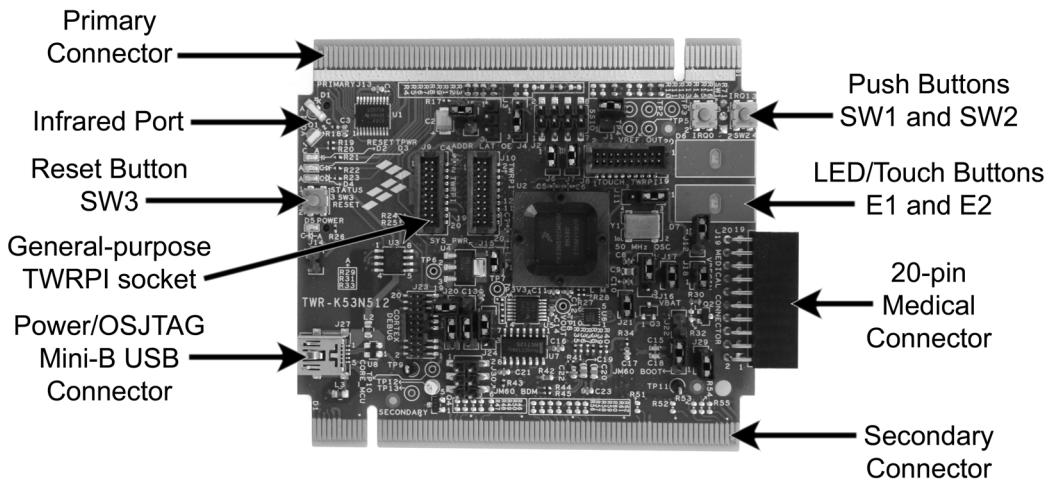


Figure 6-2 Front side of the TWR-K53N512

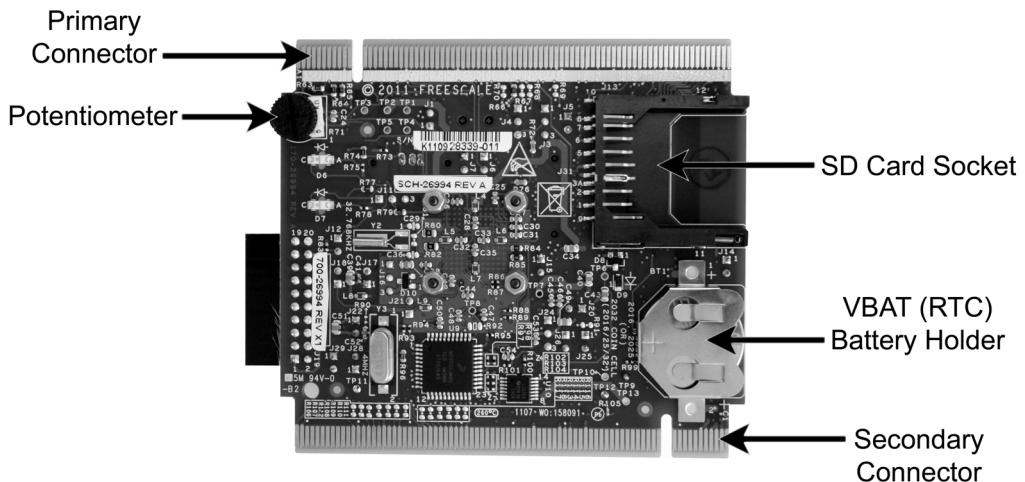


Figure 6-3 Back side of the TWR-K53N512

6-1 TWR-K53N512 PIN USAGE

Table 6-1 provides details on which K53N512 pins are used to communicate with the LEDs, pushbuttons, and other I/O interfaces onboard the TWR-K53N512. In order to avoid attempted simultaneous usage of mutually exclusive features, be aware that some port pins are used in multiple interfaces on-board and many are potentially connected to off-board resources via the Primary and Secondary Connectors.

TWR-K53N512 Pinout

Feature	Connection	Port Pin	Pin Function
OSJTAG USB-to-serial Bridge	OSJTAG Bridge RX Data	PTE9	UART5_RX
	OSJTAG Bridge TX Data	PTE8	UART5_TX
SD Card Slot	SD Clock	PTE2	SDHC0_DCLK
	SD Command	PTE3	SDHC0_CMD
	SD Data0	PTE1	SDHC0_D0
	SD Data1	PTE0	SDHC0_D1
	SD Data2	PTE5	SDHC0_D2
	SD Data3	PTE4	SDHC0_D3
	SD Card Detect	PTE28	PTE28
	SD Write Protect	PTC9	PTC9
Infrared Port	IR Transmit	PTD7	CMT_IRO
	IR Receive	PTC6	CMP0_IN0
Pushbuttons	SW1 (IRQ0)	PTC5	PTC5
	SW2 (IRQ1)	PTC13	PTC13
	SW3 (RESET)	RESET_b	RESET_b
Touch Pads	E1 / Touch	PTB17	TSI0_CH10
	E2 / Touch	PTB18	TSI0_CH11
LEDs	E1 / Orange LED	PTC7	PTC7
	E2 / Yellow LED	PTC8	PTC8
Potentiometer	Potentiometer (R71)	ADC1_DM1	ADC1_DM1
Accelerometer	I2C SDA	PTC11	I2C0_SDA
	I2C SCL	PTC10	I2C0_SCL
	IRQ	PTC12	PTD10

TWR-K53N512 Pinout

Feature	Connection	Port Pin	Pin Function
General Purpose TWRPI Socket	TWRPI AN0 (J9 Pin 8)	PTB6	ADC0_DP0/ADC1_DP3
	TWRPI AN1 (J9 Pin 9)	PTB7	ADC0_DM0/ADC1_DM3
	TWRPI AN2 (J9 Pin 12)	PTB5	ADC1_DP0/ADC0_DP3
	TWRPI ID0 (J9 Pin 17)	PTE0	ADC0_DP1
	TWRPI ID1 (J9 Pin 18)	PTE1	ADC0_DM1
	TWRPI I2C SCL (J10 Pin 3)	PTC10	I2C1_SCL
	TWRPI I2C SDA (J10 Pin 4)	PTC11	I2C1_SDA
	TWRPI SPI MISO (J10 Pin 9)	PTB23	SPI2_SIN
	TWRPI SPI MOSI (J10 Pin 10)	PTB22	SPI2_SOUT
	TWRPI SPI SS (J10 Pin 11)	PTB20	SPI2_PCS0
	TWRPI SPI CLK (J10 Pin 12)	PTB21	SPI2_SCK
	TWRPI GPIO0 (J10 Pin 15)	PTC12	PTC12
	TWRPI GPIO1 (J10 Pin 16)	PTB9	PTB9
	TWRPI GPIO2 (J10 Pin 17)	PTB10	PTB10
Touch Pad / Segment LCD TWRPI Socket	TWRPI GPIO3 (J10 Pin 18)	PTC5	PTC5
	TWRPI GPIO4 (J10 Pin 19)	PTC13	PTC13
	Electrode 0 (J8 Pin 3)	PTB0	TSI0_CH0
	Electrode 1 (J8 Pin 5)	PTB1	TSI0_CH6
	Electrode 2 (J8 Pin 7)	PTB2	TSI0_CH7
	Electrode 3 (J8 Pin 8)	PTB3	TSI0_CH8
	Electrode 4 (J8 Pin 9)	PTC0	TSI0_CH13
	Electrode 5 (J8 Pin 10)	PTC1	TSI0_CH14
	Electrode 6 (J8 Pin 11)	PTC2	TSI0_CH15
	Electrode 7 (J8 Pin 12)	PTA4	TSI0_CH5
	Electrode 8 (J8 Pin 13)	PTB16	TSI0_CH9
	Electrode 9 (J8 Pin 14)	PTB17	TSI0_CH10
	Electrode 10 (J8 Pin 15)	PTB18	TSI0_CH11
	Electrode 11 (J8 Pin 16)	PTB19	TSI0_CH12
	TWRPI ID0 (J8 Pin 17)	PTE2	ADC1_DP1
	TWRPI ID1 (J8 Pin 18)	ADC1_DM0	ADC1_DM0

TWR-K53N512 Pinout

Feature	Connection	Port Pin	Pin Function
Medical Connector	Voltage Supply (J19 Pin 1)	Pin3	Source of P-Channel MOSFET
	Ground (J19 Pin 2)	GND	Ground
	I2C Data (J19 Pin 3)	I2C1_SDA	PTC11/I2C1_SDA
	I2C Clock (J19 Pin 4)	I2C_SCL/FTM2_CH1	I2C1_SCL/FTM2_CH1 (J4 jpr)
	ADC0 Diff (+) Ch 0 (J19 Pin 5)	ADC0_DP0	ADC0_DP0/ADC1_DP3
	ADC0 Diff (-) Ch 0 (J19 Pin 6)	ADC0_DM0	ADC0_DM0/ADC1_DM3
	ADC1 Diff (+) Ch 0 (J19 Pin 7)	ADC1_DP0	ADC1_DP0/ADC0_DP3
	DAC0 Output (J19 Pin 8)	DAC0_OUT	DAC0_OUT
	OPAMP0 Output (J19 Pin 9)	OP0_OUT	ADC0_SE16/OP0_OUT
	OPAMP1 Output (J19 Pin 10)	OP1_OUT	ADC1_SE16/OP1_OUT
	OPAMP0 (-) Input (J19 Pin 11)	OP0_DM0	ADC0_DM1/OP0_DM0
	OPAMP1 (-) Input (J19 Pin 12)	OP1_DM0	ADC1_DM1/OP1_DM0
	OPAMP0 (+) Input (J19 Pin 13)	OP0_DP0	ADC0_DP1/OP0_DP0
	OPAMP1 (+) Input (J19 Pin 14)	OP1_DP0	ADC1_DP1/OP1_DP0/OP1_DM1
	TRIAMP0 (+) Input (J19 Pin 15)	TRI0_DP	TRI0_DP
	TRIAMP1 (+) Input (J19 Pin 16)	TRI1_DP	TRI1_DP
	TRIAMP0 (-) Input (J19 Pin 17)	TRI0_DM	TRI0_DM
	TRIAMP1 (-) Input (J19 Pin 18)	TRI1_DM	TRI1_DM
	TRIAMP0 Output (J19 Pin 19)	TRI0_OUT	TRI0_OUT/OP1_DM2
	TRIAMP1 Output (J19 Pin 20)	TRI1_OUT	TRI1_OUT

Table 6-1 **TWR-K53N512 Pinout**

6-2 TWR-K53N512 JUMPER SETTINGS

Table 6-2 provides details on the different configurations supported by the TWR-K53N512. Use the default settings shown in bold for all the medical applications featured in this book.

Jumper	Option	Setting	Description
J1	ADC1_DM1 Input Selection	ON	ADC1_DM1 reads POTENTIOMETER
		OFF	ADC1_DM1 reads MEDICAL CONNECTOR
J3	FlexBus Address Latch Selection	2-3	Enable FlexBus address latch
		1-2	Disable FlexBus address latch
J4	Medical Connector J19 Pin3 Selection	1-2	Select I2C1_SCL connection to MEDICAL CONNECTOR
		2-3	Select FTM2_CH1 connection to MEDICAL CONNECTOR
J5	IR Transmitter Connection	OFF	Disconnect PTD7/CMT_IRO from IR transmitter circuit (IRDA)
		ON	Connect PTD7/CMT_IRO to IR transmitter circuit (IRDA)
J6	FlexBus or SSIO Selection	ON	Use PTE7 for Flex bus
		OFF	Use PTE7 for SSIO
J7	Ethernet/TOUCH PAD TWRPI Selection	ON	Use PTB0 for Ethernet
		OFF	Use PTB0 for TOUCH PAD TWRPI
J11	Clock Input Source Selection	1-2	Connect main EXTAL to on-board 50 MHz clock
		2-3	Connect EXTAL to the CLKIN0 signal on the elevator connector
J12	SD Card/TOUCH PAD TWRPI Selection	OFF	Use PTE2 for SD card reader (SD/MMC SKT)
		ON	Use PTE2 for TOUCH PAD TWRPI
J14	IR Transmitter Filter Selection	OFF	IR input to CMP0_IN0 is not low-pass filtered by a 0.1 uF cap
		ON	IR input to CMP0_IN0 is low-pass filtered by a 0.1 uF cap

Jumper	Option	Setting	Description
J15	MCU Power Connection	ON	Connect on-board 3.3V supply to MCU
		OFF	Isolate MCU from power (connect an ammeter to measure current)
J16	VBAT Power Connection	1-2	Connect VBAT to on-board 3.3V supply
		2-3	Connect VBAT to the higher voltage between on-board 3.3V supply or coin-cell supply
J17	On-Board 50 MHz Power Connection	ON	Connect on-board 3.3V supply to on-board 50 MHz OSC
		OFF	Disconnect on-board 3.3V supply to on-board 50 MHz OSC
J18	VREGIN Power Connection	ON	Connect USB0_VBUS from Elevator to VREGIN
		OFF	Disconnect USB0_VBUS from Elevator to VREGIN
J20	SD Card/GENERAL PURPOSE TWRPI Selection	OFF	Use PTE1 for SD card reader (SD/MMC SKT)
		ON	Use PTE1 for GENERAL PURPOSE TWRPI
J21	Accelerometer Power Connection	ON	Connect accelerometer to on-board 3.3V supply
		OFF	Disconnect accelerometer from on-board 3.3V supply
J22	Off-Board Power input	OFF	J22 pin 1 can be connected to an off-board external power source. This board is only tested with 3.3V. Care should be taken not to connect to a voltage that is out of the components specification
		OFF	J22 pin 2 can be connected to the ground of the off-board external power source
J24	Off or On Board Power Input Selection	1-2	Board SYS_PWR is powered from on-board 3.3V regulator
		2-3	Board SYS_PWR is powered from off-board supply from J22 pin 2
J25	JTAG Board Power Connection	OFF	Disconnect on-board 5V supply to JTAG port
		ON	Connect on-board 5V supply to JTAG port (supports powering board from JTAG pod supporting 5V supply output)

Jumper	Option	Setting	Description
J26	SD Card/GENERAL PURPOSE TWRPI Selection	OFF	Use PTE0 for SD card reader (SD/MMC SKT)
		ON	Use PTE0 for GENERAL PURPOSE TWRPI
J28	OSJTAG Bootloader Selection	OFF	Debugger mode
		ON	OSJTAG bootloader mode (OSJTAG firmware reprogramming)
J29	Ethernet/TOUCH PAD TWRPI Selection	ON	Use PTB1 for Ethernet
		OFF	Use PTB1 for TOUCH PAD TWRPI
J32	TOUCH PAD/SLCD TWRPI Selection	1-2	PTB10_LCD_P10 pin is connected to J8 pin 3 for SLCD TWRPI
		2-3	PTB0_TSI0_CH0 pin is connected to J8 pin 3 for TOUCH PAD TWRPI. Make sure J29 and J7 are off to avoid conflict with Ethernet
J33	TOUCH PAD/SLCD TWRPI Selection	1-2	PTB11_LCD_P11 pin is connected to J8 pin 5 for SLCD TWRPI
		2-3	PTB1_TSI0_CH6 pin is connected to J8 pin 5 for TOUCH PAD TWRPI. Make sure J29 and J7 are off to avoid conflict with Ethernet
J34	On-Board 50 MHz Enable Source	OFF	On-board 50 MHz osc is enabled if J17 jumper is on. No need to have any jumper on J34
		1-2	On-board 50 MHz osc is enabled if J17 jumper is on
		2-3	On-board 50 MHz osc enable by GPIO PTA19 allowing MCU to turn off clock for lower power consumption

Table 6-2 TWR-K53N512 Jumper Settings

6-3 THE FREESCALE TOWER SYSTEM KIT FOR THE K53

The TWR-K53N512 is available as a stand-alone product as described in the previous sections or as a kit (TWR-K53N512-KIT) with the Tower Elevator Modules (TWR-ELEV) and the Tower Serial Module (TWR-SER). The TWR-K53N512 can also be combined with other Freescale Tower peripheral modules to create development platforms for a wide variety of applications as we will describe in the last section. Figure 6-4 provides an overview of the Freescale Tower System.

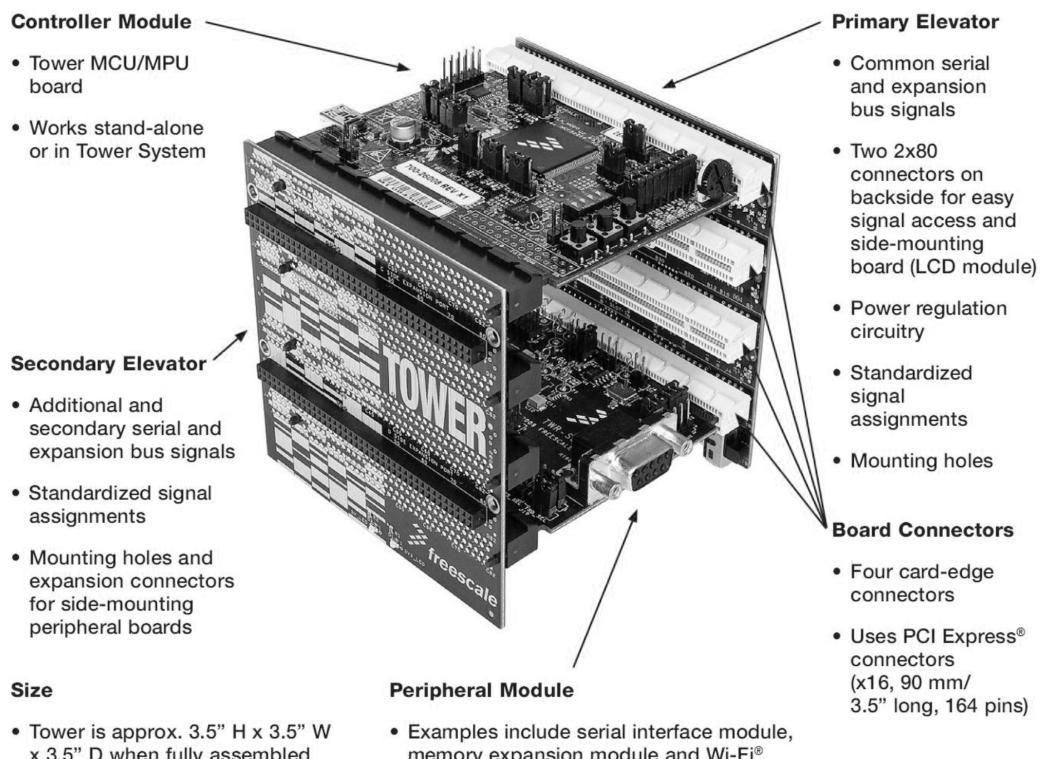


Figure 6-4 Freescale Tower System Overview

The TWR-K53N512 features two expansion card-edge connectors that interface to the Primary and Secondary Elevator boards in a Tower System. The Primary Connector (comprised of sides A and B) is utilized by the TWR-K53N512 while the Secondary Connector (comprised of sides C and D) only makes connections to the GND pins. Table 6-3 provides the pinout for the Primary Connector.

PCI Express Tower System Elevator Connector

Primary Connector Pinout					
Pin#	Side B		Pin#	Side A	
	Name	Usage		Name	Usage
B1	5V	5.0V Power	A1	5V	5.0V Power
B2	GND	Ground	A2	GND	Ground
B3	3.3V	3.3V Power	A3	3.3V	3.3V Power
B4	ELE_PS_SENSE	Elevator Power Sense	A4	3.3V	3.3V Power
B5	GND	Ground	A5	GND	Ground
B6	GND	Ground	A6	GND	Ground
B7	SDHC_CLK / SPI1_CLK	PTE2	A7	SCL0	PTD8
B8	SDHC_D3 / SPI1_CS1_b		A8	SDA0	PTD9
B9	SDHC_D3 / SPI1_CS0_b	PTE4	A9	GPIO9 / CTS1	PTC19
B10	SDHC_CMD / SPI1_MOSI	PTE1	A10	GPIO8 / SDHC_D2	PTE5
B11	SDHC_D0 / SPI1_MISO	PTE3	A11	GPIO7 / SD_WP_DET	PTE27
B12	ETH_COL		A12	ETH_CRS	
B13	ETH_RXER	PTA5	A13	ETH_MDC	PTB1
B14	ETH_TXCLK		A14	ETH_MDIO	PTB0
B15	ETH_TXEN	PTA15	A15	ETH_RXCLK	
B16	ETH_TXER		A16	ETH_RXDV	PTA14
B17	ETH_TXD3		A17	ETH_RXD3	
B18	ETH_TXD2		A18	ETH_RXD2	
B19	ETH_TXD1	PTA17	A19	ETH_RXD1	PTA12
B20	ETH_TXD0	PTA16	A20	ETH_RXD0	PTA13
B21	GPIO1 / RTS1	PTC18	A21	SSI_MCLK	PTE6
B22	GPIO2 / SDHC_D1	PTE0	A22	SSI_BCLK	PTE12
B23	GPIO3	PTE28	A23	SSI_FS	PTE11
B24	CLKIN0	PTA18	A24	SSI_RXD	PTE7
B25	CLKOUT1	PTE26	A25	SSI_TXD	PTE10
B26	GND	Ground	A26	GND	Ground
B27	AN7	PTB7	A27	AN3	PGA0_DP/ADC0_DP0/ ADC1_DP3
B28	AN6	PTB6	A28	AN2	PGA0_DM/ADC0_DM0/ ADC1_DM3
B29	AN5	PTB5	A29	AN1	PGA1_DP/ADC1_DP0/ ADC0_DP3

PCI Express Tower System Elevator Connector**Primary Connector Pinout**

Pin#	Side B		Pin#	Side A	
	Name	Usage		Name	Usage
B30	AN4	PTB4	A30	AN0	PGA1_DM/ADC1_DM0/ ADC0_DM3
B31	GND	Ground	A31	GND	Ground
B32	DAC1	DAC1_OUT	A32	DAC0	DAC0_OUT
B33	TMR3		A33	TMR1	PTA9
B34	TMR2	PTD6	A34	TMR0	PTA8
B35	GPIO4	PTB8	A35	GPIO6	PTB9
B36	3.3V	3.3V Power	A36	3.3V	3.3V Power
B37	PWM7	PTA2	A37	PWM3	PTA6
B38	PWM6	PTA1	A38	PWM2	PTC3
B39	PWM5	PTD5	A39	PWM1	PTC2
B40	PWM4	PTA7	A40	PWM0	PTC1
B41	CANRX0	PTE25	A41	RXD0	PTE25
B42	CANTX0	PTE24	A42	TXD0	PTE24
B43	1WIRE		A43	RXD1	PTC16
B44	SPI0_MISO	PTD14	A44	TXD1	PTC17
B45	SPI0_MOSI	PTD13	A45	VSS	VSSA
B46	SPI0_CS0_b	PTD11	A46	VDDA	VDDA
B47	SPI0_CS1_b	PTD15	A47	VREFA1	VREFH
B48	SPI0_CLK	PTD12	A48	VREFA2	VREFL
B49	GND	Ground	A49	GND	Ground
B50	SCL1	PTD8	A50	GPIO14	
B51	SDA1	PTD9	A51	GPIO15	
B52	GPIO5 / SD_CARD_DET	PTE28	A52	GPIO16	
B53	USB0_DP_PDOWN		A53	GPIO17	
B54	USB0_DM_PDOWN		A54	USB0_DM	USB0_DM
B55	IRQ_H	PTA24	A55	USB0_DP	USB0_DP
B56	IRQ_G	PTA24	A56	USB0_ID	
B57	IRQ_F	PTA25	A57	USB0_VBUS	VREGIN
B58	IRQ_E	PTA25	A58	TMR7	
B59	IRQ_D	PTA26	A59	TMR6	
B60	IRQ_C	PTA26	A60	TMR5	

PCI Express Tower System Elevator Connector

Primary Connector Pinout					
Pin#	Side B		Pin#	Side A	
	Name	Usage		Name	Usage
B61	IRQ_B	PTA27	A61	TMR4	
B62	IRQ_A	PTA27	A62	RSTIN_b	RESET_b
B63	EBI_ALE / EBI_CS1_b	PTD0	A63	RSTOUT_b	RESET_b
B64	EBI_CS0_b	PTD1	A64	CLKOUT0	PTC3
B65	GND	Ground	A65	GND	Ground
B66	EBI_AD15	PTB18	A66	EBI_AD14	PTC0
B67	EBI_AD16	PTB17	A67	EBI_AD13	PTC1
B68	EBI_AD17	PTB16	A68	EBI_AD12	PTC2
B69	EBI_AD18	PTB11	A69	EBI_AD11	PTC4
B70	EBI_AD19	PTB10	A70	EBI_AD10	PTC5
B71	EBI_R/W_b	PTC11	A71	EBI_AD9	PTC6
B72	EBI_OE_b	PTB19	A72	EBI_AD8	PTC7
B73	EBI_D7	PTB20	A73	EBI_AD7	PTC8
B74	EBI_D6	PTB21	A74	EBI_AD6	PTC9
B75	EBI_D5	PTB22	A75	EBI_AD5	PTC10
B76	EBI_D4	PTB23	A76	EBI_AD4	PTD2
B77	EBI_D3	PTC12	A77	EBI_AD3	PTD3
B78	EBI_D2	PTC13	A78	EBI_AD2	PTD4
B79	EBI_D1	PTC14	A79	EBI_AD1	PTD5
B80	EBI_D0	PTC15	A80	EBI_AD0	PTD6
B81	GND	Ground	A81	GND	Ground
B82	3.3V	3.3V Power	A82	3.3V	3.3V Power

Table 6-3 TWR-K53N512 Primary Elevator Connector Pinout

6-4 TOWER SYSTEM MODULE TWR-SER

The TWR-SER Serial Module from Freescale provides USB, Ethernet, CAN and RS-232/485 connectivity solutions for designers developing with the Freescale Tower System. This peripheral module is designed to be combined and used with other microcontroller and peripheral modules in the Tower System.

For the examples provided in this book, the TWR-SER implements the Ethernet Physical Layer for the Ethernet controller on the Kinetis K53 MCU onboard the TWR-K53N512 controller.

For the purpose of Ethernet connectivity the TWR-SER peripheral module illustrated in Figure 6-5 offers a 10/100 Ethernet PHY with MII and RMII interface and it also provides the RJ45 Ethernet connector with integrated magnetics and LEDs.

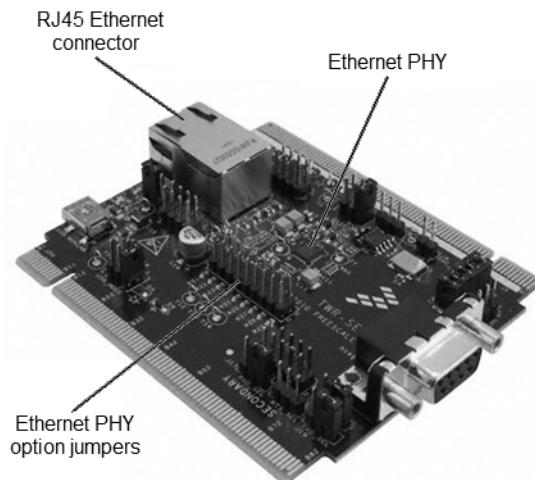
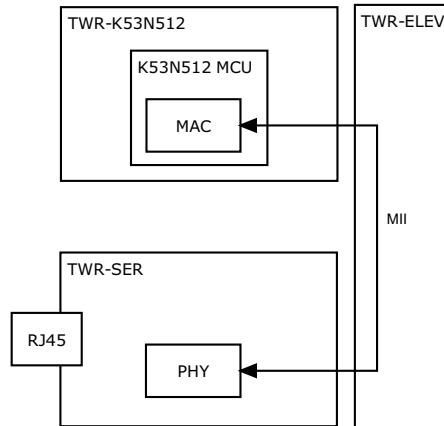
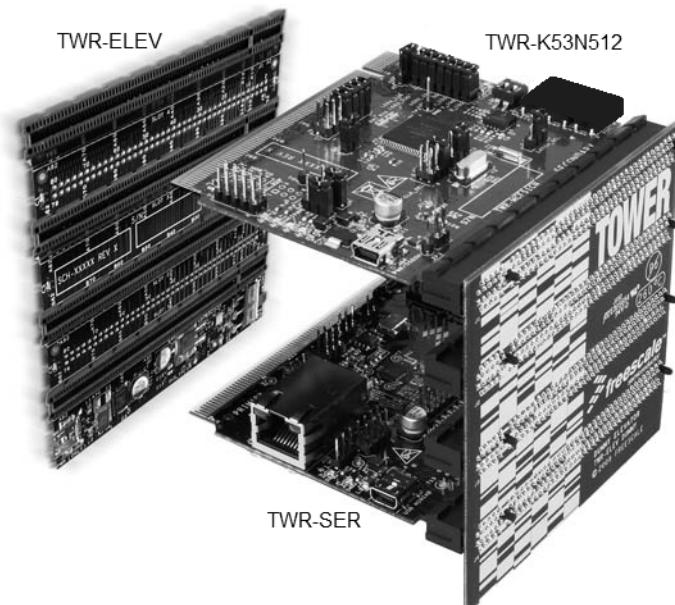


Figure 6-5 **Freescale TWR-SER peripheral module**

The connection between the TWR-K53N512 controller and the TWR-SER peripheral module is through the TWR-ELEV elevator module as shown in Figure 6-6.

Figure 6-6 **MAC and PHY interface**

The TWR-K53N512 controller and the TWR-SER peripheral modules are part of the evaluation kit TWR-K53N512-KIT offered by Freescale and shown in Figure 6-7

Figure 6-7 **TWR-K53N512-KIT**

6-5 TWR-SER JUMPER SETTINGS

Table 6-2 provides details on the different configurations supported by the TWR-SER. Use the default settings shown in bold for all the medical applications featured in this book.

Jumper	Option	Setting	Description
J2	Ethernet PHY Clock Select	1-2	25 MHz
		3-4	50 MHz
		5-6	CLOCKOUT0
J3	CLOCKIN0 Driver Select	1-2	Enable FlexBus address latch
		2-3	Disable FlexBus address latch
J5	CAN Selection Options	1-2	Put CAN transceiver into sleep mode
		3-4	Connect sleep pin to CAN pin (B43)
		5-6	Connect RXD pin to CANRX pin (B41)
		7-8	Connect TXD pin to CANTX PIN (B42)
		9-10	Apply 120 ohm termination resistor
J6	Ethernet PHY Interrupt Select	1-2	IRQ_H
		3-4	IRQ_F
		5-6	IRQ_D
		7-8	IRQ_B
J10	USB VBUS Select	1-2	Supply 5V on USB Connector (host mode)
		2-3	Source 5V from USB (device mode)
J11	USB OTG Interrupt Select	1-2	IRQ_H
		3-4	IRQ_F
		5-6	IRQ_D
		7-8	IRQ_B

Jumper	Option	Setting	Description
J12	Ethernet PHY Configuration	1-2	Pull-up PHYAD2; PHY Address Select
		3-4	Pull-up PHYAD1; PHY Address Select
		5-6	Pull-down PHYAD0; PHY Address Select
		7-8	Pull-up CONFIG2; Loopback Select
		9-10	Pull-up CONFIG0; RMII Select
		11-12	Pull-up ISO; Isolation Mode Select
		13-14	Pull-down SPEED; 10Mbps Select
		15-16	Pull-down DUPLEX; Half-duplex Select
		17-18	Pull-down NWAYEN; Disable Auto-Negotiation
J13	Misc RS-232/485 Config	1-2	Connect RS-485 Receive En and Driver En
		3-4	Connect RS-485 RX+ to TX+; Loopback
		5-6	Connect RS-485 RX- to TX-; Loopback
		7-8	Enable ELE_CTS (A9) as RS-232 CTS
		9-10	Supply 5V on DB9 pin 6
J15	RS-232 / RS-485 Select	1-2	RS-232
		3-4	RS-485
J16	USB Mode Select	1-2	Host mode
		3-4	Device mode
		5-6	OTG mode
J17	RS-232 / RS-485 RX Select	1-2	RS-232
		2-3	RS-485
J18	RS-232 / RS-485 RTS Select	1-2	RS-232
		2-3	RS-485

Table 6-4 **TWR-SER Jumper Settings**

6-6 EXPANDING THE CAPABILITIES

The modularity and affordability of the Tower System makes it simple to add new system capabilities, like Wi-Fi, Sensors, Graphical LCD, and more. Sold separately as individual development boards, these peripheral modules quickly plug into a variety of Tower System configurations for easy, quick prototyping.

Figure 6-8 shows a Prototyping Module called TWR-PROTO that provides an easy way for designers to add custom circuitry to their Tower System designs. The TWR-PROTO module is a 9 x 8 cm board with card-edge connectors that allows it to be plugged directly into the Tower System. The perfboard area provides access to all of the signals from the TWR-K53N512 as well as a generous 8.3 x 3.8 cm prototyping area where you could create your own biomedical signal conditioning circuits if you choose not to buy the Analog-Front-Ends plugins.

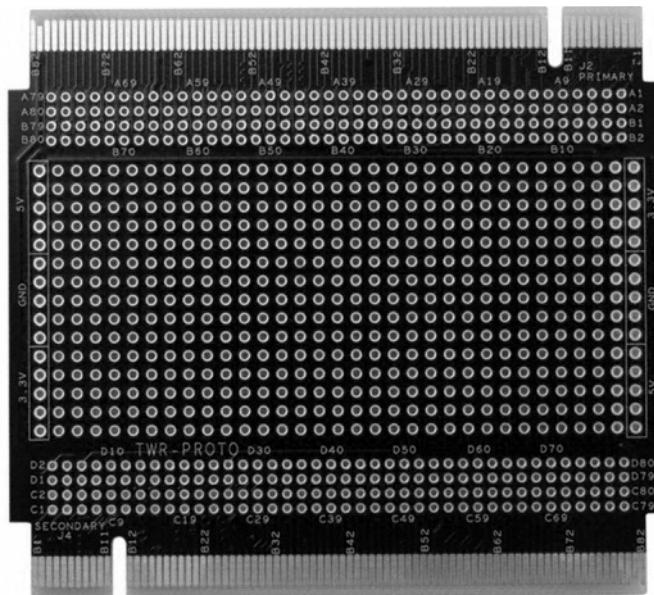


Figure 6-8 Prototyping Board TWR-PROTO

A graphical LCD module called TWR-LCD provides an easy way for designers to add an LCD interface to their Tower System designs. It features a 3.2" QVGA TFT LCD display and attaches to the outer edge of the Tower System elevator modules by Side Expansion Port connectors as shown in Figure 6-9. This peripheral module will not be used in these examples as we will use µC/Probe to prototype the front panel of all four medical devices but still it is a very nice option in case you want to take the examples further.



Figure 6-9 Graphical LCD Board TWR-LCD

Check the Freescale website at www.freescale.com/tower for more information on the TWR-PROTO, TWR-LCD and much more peripheral modules from Freescale and other partners.

Chapter

7

Setup

In this chapter you will learn how to setup an environment to run the µC/OS-II-based medical application projects.

To run the examples provided with this book, it will be necessary to download a number of files from the Internet:

- 1 The µC/OS-II source code and medical application sample projects for the TWR-K53N512 from the Micrium website
- 2 µC/Probe from the Micrium website
- 3 The IAR Embedded Workbench for ARM, 32KB KickStart edition from the IAR website

Each of these downloads will be described in detail throughout this chapter.

7-1 DOWNLOADING µC/OS-II PROJECTS FOR THIS BOOK

To obtain the µC/OS-II source code and projects for the medical application examples provided in this book, simply point your favorite browser to:

www.Micrium.com/Books/Micrium-uCOS-II

You will be required to register. This means that you'll have to provide information about yourself. This information will be used for market research purposes and will allow us to contact you should new updates of µC/OS-II for this book become available. Your information will be held in strict confidence.

Download and execute the following file:

Micrium-Book-uCOS-II-TWR-K53N512.exe

Figure 7-1 shows the directory structure created by this executable.

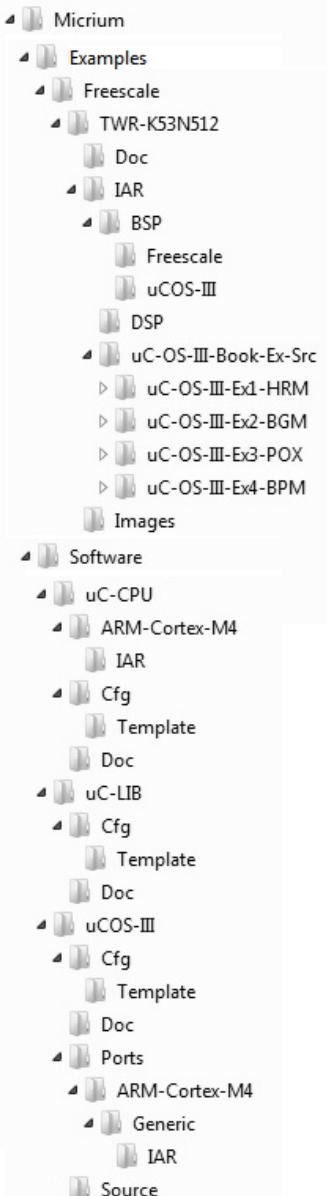


Figure 7-1 **μC/OS-II Project Directories**

All files are placed under the \Micrium directory. There are two main sub-directories: \Examples and \Software and they are described below.

7-1-1 \Examples

This is the standard Micrium sub-directory where all evaluation board examples are placed. The sub-directory contains additional sub-directories organizing evaluation boards by manufacturers. In this case, \Freescale is the manufacturer of the TWR-K53N512 board, and projects for this board are placed under: \TWR-K53N512.

The \Examples\Freescale\TWR-K53N512 sub-directory contains further directories.

\Doc contains a series of schematics and reference manuals:

```
K53 MCU Reference Manual.pdf  
TWR-K53N512 Schematics.pdf  
TWR-ELEV Schematics  
MED-EKG Schematics.pdf  
MED-GLU Schematics.pdf  
MED-SPO2 Schematics.pdf  
MED-BPM Schematics.pdf
```

\BSP contains Board Support Package (BSP) files used to support the peripherals found on the TWR-K53N512 evaluation board. The contents of these files will be described as needed within the sample projects. This sub-directory contains the following files:

```
bsp.c  
bsp.h  
bsp_int.c  
bsp_adc.c  
bsp_dac.c  
bsp_ser.c  
\Freescale\K53X_Flash.icf  
\Freescale\MK53N512CMD100.h  
\OS\uCOS-II\bsp_os.c  
\OS\uCOS-II\bsp_os.h  
\TCPIP\net_bsp_ether.c  
\TCPIP\net_bsp_wifi.c
```

\TWR-K53N512\OS2-TCPPIP-HRM is the directory that contains all the source code files for the Heart Rate Monitor (HRM) application.

\TWR-K53N512\OS2-TCPPIP-BGM is the directory that contains all the source code files for the Blood Glucose Meter (BGM) application.

\TWR-K53N512\OS2-TCPPIP-POX is the directory that contains all the source code files for the Pulse Oximeter (POX) application.

\TWR-K53N512\OS2-TCPPIP-BPM is the directory that contains all the source code files for the Blood Pressure Monitor (BPM) application.

7-1-2 \Software

\UC-CPU

This sub-directory contains the generic and Cortex-M4-specific files for the µC/CPU module. This sub-directory contains the following files:

```
cpu_core.c  
cpu_core.h  
cpu_def.h  
\Cfg\Template\cpu_cfg.h  
\ARM-Cortex-M4\IAR\cpu.h
```

\UC-LIB

This sub-directory contains the source code of the functions used to manipulate ASCII strings, perform memory copies, and more. We refer to these files as being part of the µC/LIB module. `lib_def.h` contains a number of useful `#defines`, such as `DEF_FALSE`, `DEF_TRUE`, `DEF_ON`, `DEF_OFF`, `DEF_ENABLED`, `DEF_DISABLED`, and dozens more. µC/LIB also declares such macros as `DEF_MIN()`, `DEF_MAX()`, `DEF_ABS()`, and more.

This sub-directory contains the following files:

```
lib_ascii.c  
lib_ascii.h
```

```
lib_def.h
lib_math.c
lib_math.h
lib_mem.c
lib_mem.h
lib_str.c
lib_str.h
\Doc\uC-LIB_Manual.pdf
\Doc\uC-LIB-ReleaseNotes.pdf
```

\UCOS-II

This sub-directory contains the full source code of µC/OS-II:

```
\Cfg\Template\os_cfg.h
\Ports\ARM-Cortex-M4\Generic\IAR\os_cpu_a.asm
\Ports\ARM-Cortex-M4\Generic\IAR\os_cpu_c.c
\Ports\ARM-Cortex-M4\Generic\IAR\os_cpu.h
\Source\os_core.c
\Source\os_dbg.c
\Source\os_flag.c
\Source\os_mbox.c
\Source\os_mem.c
\Source\os_mutex.c
\Source\os_q.c
\Source\os_sem.c
\Source\os_task.c
\Source\os_time.c
\Source\os_tmr.c
\Source\ucos_ii.h
```

7-2 DOWNLOADING µC/PROBE

µC/Probe is an application that allows users to display or change the value (at run time) of virtually any variable or memory location on a connected embedded target. See Appendix B, “Micrium’s µC/Probe” on page 275 for a brief introduction.

µC/Probe is used in all of the examples described in Chapter 4, “Introduction to Medical Applications” on page 49 to gain run-time visibility.

The Educational Edition of µC/Probe is available for your evaluation free of charge. It may be used without a license as long the software is used for educational purposes:

<http://micrium.com/tools/ucprobe/software-and-docs/>

When you are satisfied that Micrium’s µC/Probe is the right tool for you, you can purchase µC/Probe online and receive product updates and technical support.

µC/Probe is available for purchase in two editions, Basic and Professional.

µC/Probe Edition	Description
Basic Edition	All features of the free Educational Edition, but with no time limit, and the ability to import/export data screens.
Professional Edition	All features of the Basic Edition, plus: <ul style="list-style-type: none"> • Data Log Control • Microsoft® Excel® Bridge Control • MQTT Client Control • Numeric Up/Down Control • Radio Buttons • Scatter X-Y Chart • Scripting Control • Terminal Window Control • Tree View Control • µC/Trace Trigger Control

Both the Basic and Professional editions of µC/Probe can be purchased with a renewable or permanent license. All license types are locked to a specific computer.

To purchase µC/Probe go to: <http://micrium.com/tools/ucprobe/buy/>

7-3 DOWNLOADING THE IAR EMBEDDED WORKBENCH FOR ARM

Examples provided with this book were tested using the IAR Embedded Workbench for ARM V7.10. The evaluation license of IAR Embedded Workbench is completely free of charge and allows you to try the integrated development environment and evaluate its efficiency and ease of use. The evaluation license is intended for prospective customers to test and evaluate IAR Embedded Workbench.

You have two evaluation options:

- A 30-day time-limited but fully functional evaluation license
- A size-limited Kickstart license without any time limit

The same installer is used for both the 30-day time-limited and the Kickstart (size-limited) evaluation edition. You select which evaluation license type you want to use after the installation. When you start the product for the first time, you will be asked to register to get your evaluation license.

The evaluation license can be upgraded to a standard license of the product when you purchase the product.

Download the installer from the following link and follow the on-screen instructions to install IAR Systems Embedded Workbench for ARM:

<http://supp.iar.com/Download/SW/?item=EWARM-EVAL>

7-4 SETTING UP THE HARDWARE

It is assumed that the following hardware parts are available:

- 1 A PC running Microsoft Windows 7 or 8 (32-bit or 64-bit)
- 2 The TWR-K53N512-KIT Tower System from Freescale
- 3 The MED-EKG module from Freescale for the heart rate monitoring application

- 4 The MED-GLU module from Freescale for the blood glucose meter application
- 5 The MED-SPO2 module from Freescale for the pulse oximetry application
- 6 The MED-BPM module from Freescale for the blood pressure monitor application
- 7 The J-Link debug probe from Segger

Figure 7-2 shows how to connect the TWR-K53N512 to a PC. The tower system is powered from the Mini-B USB connector in J5 of the elevator module to one of the PC's USB ports.

The J-Link debug probe is not only used to download and debug code but also to communicate with µC/Probe. Notice that a 19-pin adapter for ARM Cortex-M from IAR allows JTAG, SWD and SWO connections between J-Link debug probe and Cortex-M microcontrollers. It adapts from the 20-pin 0.1" JTAG connector to the 19-pin 0.05" Samtec FTSH connector in J23 of the TWR-K53N512.

You may need to install the included battery into the battery holder in VBAT (RTC) of the TWR-K53N512 before powering up the system.

The last step to setup the environment to run the examples included in this book is to power up the tower system by turning on the switch at SW1 of the elevator module and let the PC automatically configure all the necessary USB drivers.

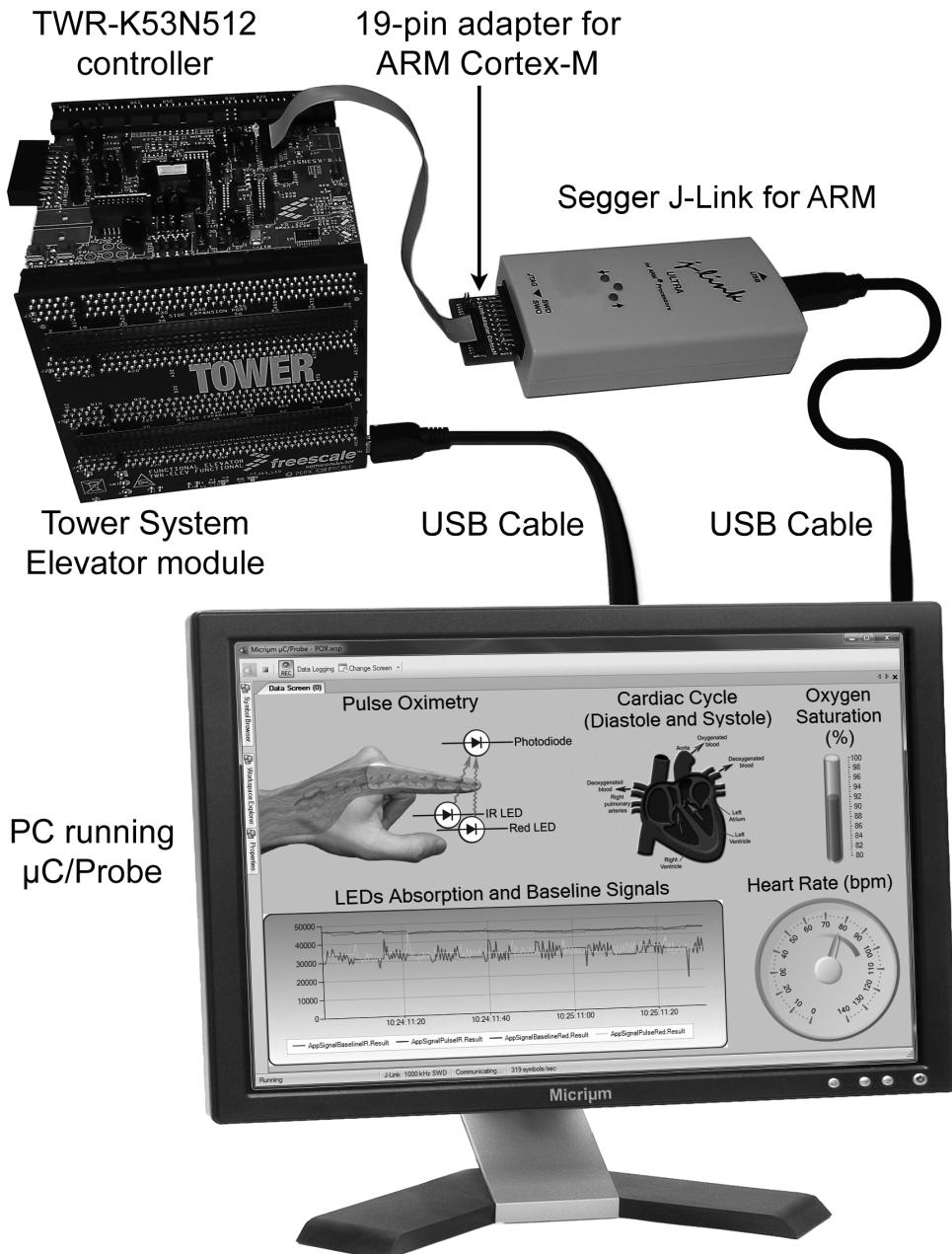


Figure 7-2 Connecting a PC to the TWR-K53N512

7-5 DOWNLOADING THE TWR-K53N512 DOCUMENTATION

You can download the latest TWR-K53N512 datasheets and programming manuals from:

<http://www.freescale.com/TWR-K53N512>

Chapter 8

ECG / Heart Rate Monitor

The *electrocardiogram* (ECG or EKG) is a plot of the electrical activity of the heart over time. It is perhaps the most frequent test used at any emergency room and intensive care unit.

This chapter demonstrates a basic implementation of an ECG-based heart rate monitor built using µC/OS-II and Freescale products.

The chapter starts with an illustrated introduction to some of the anatomical and physiological fundamentals of the heart. The next section continues the introduction, but with emphasis on the acquisition of the ECG signal and presents the design of a heart rate monitor (HRM). The last part of the chapter deals with using the tools to run the example and a description of how the code works.

8-1 THE HEART

The blood carries the oxygen that the organs need to function properly.

The heart is a muscular organ responsible for pumping blood throughout the body. It is located in the middle of the thorax, slightly offset to the left and surrounded by the lungs.

The heart is composed of four chambers; two *atriums* and two *ventricles* as shown in Figure 8-1, and it is the muscular contraction of these chambers what moves the blood throughout the circulatory system as described by the arrows pointing away from the heart and towards the heart in Figure 8-2. The arteries carry blood away from the heart while the veins carry blood towards the heart. The heart valves shown in Figure 8-1 make sure the blood flows through them in only one direction.

Chapter 8

The *aorta* is the largest artery in the body and carries oxygenated blood to all the body. The superior and inferior *venae cavae* are the veins that bring the deoxygenated blood from the body into the heart. The pulmonary arteries and veins allow the exchange of deoxygenated and oxygenated blood between the heart and lungs.

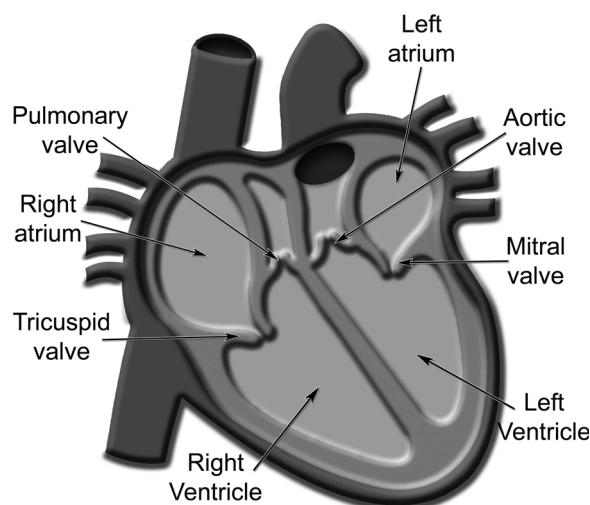


Figure 8-1 Heart chambers and valves

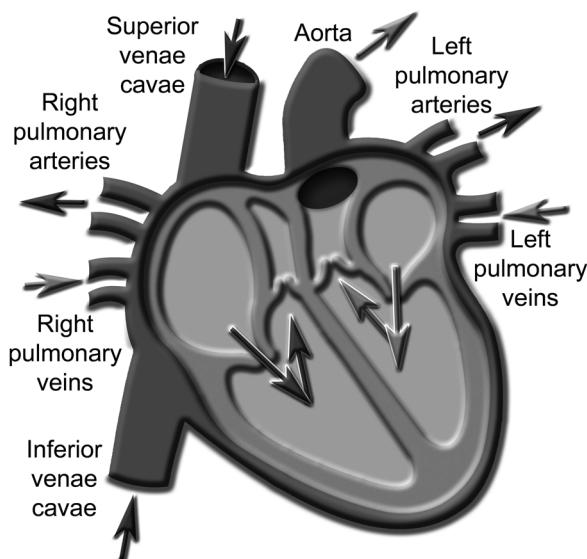


Figure 8-2 Heart arteries and veins

Figure 8-3 shows the principal parts of the electrical system of the heart. The electrical system of the heart is made of nodes and bundles of specialized cells capable of generating, carrying and controlling the electrical impulse that triggers the muscular contraction of the atriums and ventricles.

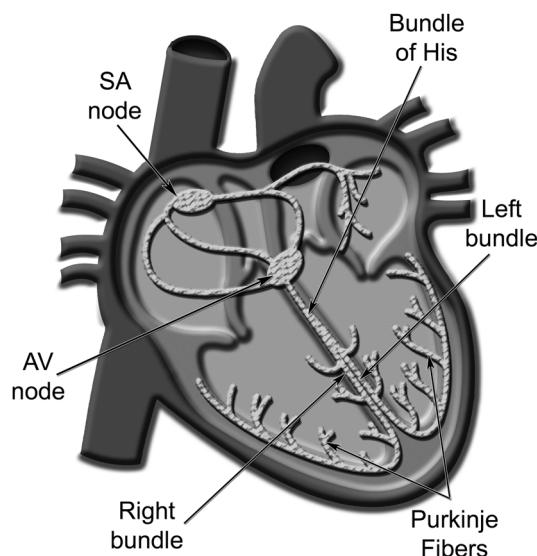


Figure 8-3 Electrical system of the heart

Each part of this electrical system generates its own electrical impulse and the combination of all those impulses make the typical waveform of an ECG as shown in Figure 8-4.

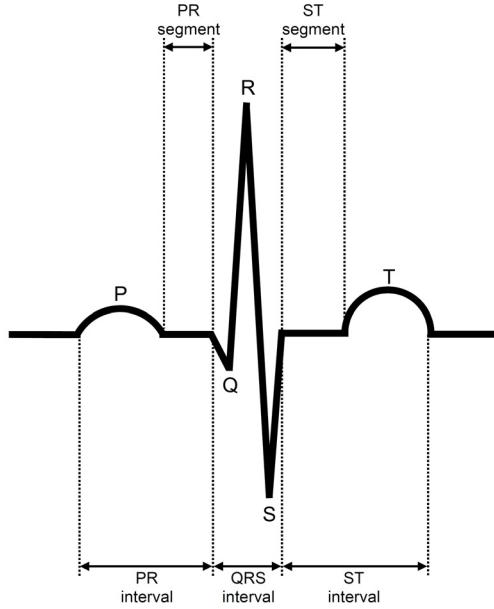
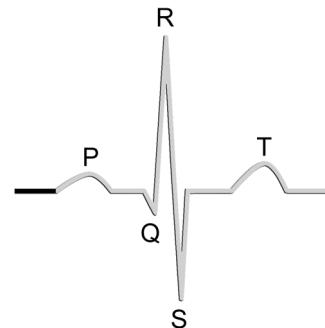
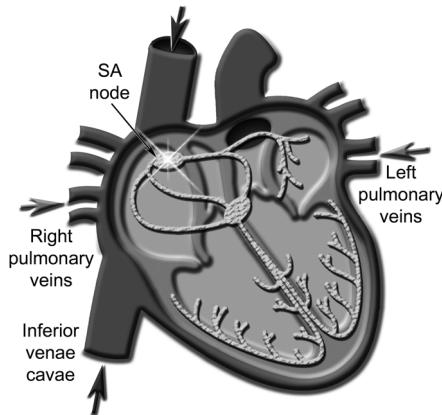


Figure 8-4 Typical ECG waveform

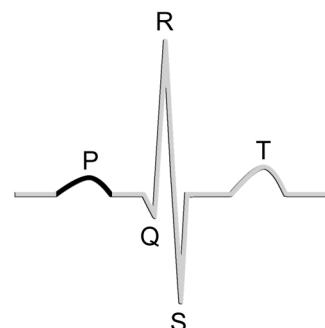
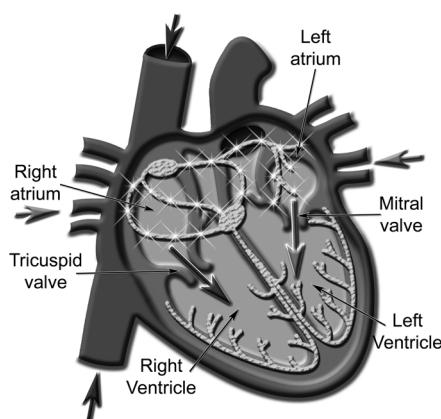
The P wave represents atrium contraction; the Q, R and S waves represent contraction of the ventricles and the T wave the repolarization of the ventricles. These waves and all the rest of time-domain features in the ECG waveform are very important when studying an ECG recording. Cardiologists are particularly interested in any deviations from a normal ECG waveform in the shape of the waves, frequency of the QRS complex, level of the segments and the duration of the intervals and segments.

The electrical activity of the heart is based on the depolarization and repolarization of the cells of the heart and the depolarization of these heart muscle cells trigger the mechanism of muscular contraction by the following coordinated series of repeated events:

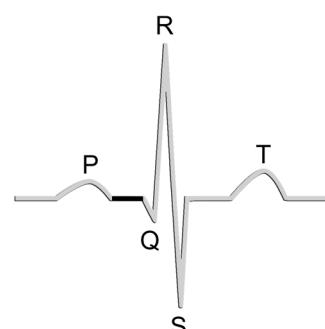
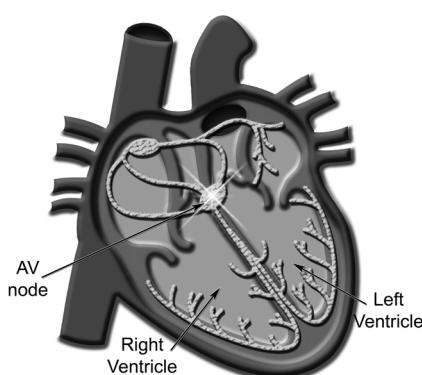
(1)



(2)

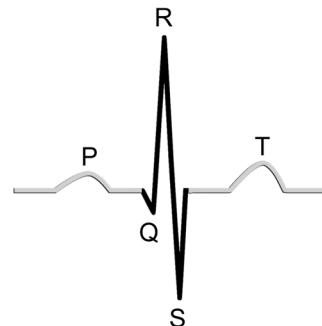
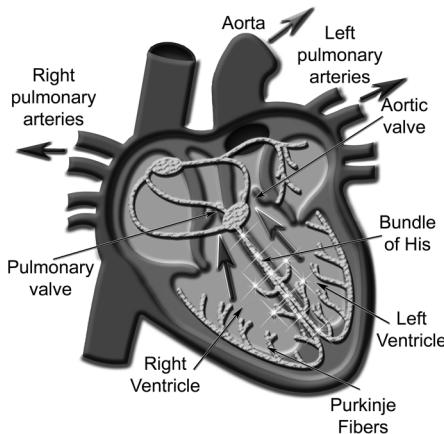


(3)

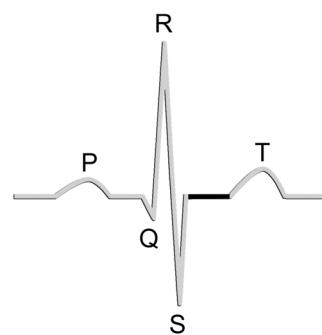
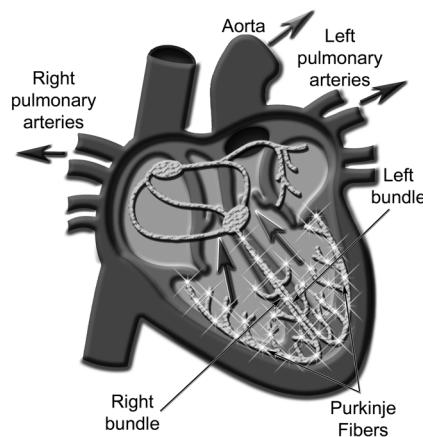


Chapter 8

(4)



(5)



(6)

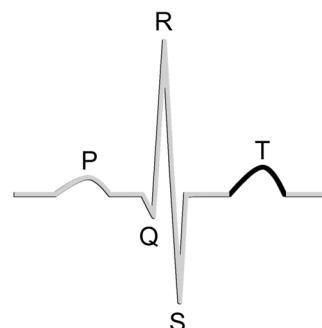


Figure 8-5 Cardiac cycle and corresponding ECG

- F8-5(1) The right atrium keeps filling up with the deoxygenated blood returning to the heart from the whole body through the superior and inferior vena cavae. At the same time the left atrium keeps filling up with the oxygenated blood coming from the lungs through the pulmonary veins. The electrical impulse starts at a specialized bundle of neurons called the *sinoatrial (SA)* node, located at the top of the right atrium. The SA node is considered the natural pacemaker.
- F8-5(2) The electrical impulse flows through the atriums activating the contraction of the right and then the left atrium. The blood passes to the right and left ventricles through the *tricuspid* and *mitral* valves due to the muscular contraction of the right and left atriums respectively. This event can be seen in the ECG waveform in the form of the P wave.
- F8-5(3) The electrical impulse reaches the *atrioventricular (AV)* node where it gets delayed for about 100 ms before making its way through the rest of bundles in order to give the ventricles time to fill with blood. This event can be seen in the ECG in the form of the PR segment.
- F8-5(4) The current flows through a specialized bundle of heart muscle cells known as the *bundle of His* to reach the left and right ventricles and flows through them by the *Purkinje fibers* generating the ventricular contractions.
- The deoxygenated blood is pumped to the lungs through the pulmonary valve and pulmonary artery due to the muscular contraction of the right ventricle.
- The left ventricle contracts and propels the oxygenated blood through the aortic valve and aorta in order to be distributed to the entire body. The QRS complex in the ECG waveform represents this event of ventricular depolarization.
- F8-5(5) Ventricles remain depolarized (contracted) for about 80 ms propelling the blood through the aorta and pulmonary arteries. This can be seen in the ECG waveform in the form of the ST segment.
- F8-5(6) The last event in the cardiac cycle is the repolarization of the heart tissue which is represented by the T wave in the ECG waveform. During this event the atriums and ventricles of the heart are relaxed to allow blood to fill them in and start the cycle again.

This series of events is repeated over and over again and it is known as the *heartbeat*. Normal resting heart rates range from 60 to 100 beats per minute. Even though for well-trained athletes, a normal resting heart rate may be closer to 40 bpm as their hearts work more efficiently.

It is important for the normal operation of the heart for these events to happen one after the other at the same rhythm. Any variation in the ECG waveform means that one or more of these events are not happening correctly and may suggest a cardiovascular disease.

The next section will explain the fundamentals of biopotentials and how to measure them. Then, section 8-3 “ECG Leads” on page 121 will describe all the different configurations used in ECG to measure the electrical activity of the heart as seen from different planes (angles).

8-2 BIOLOGICAL ELECTRICAL POTENTIALS

As we saw in the previous section 8-1 “The Heart” on page 111, the heart cells are a special type of muscle cells due to their cellular nature and function they perform. Their electrochemical activity leads to the contraction of the atriums and ventricles to make the heart pump the blood.

This electrochemical activity generates currents that flow inside the thoracic cavity and their corresponding electrical biopotentials can be measured and recorded at the body surface by using biopotential electrodes.

The electrical biopotentials generated by the heart can be represented as vector quantity. In order to understand the electrical activity of the heart, it is a common assumption that the heart can be represented as a dipole located in the thorax, with a specific polarity at one instance, and inverted polarity in the next. The potential in a specific instance is defined by the amount of charge, and the separation between charges as shown in Figure 8-6.

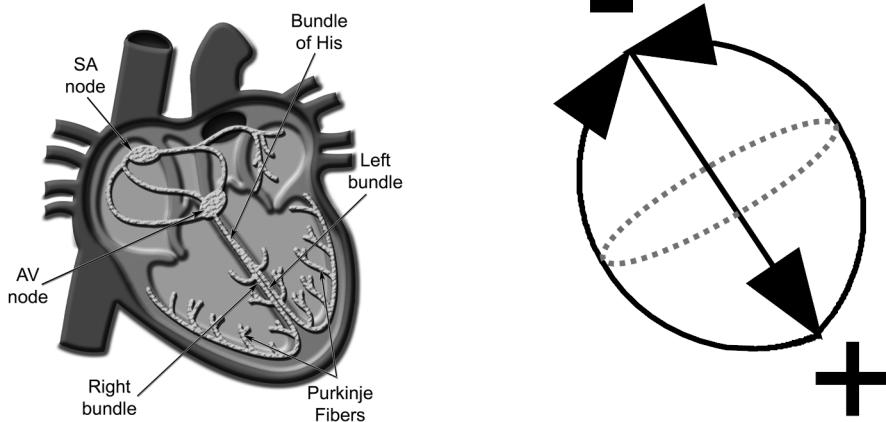
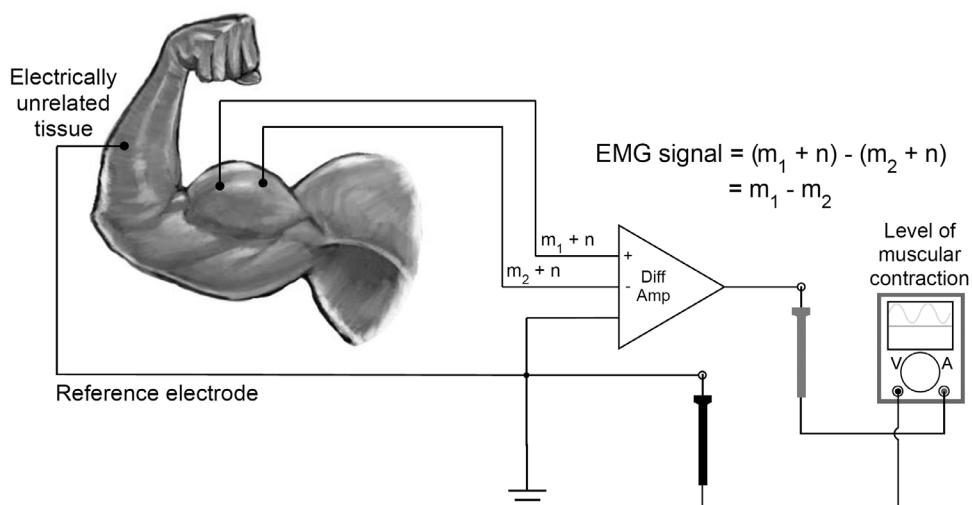


Figure 8-6 Heart represented as an electrical dipole: cardiac vector

In order to explain the fundamentals of medical instrumentation and how to measure the electrical activity of a muscle in general, let's consider Figure 8-7 which shows an example of acquiring signals between two points on a bicep, using biopotential electrodes and a differential operational amplifier.

Figure 8-7 Differential amplification: m is signal from muscle, n is noise

The output of the operational amplifier represents the level of muscular contraction and is the difference of magnitudes of m_1 and m_2 . Each pair of electrodes or an electrode combination is defined as a *lead* and this instrumentation principle is the same for any other muscle including the heart.

In diagnostic ECG, cardiologists are interested in looking at the electrical activity of the heart seen from different angles for a better diagnosis and that is why a typical diagnostic ECG system is comprised of 12-leads. Other ECG systems are only 3-lead or 5-lead and are used for continuous monitoring like those found in an intensive care unit or operation room (see Figure 8-8).



Figure 8-8 Intensive care unit monitor

8-3 ECG LEADS

There are three basic leads used in cardiology and the electrodes are placed on the limbs: Left Arm (LA), Right Arm (RA) and Left Leg (LL) as shown in Figure 8-9:

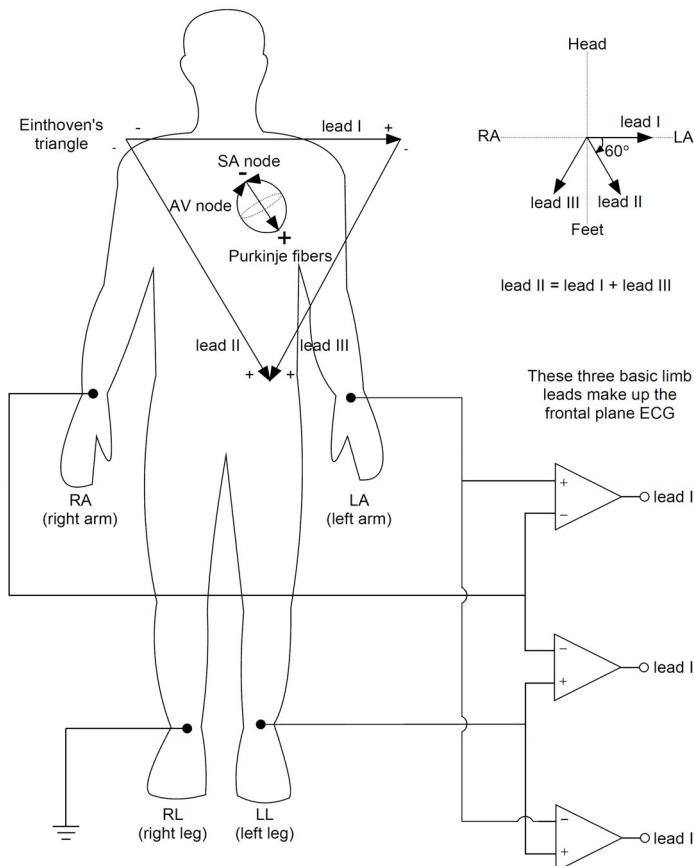


Figure 8-9 Heart as an electrical dipole and the three basic limb leads

Lead I is the vector at 0° and is the voltage between the left arm and right arm electrodes.

Lead II is the vector at 60° and is the voltage between the left leg and right arm electrodes.

Lead III is the vector at 120° and is the voltage between the left leg and left arm electrodes.

The three basic leads make up the frontal plane, which is known as the *Einthoven's triangle*.

The Einthoven's triangle is assumed to be equilateral with the heart at the center which is equivalent to the vector sum of all the electrical activity of the heart going in all directions. Assuming this triangle is equilateral, what is known as the *Einthoven's law* states that:

$$\text{lead I} + \text{lead III} = \text{lead II}$$

Figure 8-10, Figure 8-11 and Figure 8-12 show the result of combining the three basic limb leads and applying the Einthoven's law to create augmented strength versions of the vectors by combining a pair of leads and using them as a reference.

Augmented lead *aVR* represents the view of the heart from the right shoulder to the center of the heart as shown in Figure 8-10:

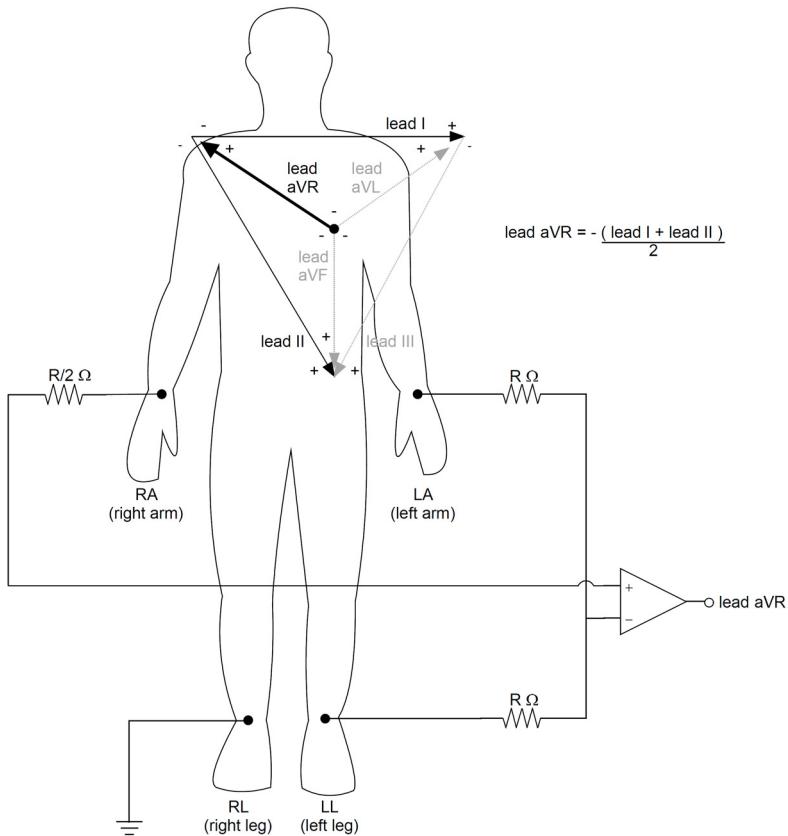


Figure 8-10 Augmented lead aVR

The resistive network is necessary in order to create a reference point between the electrodes such that the voltage at this reference point gets averaged out. This central reference point is known as the *Wilson's central terminal* and corresponds to the center of the heart.

Augmented lead *aVL* represents the view of the heart from the left shoulder to the center of the heart as shown in Figure 8-11.

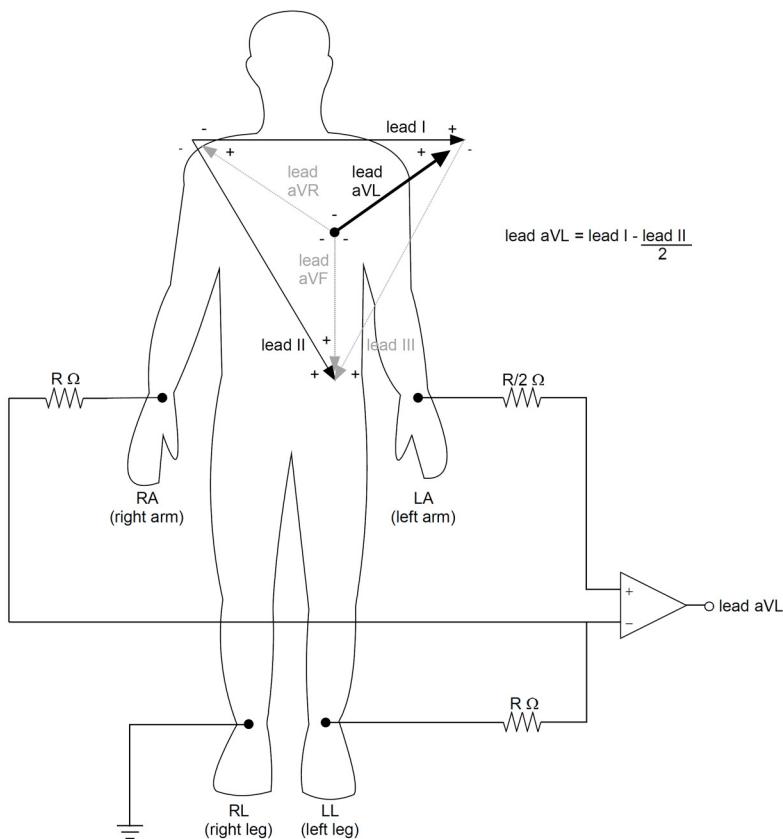


Figure 8-11 Augmented lead *aVL*

The last augmented lead is called *aVF* and it represents the view of the heart from the left hip to the center of the heart. The connections for this lead are shown in Figure 8-12.

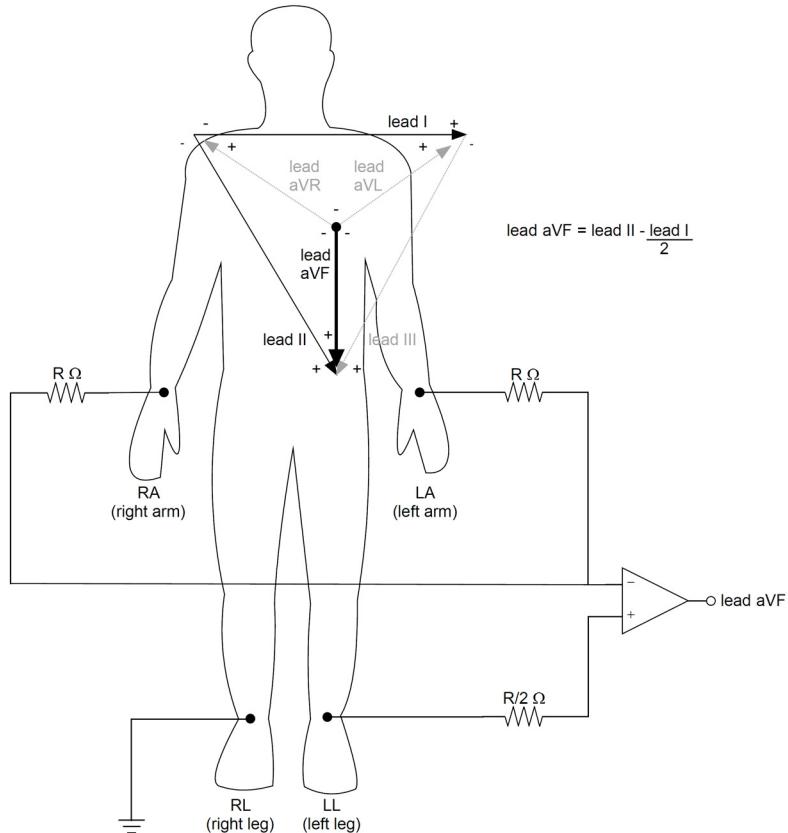


Figure 8-12 Augmented lead aVF

The last group of leads are used in diagnostic ECG when cardiologists want to see the ECG in the horizontal plane and they are called *precordial leads*.

Precordial leads are placed at certain positions on the chest. Each of the 6-precordial leads measures the potential between that specific position on the chest and the Wilson's central terminal as shown in Figure 8-13 which also shows the connections of all the 10 biopotential electrodes necessary to record a 12-lead ECG including the 3-basic limb leads (I, II and III), 3-augmented limb leads (aVR, aVL and aVF) and the 6-precordial leads (V1, V2, V3, V4, V5 and V6).

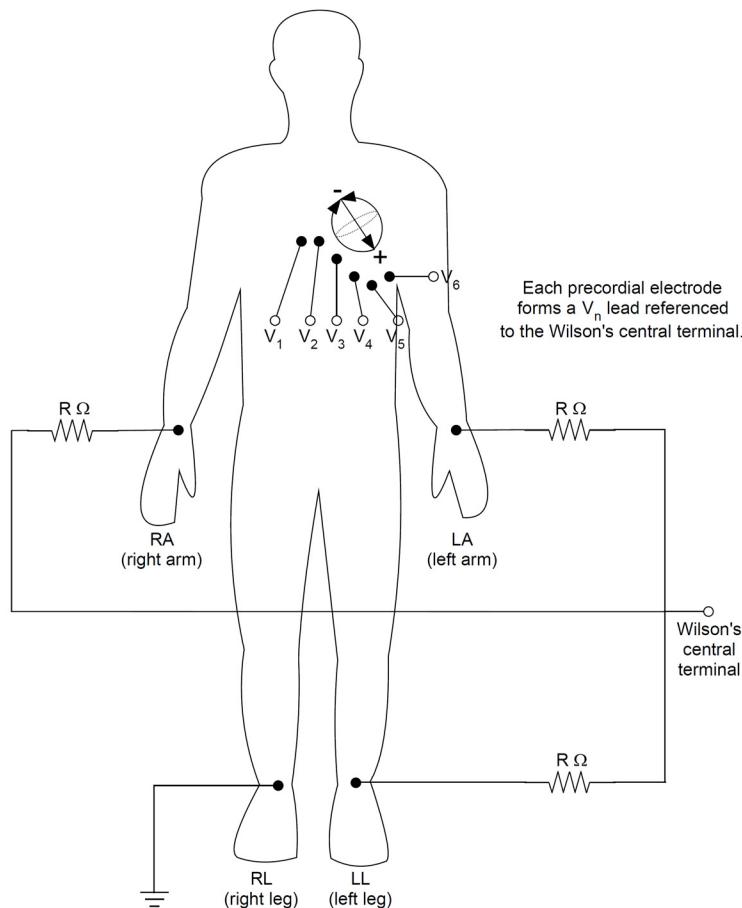


Figure 8-13 Precordial leads

Figure 8-14 shows an example of what is known as a 12-lead ECG strip of a healthy patient at rest that features 12 different angles of view of the heart.

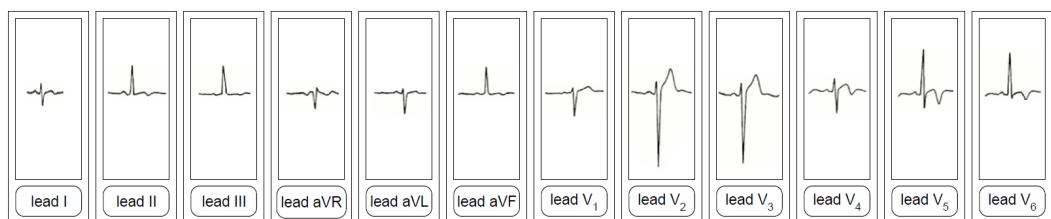


Figure 8-14 12-lead ECG strip

8-4 CARDIOVASCULAR DISEASES (CVD)

Cardiovascular diseases or CVDs are a group of diseases of the heart and blood vessels that include:

- Coronary heart disease: disease of the blood vessels supplying the heart muscle.
- Cerebrovascular disease: disease of the blood vessels supplying the brain.
- Peripheral arterial disease: disease of the blood vessels supplying the arms and legs.
- Rheumatic heart disease: damage to the heart muscle and heart valves from inflammatory fever, caused by streptococcal bacteria among others.
- Congenital heart disease: malformations of heart structure existing at birth.
- Deep vein thrombosis and pulmonary embolism: blood clots in the leg veins, which can dislodge and move to the heart and lungs.

Heart attacks and strokes are usually acute events and are mainly caused by a blockage that prevents blood from flowing to the heart or brain. The most common reason for this is a build-up of fatty deposits on the inner walls of the blood vessels that supply the heart or brain. Strokes can also be caused by bleeding from a blood vessel in the brain or from blood clots.

According to the World Health Organization (WHO) and their fact sheet #317 of January 2011 these are some of the key facts regarding CVDs:

- CVDs are the number one cause of death globally: more people die annually from CVDs than from any other cause.
- An estimated 17.1 million people died from CVDs in 2004, representing 29% of all global deaths. Of these deaths, an estimated 7.2 million were due to coronary heart disease and 5.7 million were due to stroke.
- Low- and middle-income countries are disproportionately affected: 82% of CVD deaths take place in low- and middle-income countries and occur almost equally in men and women.

It is well known that heart disease and stroke can be prevented through a healthy diet and physical activity but the fact that over 80% of deaths by cardiovascular diseases take place in low- and middle-income countries calls for efforts towards not only making ECG more affordable but also developing intelligent ECG algorithms to help doctors in the diagnosis in those countries where their availability can be limited.

Since survivors of a heart attack or stroke are at high risk of recurrences and at high risk of dying from them, making more affordable and intelligent ECG systems would also make them able to target the home market in developed countries. Many people at risk would find a portable ECG system very convenient if it allows them to still keep taking care of themselves without taking time off of their daily routines.

This type of portable home medical device needs low power consumption, high data processing and a wired or wireless communication interface. Freescale's Kinetis MCUs enable the development of such devices by combining the latest low-power innovations and high performance, high precision mixed-signal capabilities with a broad range of connectivity, human-machine interface, and safety and security peripherals. Kinetis MCUs are supported by Micrium's real time kernel and this example application makes an ideal starting point for a new medical application.

8-5 ECG DESIGN

The preceding sections described the fundamentals of the heart, its electrical activity, how to measure the activity with skin electrodes and how important it is for physicians to have the ECG as a tool to make the right diagnosis for all the different cardiovascular diseases.

The next sections will describe the design of an Electrocardiograph and then we will take a piece of that design in order to implement an ECG-based heart rate monitor, including hardware and software.

When it comes to medical instrumentation, the signal acquisition is the first consideration. In the case of an ECG, the electrodes on the skin detect the small voltages generated by the heart activity, but the signal is weak and contains a lot of ambient noise.

First it is necessary to amplify the signal and filter the noise, so we can extract the ECG signal.

Noise and interference signals acquired by this type of systems are due to:

- The electric power system (50Hz / 60Hz interference).
- The common-mode voltage which is usually even higher than the ECG signal itself.
- Muscle contraction of anything other than the heart.
- Respiration.
- Electromagnetic interference (EMI) and electromagnetic emissions (EME) from electronic components.

Most of the interference signals can be minimized by installing a proper grounding system in the room where the ECG is to be located, proper training of medical personnel on ECG and periodic maintenance of the power system and medical equipment. But probably one of the most important factors is the quality in the design of the ECG system.

Figure 8-15 shows the block diagram of a typical ECG signal conditioning system. The right leg electrode is connected to the system in order to minimize the effects of the common mode voltage by making adjustments to the baseline according to changes in the potential of this electrode.

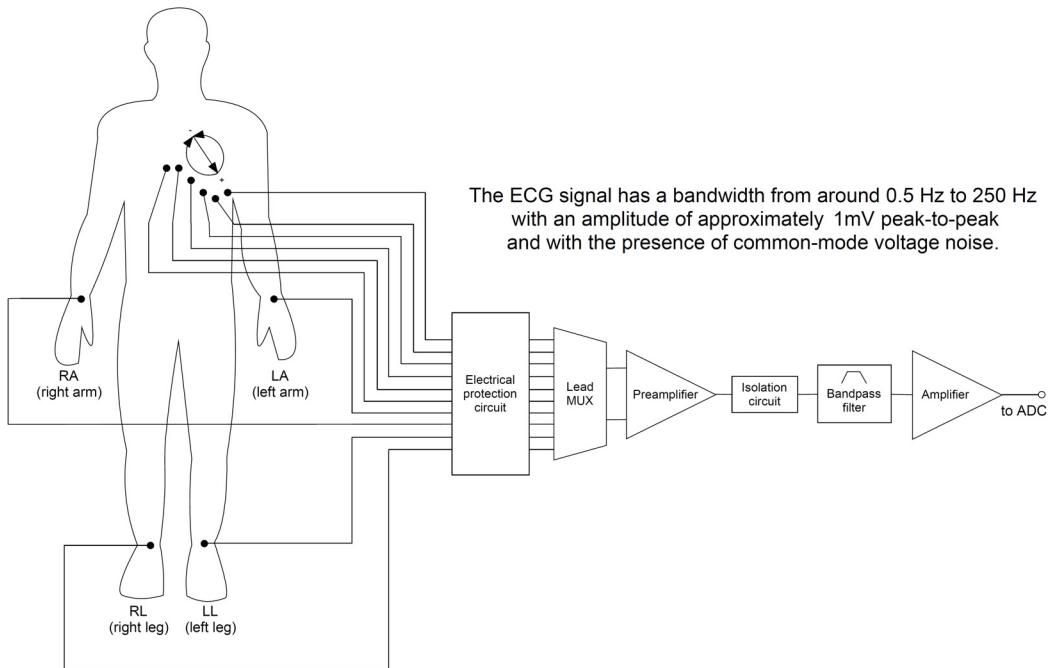


Figure 8-15 Block diagram of a typical ECG signal acquisition and conditioning system

Notice that the design only has a single analog channel for 12-leads and the lead MUX is used to select the lead to be recorded. Other designs feature 12 different analog channels to make an ECG system capable of recording all 12-leads simultaneously.

The example presented in this chapter is based on a single channel ECG module offered by Freescale that takes all these factors in consideration to make a high quality ECG signal conditioning system that is compatible with their popular tower development system. The MED-EKG module is designed to work with the TWR-K53N512 controller which is based on their latest Kinetis ARM® Cortex™-M4 MCU.

Figure 8-16 shows the block diagram of this heart rate monitor:

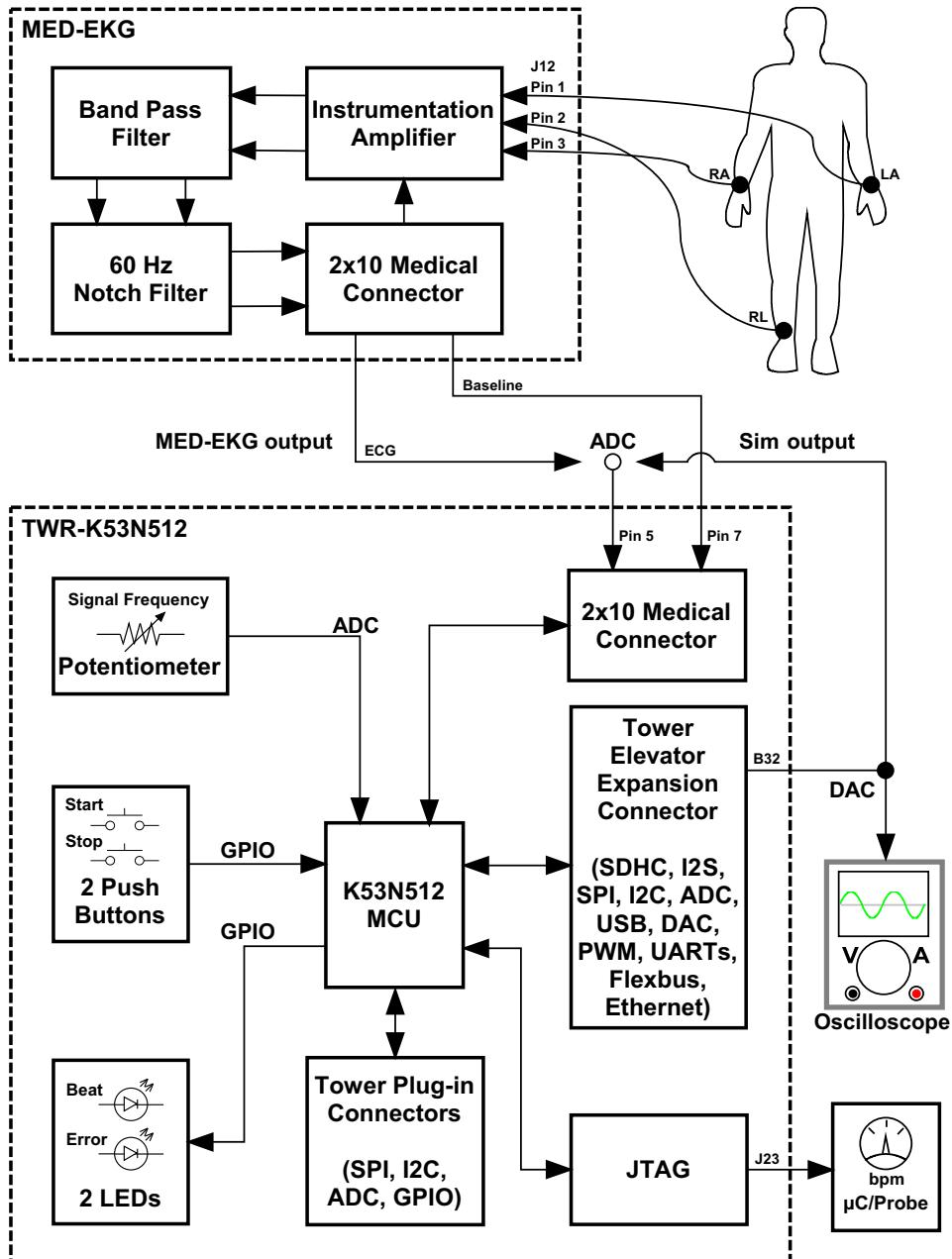


Figure 8-16 Heart Rate Monitor Block Diagram

The two pushbuttons onboard the TWR-K53N512 are used to either start or stop the application and the LEDs are used to indicate an error by turning on the orange LED or to indicate every heartbeat by blinking the yellow LED.

The 2x10 medical connector of the MED-EKG not only provides connectivity with the TWR-K53N512 board to make the ECG and Baseline signals available to the ADC modules but also allows the use of the on-chip op-amps and transimpedance amplifiers of the Kinetis 32-bit microcontroller to implement the required signal conditioning. The rest of the signals have been omitted from the block diagram for the sake of simplicity but Table 8-1 shows the signal present in each pin of the medical connector.

MED-EKG Signal	Pin		MED-EKG Signal
VDD	1	2	GND
ECG_IIC_SDA	3	4	ECG_IIC_SCL
ECG Signal	5	6	GND
Baseline Signal	7	8	DAC
OPAMP0 VOUT0	9	10	OPAMP1 VOUT1
OPAMP0 INN0-	11	12	OPAMP1 INN1-
OPAMP0 INP0+	13	14	OPAMP1 INP1+
TRIAMP0 INP0+	15	16	TRIAMP1 INP1+
TRIAMP0 INN0-	17	18	TRIAMP1 INN1-
TRIAMP0 VOUT0	19	20	TRIAMP1 VOUT1

Table 8-1 Medical Connector 2x10 Pin Header Connections

Notice from the block diagram in Figure 8-16 that in the absence of the MED-EKG module you can still run the application by using the simulated ECG signal coming out of the DAC module by connecting a jumper wire between pin 5 of the medical connector and pin B32 of the elevator module in the tower system. The potentiometer is used in such situation to increase or decrease the simulated heart rate.

Chapter 8

The communication with µC/Probe which will be used to display the results of the heart rate monitor is via the J-Link debug probe from Segger through the JTAG connector located in J23.

The signal conditioning circuit onboard the MED-EKG module features a typical instrumentation amplifier, which is a standard differential amplifier with its two inputs buffered by two op-amps in an inverting amplifier configuration. Having the inputs buffered by these two extra op-amps provide high input impedance and high common-mode rejection to those large 60Hz voltages that exist on the body.

The band-pass filter passes all frequencies between 0.5Hz and 250Hz which eliminates the DC offset and the notch filter attenuates the 60Hz noise. The last high-pass filter and low-pass filter provide the last stage of amplification. The signal before this final stage is the baseline signal while the signal after this final stage is the actual ECG signal.

The firmware needs to monitor the baseline signal to make sure the gain is not too high such that it saturates the amplifier's output and the compensation signal is used to close the loop and provide a positive feedback for these adjustments as shown in Figure 8-17.

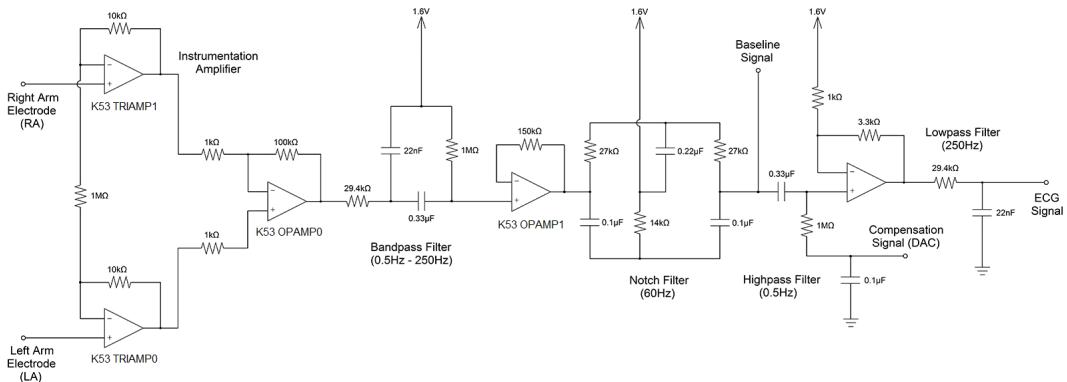


Figure 8-17 MED-EKG Signal Conditioning Circuit

Notice that the labels OPAMP0, OPAMP1, TRIAMP0 and TRIAMP1 in Table 8-1 and Figure 8-17 indicate the on-chip internal operational and transimpedance amplifiers in the Kinetis MCU.

Figure 8-18 shows the actual MED-EKG module connected to the TWR-K53N512 controller in a tower system configuration.

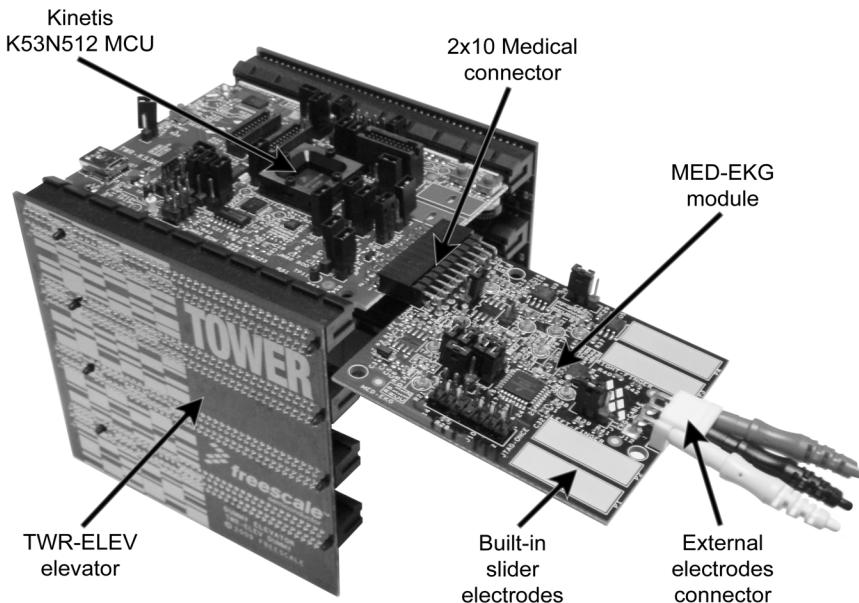


Figure 8-18 TWR-K53N512 and MED-EKG

The MED-EKG is powered by default through connector J1, but can also be powered by the JTAG-ONCE port (do not apply both power sources at the same time).

The power switch is controlled by the TWR-K53N512 pin PTE7 from the corresponding Tower System MCU module. To turn on the MED-EKG module in your software, make sure to set PTE7 pin to output low (active low). To turn off, set PTE7 to output high. This has already been done in the MED-EKG example software in the TWR-K53N512.

More information about the MED-EKG including user manual and schematics can be downloaded from Freescale's website at www.freescale.com by searching for part number MED-EKG.

8-6 RUNNING THE EXAMPLE PROJECT

This section describes the steps you need to follow to run the heart rate monitoring example. Start by connecting the MED-EKG module to the TWR-K53N512 controller through the 2x10 medical connector as shown in Figure 8-18.

Before powering up the system, make sure the jumper settings in the MED-EKG module are set as shown in Table 8-2.

The default installed jumper settings are shown in bold.

Jumper	Option	Setting	Description
J1	Medical board connector	open	Connection with TWR-K53N512 medical board
J2	Connection of MM OPAMP2 INP2-	1-2	Directly to Vref 1.6V
		2-3	Trough R12 to OUT2
J3	Right electrode gain (must be the same as J4)	1-2	Gain is 100x
		2-3	Gain is 10x
J4	Left electrode gain (must be the same as J3)	1-2	Gain is 100x
		2-3	Gain is 10x
J5	Enabling R17 connection	open	R17 connection is open
		shunt	R17 is connected to Vref1.6V
J6	Right electrode connection (must be the same as J7)	1-2	Right electrode is connected to external INA input (INstrumentation Amplifier)
		2-3	Right electrode is connected to internal amplifiers
J7	Left electrode connection (must be the same as J6)	1-2	Left electrode is connected to external INA input (INstrumentation Amplifier)
		2-3	Left electrode is connected to internal amplifiers
J8	ADC input to DSC	1-2	DSC is feed from internal amplifiers output
		2-3	DSC is feed from signal selected in J9

Jumper	Option	Setting	Description
J9	Signal selector jumper	1-2	Selected signal is the external INA circuit
		2-3	Selected signal is the output of internal amplifier OUT2
J10	JTAG-ONCE header	open	It is used to program the DSC
J11	Reference electrodes selector	1-2	Reference electrodes are connected to ground
		2-3	Reference electrodes are connected to Vref1.6V
J12	External electrodes connector	open	It is used to connect external electrodes

Table 8-2 MED-EKG Jumper Settings Table

Notice from Table 8-2 that the MEG-EKG is configured by default to use Kinetis on-chip analog modules. The option that uses external Instrumentation Amplifier (INA) is not functional by default.

You should also notice that when J3 and J4 are set to 100x, you must use external electrodes via J12 instead of using the on-board slider contacts. This is because the on-board slider contacts yield more noise due to the non-secure connection with the finger tips and any noise at the first stage of instrumentation where high amplification occurs can easily result in a saturated output so you will not be able to see the ECG output in the final amplification stage.

Even though the on-board slider electrodes can be used when external electrodes are not available, we strongly recommend the use of standard medical grade external electrodes available from any medical supplies store. These electrodes have a conductive adhesive gel that ensures good skin contact and are shown in Figure 8-19:

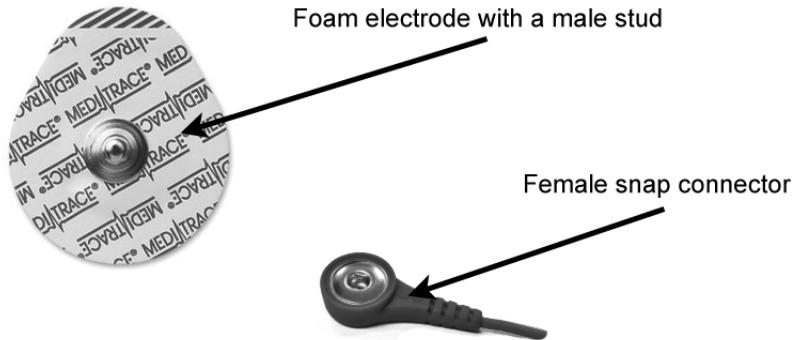


Figure 8-19 **Medical grade external electrode and connector**

According to the American Heart Association, the arm electrodes may be placed on any part of the arms as long as they are below the shoulders and the left leg electrode may be placed on any part of the left leg as long as it is below the waist. We recommend placing the red wire electrode on the right wrist, the white wire electrode on the left wrist and the black wire electrode on the left side abdominal area or on the left ankle as shown in Figure 8-20:

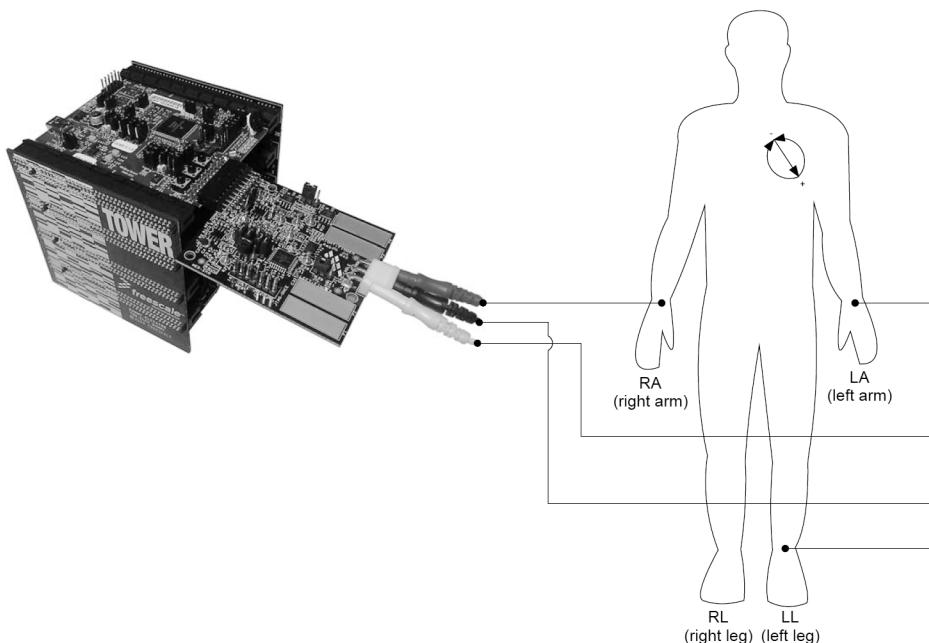


Figure 8-20 **External electrodes connection example**

Assuming you have followed all the setup steps described in Chapter 7, “Setup” on page 101, proceed to connect one end of the Segger’s J-Link for ARM debug probe to the JTAG connector on J23 of the TWR-K53N512 and the other end of the probe to any USB port available in your PC.

Run IAR’s Embedded Workbench for ARM (EWARM) and open the workspace at:

```
$\Micrium\Examples\Freescale\TWR-K53N512\OS2-TCPIP-HRM\IAR\OS2-TCPIP-HRM.eww
```

The workspace window shows all of the files in the project which are sorted into groups represented by folder icons. Figure 8-21 shows the project files for OS2-TCPIP-HRM in the workspace explorer window.

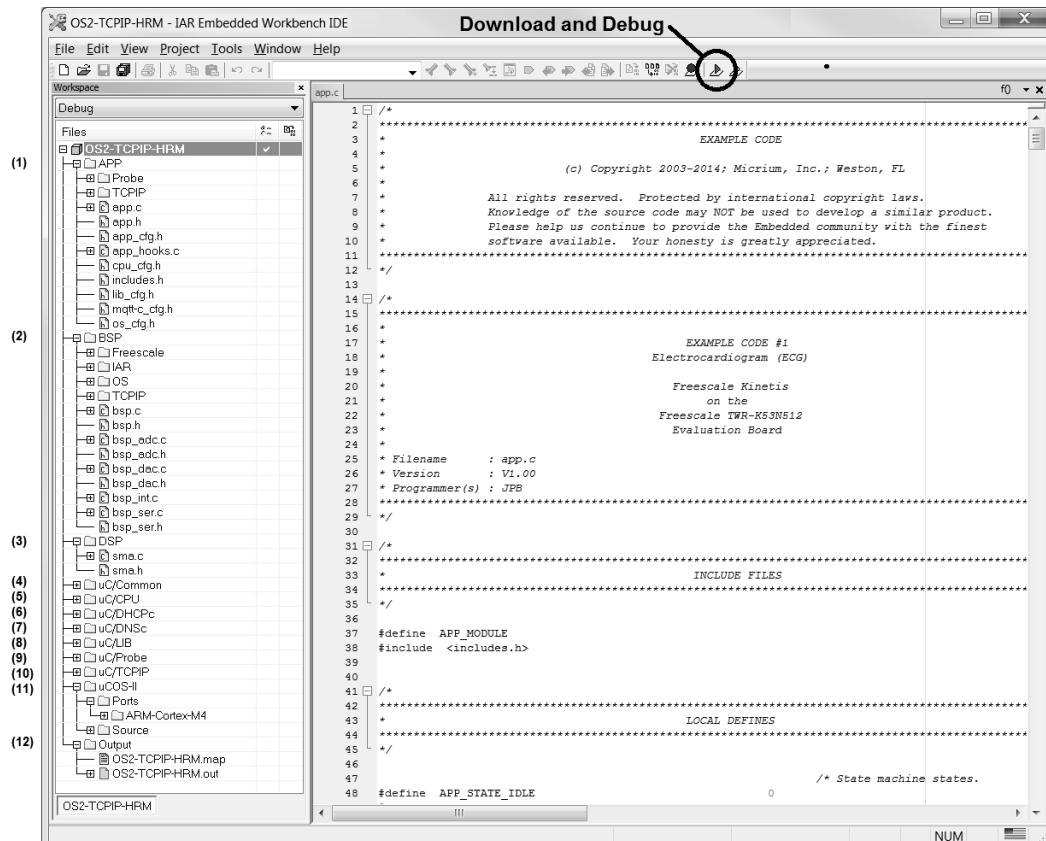


Figure 8-21 Running the project with EWARM

- F8-21(1) The **APP** group includes all of the application files for this example, including the header files used to configure the application.
- F8-21(2) The **BSP** group contains the files that comprise the board support package. The board support package includes the code used to control the peripherals on the board. For this example, the software to control the LEDs, pushbuttons, ADCs, DACs, operational amplifiers and transimpedance amplifiers will be used. The subgroup Freescale includes the header files for the Kinetis K50 family of microcontrollers.
- F8-21(3) The **DSP** group contains the files that define some of the Digital Signal Processing functions called by the application.
- F8-21(4) The **uC/Common** group contains the kernel abstraction layer.
- F8-21(5) The **uC/CPU** group contains the μ C/CPU source code files.
- F8-21(6) The **uC/DHCPC** group contains the header files for the DHCP client that allows you to negotiate an IP address with your network's router.
- F8-21(7) The **uC/DNSc** group contains the header files for the DNS client that allows you to translate domain names into IP addresses.
- F8-21(8) The **uC/LIB** group contains the μ C/LIB source code files.
- F8-21(9) The **uC/Probe** group contains the source code to enable communication with μ C/Probe via TCP/IP.
- F8-21(10) The **uC/TCP-IP** group contains the basic header files for the TCP/IP stack.
- F8-21(11) The **uC/OS-II** group contains the μ COS-II source code.
- F8-21(12) The **Output** group contains the files generated by the compiler/linker.

Start the debugger by clicking the “Download and Debug” icon as shown in Figure 8-21. The application is programmed to the internal Flash of the K53N512 microcontroller and the debugger starts. The code automatically starts executing and stops at the `main()` function in the `app.c` file.

Click the “Go” icon in IAR Embedded Workbench’s debugging tools to run the application as shown in Figure 8-22.

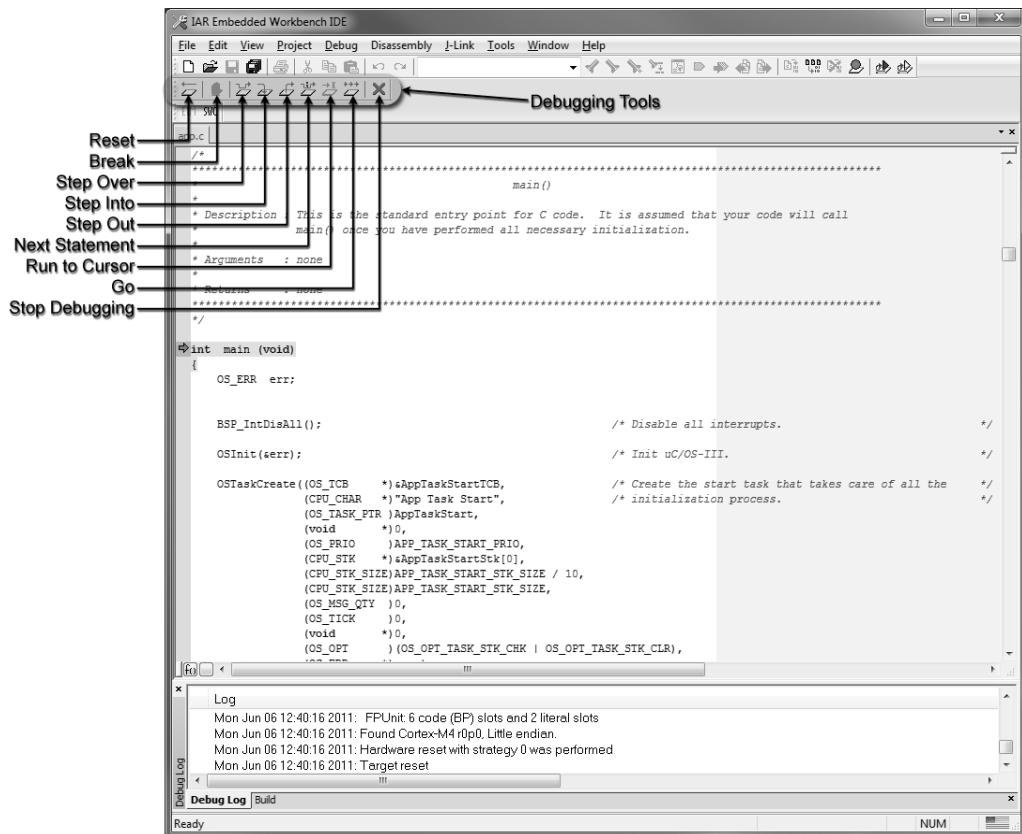


Figure 8-22 Downloading the Application and Starting the Debugger

You can test the application by starting μC/Probe and open the workspace for the heart rate monitor at:

```
$\Micrium\Examples\Freescale\TWR-K53N512\OS2-TCPIP-HRM\IAR\
OS2-TCPIP-HRM.wspx
```

After μC/Probe starts, click the run button (green triangle). Once μC/Probe is running, you will see a screen similar to the one shown in Figure 8-23.

Chapter 8

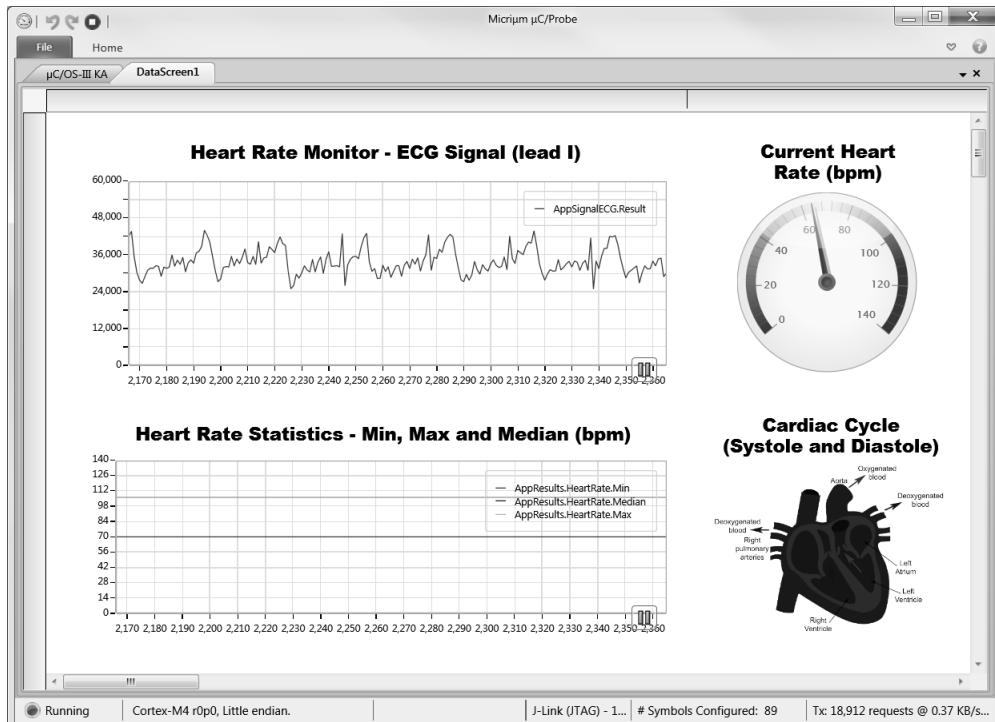


Figure 8-23 µC/Probe Heart Rate Monitor Dashboard

Press the pushbutton on the TWR-K53N512 board labeled as SW1 to start the application and you will start seeing the ECG tracing on the dashboard. The heart rate in beats-per-minute is calculated in real-time and displayed in the gauge. Other heart rate statistics are plotted in one of the charts as shown in Figure 8-23.

8-7 HOW THE CODE WORKS

The heart rate monitoring application consists of four different tasks:

- User IF task: The user interface task monitors the state of the two pushbuttons onboard the TWR-K53N512. The pushbuttons are used to either start or stop the application. The task is defined by `AppTaskUserIF()` in `app.c`.
- Sim task: The simulator task monitors the state of the potentiometer and updates the frequency of an ECG simulated signal generated by the DAC according to the position of the potentiometer. The task is defined by `AppTaskSim()` in `app.c`.
- DAQ task: The data acquisition task is the main task in the application and implements the state machine that processes the analog input samples to calculate the heart rate. The task is defined by `AppTaskDAQ()` in `app.c`.
- Heartbeat task: The heartbeat task is responsible for controlling the heart animation in `µC/Probe` by updating the value of the global variable `AppCardiacCycleState` that can take one of three states: Diastole, Systole or None. The heartbeat task is defined by `AppTaskHeartbeat()` in `app.c`

The interaction among the different tasks is facilitated by the use of µC/OS-II's semaphores and message queues and it is illustrated in Figure 8-24..

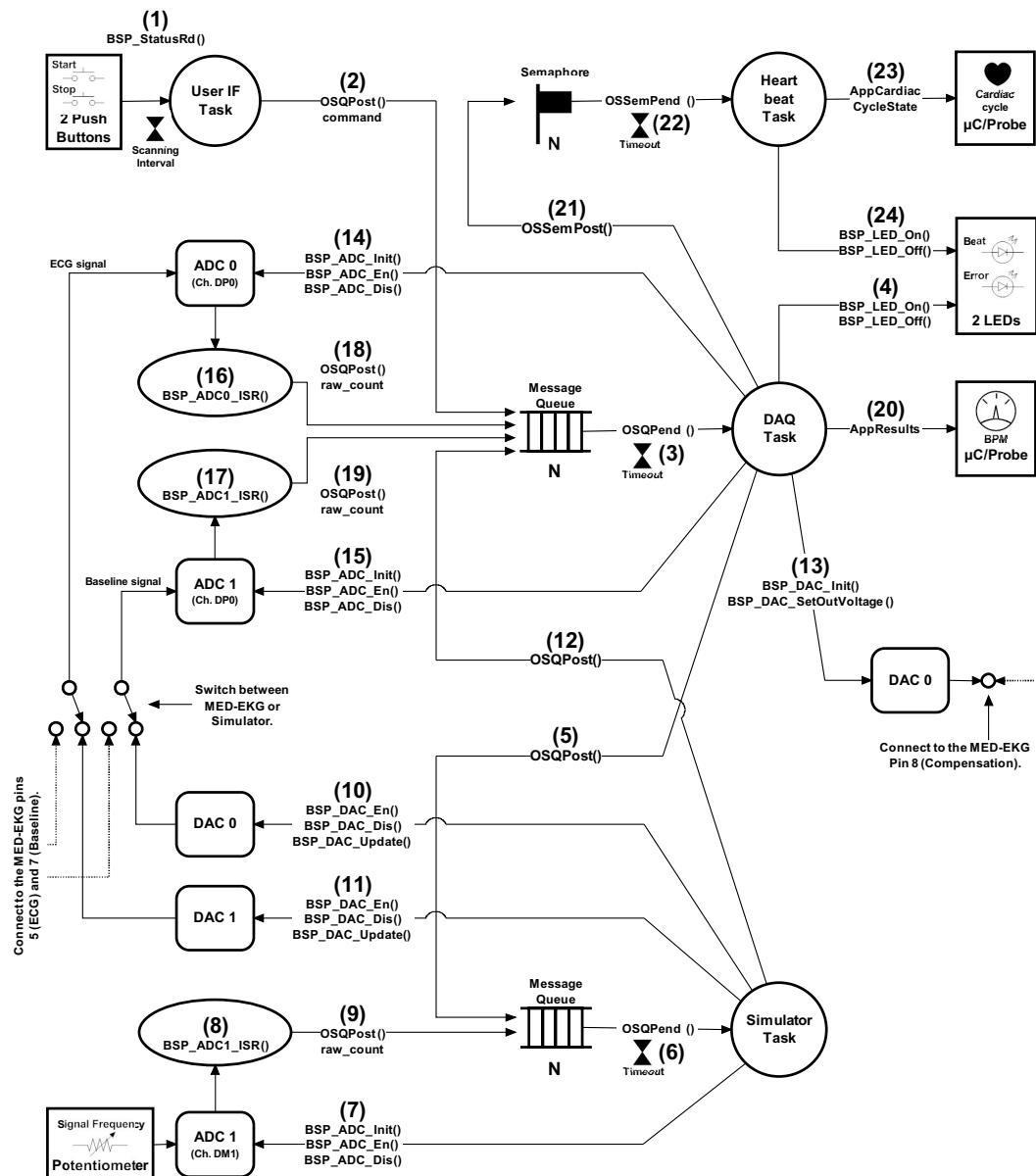


Figure 8-24 Interaction of Application Tasks

-
- F8-24(1) The state of the push buttons is monitored by the user IF task at a scanning interval defined by `BSP_STATUS_CHECK_INTERVAL`. See `AppTaskUserIF()` in `app.c` and `BSP_StatusRd()` in `bsp.c`.
- F8-24(2) Depending on the pushbutton pressed by the user a message from the user IF task to either start or stop the application is posted to the DAQ task's built-in message queue by calling `OSTaskQPost()`. See `AppTaskUserIF()` in `app.c`.
- F8-24(3) The DAQ task can receive data messages directly from the ADCs ISRs or command messages from the other two tasks. When `OSTaskQPend()` is called, the message is retrieved and processed in two different ways depending on the sender and the type of message. If the message comes from one of the ADC's ISR then the message is processed by calling the function `AppTaskDAQ_ProcessData()` and if the message comes from any other task then the message is processed by calling the function `AppTaskDAQ_ProcessCmd()` in `app.c`.
- F8-24(4) If the message retrieved by the DAQ task is a command to start the heart rate monitoring then the DAQ task turns off the LEDs by calling the function `BSP_LED_Off()` in `bsp.c`.
- F8-24(5) If the message retrieved by the DAQ task is a command to start the heart rate monitoring in simulation mode then the DAQ task posts a message to the sim task's built-in message queue to start the simulator. See `AppTaskDAO()` in `app.c` and `AppTaskDAO_ProcessCmd()` in `app.c`.
- F8-24(6) In similar fashion, the sim task can receive messages directly from the ADC1's ISR or from the DAQ task. When `OSTaskQPend()` is called, the message is retrieved and processed in different ways depending on the sender. The sim task retrieves the received message from its message queue and proceeds according to the command issued by the DAQ task as follows:
- F8-24(7) If the message contains a start command then the sim task starts ADC1 channel DM1 in order to read the potentiometer. See `AppTaskSim()` in `app.c`.
- F8-24(8) The CPU is interrupted by the completion of a conversion of ADC1 Channel DM1 and the interrupt is handled by `BSP_ADC1_ISR()` in `bsp_adc.c`.

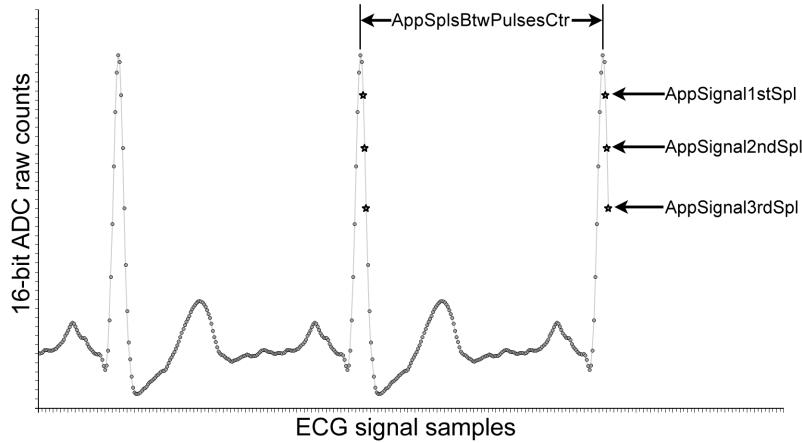
- F8-24(9) The ADC1 ISR posts the result of the conversion in the form of a 16-bit raw count to the sim task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC1_ISR()` in `bsp_adc.c`.
- F8-24(10) The sim task retrieves the message from the queue and configures the DAC0 to output a simulated ECG baseline waveform. See `AppTaskSim()` in `app.c`.
- F8-24(11) The sim task retrieves the message from the queue and configures the DAC1 to output a simulated ECG waveform at a rate proportional to the raw count of the ADC1 channel DM1. See `AppTaskSim()` in `app.c`.
- F8-24(12) The sim task posts a message to the DAQ task's built-in message queue notifying the DAQ task that the simulator is running. See `AppTaskSim()` in `app.c`.
- F8-24(13) The DAQ task retrieves the message from the queue and initializes DAC0 to output a voltage reference. See `AppTaskDAQ()` and `AppTaskDAQ_ProcessCmd()` in `app.c`.
- F8-24(14) The DAQ task retrieves the message from the queue and enables ADC0 channel DP0 to convert the ECG signal. See `AppTaskDAQ()`, `AppTaskDAQ_ProcessCmd()` and `AppTaskDAQ_ExecCmd()` in `app.c`.
- F8-24(15) The DAQ task retrieves the message from the queue and enables ADC1 channel DP0 to convert the Baseline signal. See `AppTaskDAQ()`, `AppTaskDAQ_ProcessCmd()` and `AppTaskDAQ_ExecCmd()` in `app.c`.
- F8-24(16) The CPU is interrupted by the completion of a conversion of ADC0 channel DP0 and the interrupt is handled by `BSP_ADC0_ISR` in `bsp_adc.c`.
- F8-24(17) About the same time, the CPU is interrupted by the completion of a conversion of ADC1 channel DP0 and the interrupt is handled by `BSP_ADC1_ISR` in `bsp_adc.c`.
- F8-24(18) The ADC0 ISR posts the result of the conversion in the form of a 16-bit raw count to the DAQ task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC0_ISR()` in `bsp_adc.c`.

-
- F8-24(19) The ADC1 ISR posts the result of the conversion in the form of a 16-bit raw count to the DAQ task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC1_ISR()` in `bsp_adc.c`.
- F8-24(20) The DAQ task retrieves the message from the queue and uses the raw counts from ADC0 and ADC1 to calculate the heart rate and display the value in μ C/Probe by calling the function `AppTaskDAQ_ProcessData()` and `AppCalcResults()` in `app.c`.
- F8-24(21) The DAQ task sends a signal to the heartbeat task through its local semaphore by calling the function `OSTaskSemPost()`. The signal is to notify the heartbeat task that a new heart rate has been calculated. See `AppCalcResults()` in `app.c`.
- F8-24(22) The heartbeat task normally waits for a signal from the DAQ task in order to update a timer that drives the cardiac cycle animation in μ C/Probe. See `AppTaskHeartbeat()` in `app.c`.
- F8-24(23) The heartbeat task updates the global variable `AppCardiacCycleState` toggling between diastolic and systolic states at the same rate calculated by the DAQ task. See `AppTaskHeartbeat()` in `app.c`.
- F8-24(24) The heartbeat task toggles the LED at the same rate of the heartbeat by calling the function `BSP_LED_On()` and `BSP_LED_Off()` in `bsp.c`.

8-7-1 BIOMEDICAL SIGNAL ANALYSIS

The algorithm to calculate the heart rate off the ECG signal relies on the time-domain features of the QRS complex such as the steep slope and high amplitude. The algorithm is defined by `AppTaskDAQ_DetectQRS()` in `app.c`

The DAQ task keeps a sliding window of three samples: `AppSignal1stSpl`, `AppSignal2ndSpl` and `AppSignal3rdSpl`. It also keeps a count of every sample processed in `AppSplsBtwPulsesCtr` and the counter gets reset once a QRS complex (pulse) is detected as illustrated in Figure 8-25.

Figure 8-25 **QRS Detection Algorithm**

Assuming that the QRS complex has a high amplitude and steep slope, whenever AppSignal1stSpl is greater than both AppSignal2ndSpl and AppSignal3rdSpl and the difference between AppSignal1stSpl and AppSignal3rdSpl is greater than a threshold defined by APP_MIN_PULSE_SLOPE a QRS complex or pulse is detected.

The number of samples between pulses is kept in AppSplsBtwPulsesCtr and the heart rate in beats-per-minute is calculated by:

```
AppResults.HeartRate.Cur = (60 * APP_SAMPLING_RATE) / AppSplsBtwPulsesCtr;
```

Listing 8-1 **Heart Rate Calculation**

The result gets inserted into a histogram to keep running statistics of the heart rate which are stored in the following variables:

```
AppResults.HeartRate.Min
AppResults.HeartRate.Max
AppResults.HeartRate.Median
```

Listing 8-2 **Heart Rate Statistics**

8-8 SUMMARY

By providing the right amount of detail with simplified terminology, this chapter explained the anatomy and physiology of the heart as far as Electrocardiography is concerned. In a similar fashion, it explained the theory of operation of an Electrocardiograph including the hardware and software.

This example application demonstrated how simple it is to implement an ECG based heart rate monitor using a combination of Micrium's μ C/OS-II and Freescale hardware. It also demonstrated two of the most important features of μ C/OS-II: *Semaphores* and *Message Queues*.

Signaling a task through its built-in semaphore is a typical method of synchronization, and in this example we used the heartbeat task's built-in semaphore to synchronize the activities of the DAQ task and the heartbeat task. The heartbeat task is signaled by the DAQ task when a new heart rate result has been calculated and depending on the heartbeat task priority, the scheduler is run. The heartbeat task may then update the cardiac cycle animation with the new heart rate result.

Message queues are built into each task and the user can send messages directly to a task from another task or an ISR. Task message queues are widely used in this example because they allow the developer to encapsulate each task's functionality into a clean and simple message-based API. For example, we used the DAQ task's built-in message queue as a means to receive different types of commands from other tasks to perform an action like start or stop the simulator, start or stop the data acquisition and process samples from the ADC's ISRs.

The need to use an RTOS in this particular example may not seem very obvious, but we hope you see that it delivers a solid platform to create a commercial medical product based on a task-oriented approach. Dividing the firmware of a medical product into different tasks in such a way that each task is responsible for some part of the application makes it easier to develop and maintain. It promotes team work and code reuse by distributing the tasks among different developers and most important, it reduces the economic impact of recertification every time you want to add new functionality or update one part of the application.

Imagine you want to take this design to the next level and be able to send the ECG samples to a computer via USB for display or analysis purposes. Or even, imagine you want to implement a web service client that submits the actual ECG waveform to a medical website for further diagnosis. Whether it is USB or Ethernet, chances are that you may need at least two more tasks to handle the reception and transmission of packets and one more task to handle all time-outs related to the communication protocol.

In the same way, LCD or any other type of display control is also a great candidate to encapsulate into a task and use the task message queue to receive commands to update the display.

Freescale and Micrium support the addition of other functionality to this heart rate monitor by offering more tower system compatible peripherals and software stacks. Visit the Freescale and Micrium websites to learn more about other products that can help you take your design to the next level.

Chapter 9

Blood Glucose Meter

The Blood Glucose Meter (BGM) is a device that measures the level of glucose in the blood. It is indispensable in the diagnosis and long-term management of diabetes.

This chapter demonstrates a basic implementation of a blood glucose meter built using µC/OS-II and Freescale products.

The chapter starts with an illustrated introduction to some of the physiological fundamentals of the system that regulates the levels of glucose in the body known as the *endocrine system*. The next section continues the introduction, but with emphasis on the measurement of the glucose concentration in the blood by using a biochemical sensor and presents the design of a Blood Glucose Meter (BGM). The last part of the chapter deals with using the tools to run the example and a description of how the code works.

9-1 GLUCOSE

Glucose is a type of sugar represented by the formula $C_6H_{12}O_6$ also known as *dextrose*. This carbohydrate is a source of energy in most living organisms including humans. It is also essential as a precursor in the creation of proteins and the production and degradation of a special group of molecules known as *lipids* as we will describe in the next paragraphs.

The levels of concentration of glucose in the bloodstream are regulated by two chemicals known as *hormones* that are released by the *pancreas*. The two hormones have opposite effects and are known as *insulin* and *glucagon*. Insulin is normally produced by a special type of cells in the pancreas called *beta cells* while glucagon is produced by *alpha cells*.

High levels of glucose in the blood trigger the pancreas to release insulin into the bloodstream, while low levels of glucose in the blood signal the pancreas to release glucagon; with the only purpose to maintain stable levels of glucose in the blood. These chemicals are the means of communication between the pancreas and the rest of the body's

Chapter 9

cells to let them know that the glucose level is either high or low. The cells of the nervous system are the only cells that do not require any messages. In other words, they require a constant supply of glucose because that is the main source of energy they consume.

The key players in this closed-loop control system include the *small intestine*, the *pancreas*, the *liver*, the body tissue cells, the *adipose* tissue cells and are shown in Figure 9-1.

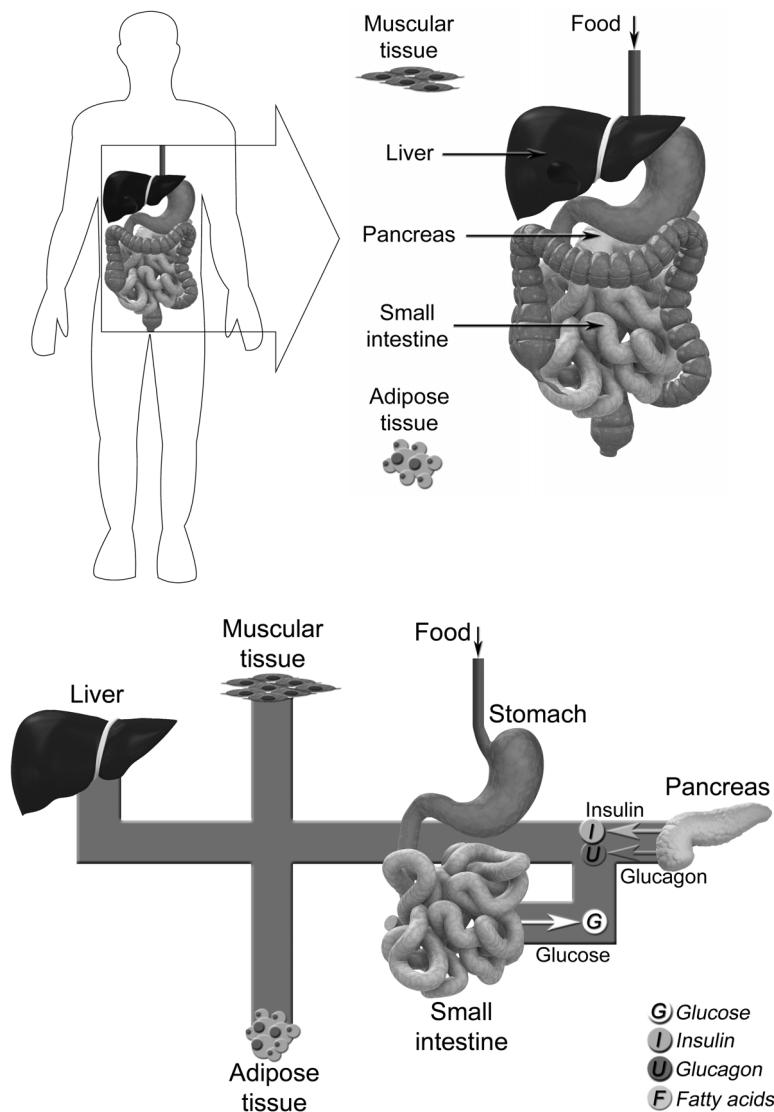
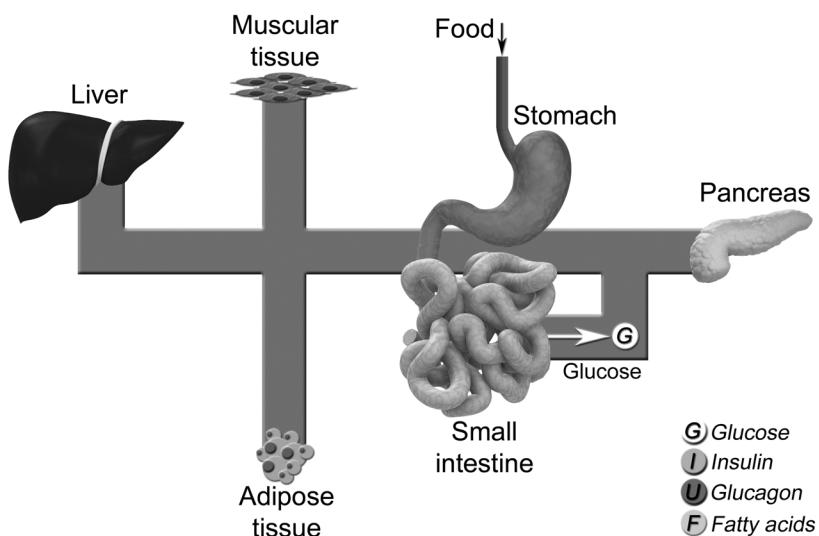


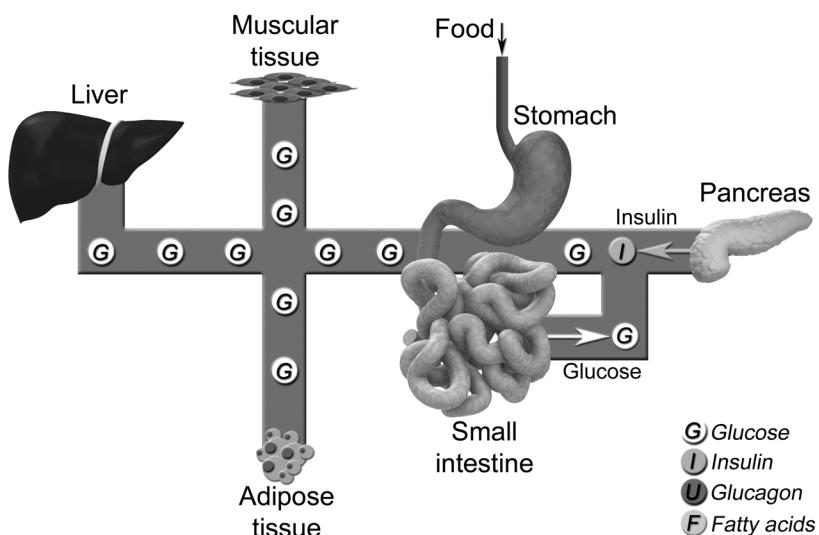
Figure 9-1 Key players in the natural blood glucose regulation

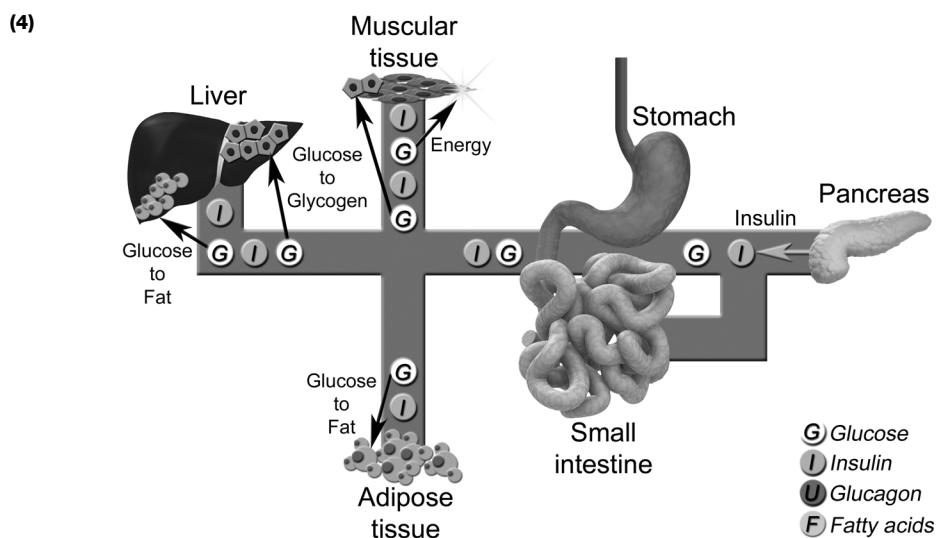
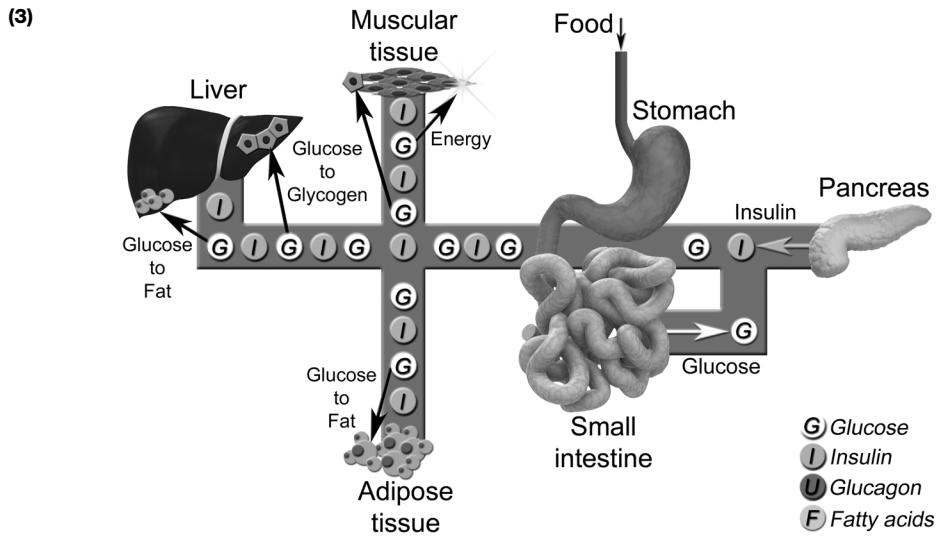
The regulation of glucose levels in the blood by this feedback system can be broken down into a series of events illustrated in Figure 9-2 that, for sake of illustration, takes apart all the key players and interconnects them via the circulatory system only (bloodstream):

(1)



(2)





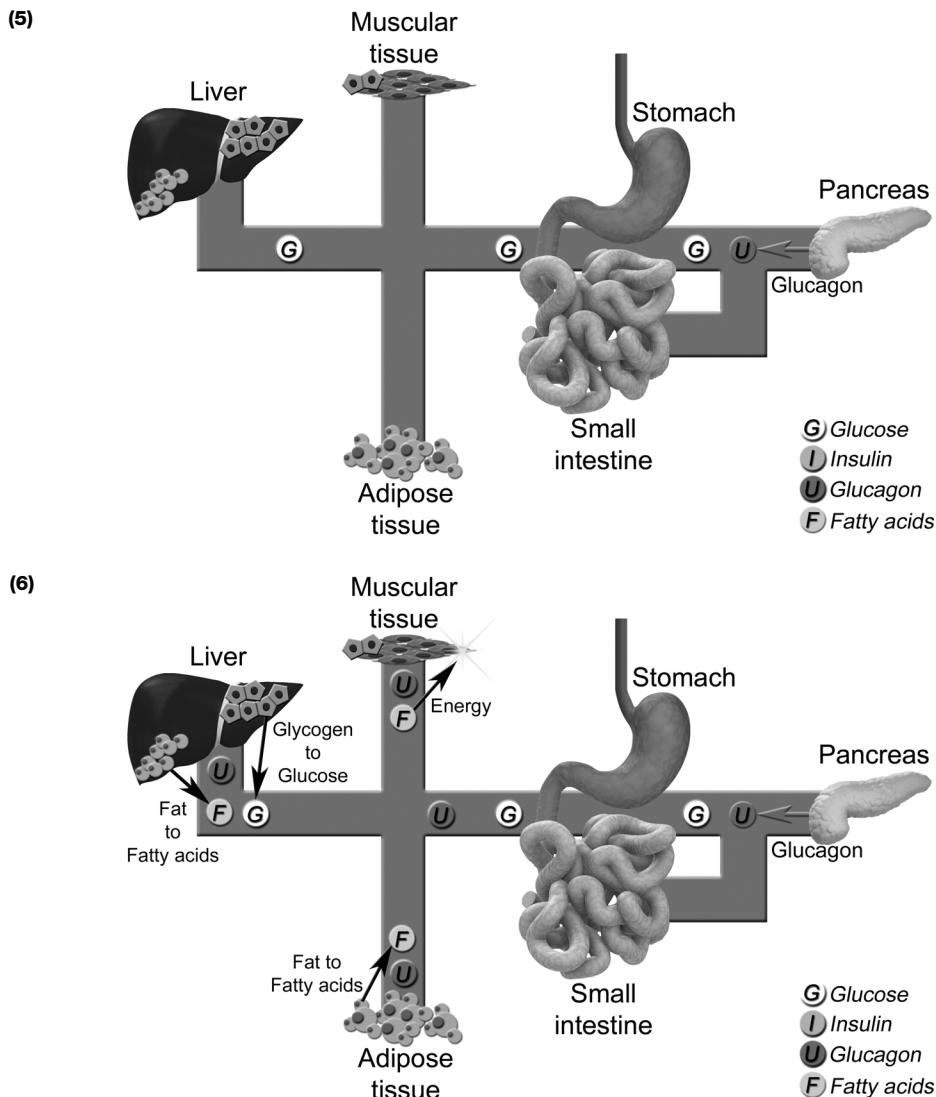


Figure 9-2 Natural blood glucose regulation

- F9-2(1) Food is absorbed by the small intestine and the carbohydrates in the food are digested into a type of sugar known as *monosaccharides* and more specifically glucose. The glucose gets released into the bloodstream.
- F9-2(2) The increased levels of glucose in the blood trigger the pancreas to release a type of chemical known as *insulin* into the bloodstream.

- F9-2(3) The insulin makes its way to the body cells (like muscle cells) through the circulatory system and stimulates or enables them to take glucose as their source of energy. The muscle cells also use some glucose to build molecules capable of storing energy in the form of a special type of molecule known as *glycogen*. This secondary energy storage is useful in the event the glucose level goes down in the future.

The insulin also reaches the liver cells to promote the conversion of glucose into fat and glycogen as molecules for secondary energy storage. The glycogen is stored in the liver.

In similar fashion, the insulin reaches the fat cells of the adipose tissue to stimulate the uptake of glucose and the conversion into more fat.

- F9-2(4) As long as the levels of glucose remain above a certain threshold equivalent to the amount of glucose circulating through the bloodstream when the small intestine is not absorbing anymore glucose, the pancreas will keep releasing insulin and the body, liver and fat cells will keep using the glucose. But of course, the glucose levels will eventually decrease if the stomach and intestines are empty.

- F9-2(5) When the glucose levels in the blood are low, the pancreas stops releasing insulin and instead starts releasing a chemical known as glucagon into the bloodstream. It is important to note that some glucose is still circulating but it is reserved for the nervous system.

- F9-2(6) The glucagon reaches the liver and enables it to convert their reserve energy molecules known as glycogen into glucose which is released into the bloodstream. While this glucose is mostly reserved for the cells of the nervous system the liver also converts fat into fatty acids which many body cell types like the heart can use for energy.

The glucagon gets to the body cells and enables them to switch their source of energy from glucose to fatty acids. Some cells from the body like the muscle are also capable of using the small percentage of glycogen stored in the muscle and turn it into glucose for the muscular contraction.

The body will keep burning fat from the adipose tissue in the absence of glucose and the last resource of energy is the protein of the muscles that can be degraded into glucose; certainly a non-desirable situation that is avoided by the stomach and pancreas by the release of a hormone known as *ghrelin* that stimulates hunger. The cycle completes when the glucose levels come back up by the digestion of new carbohydrate rich food.

9-2 DIABETES MELLITUS

Diabetes is a chronic disease that results either when the pancreas does not produce enough insulin or when the body cells cannot effectively respond to the insulin it produces. Either way, the effect is a high level of glucose in the bloodstream known as *hyperglycaemia* that over time, can damage the heart, blood vessels, eyes, kidneys and nerves among others.

According to the World Health Organization (WHO) and their Fact sheet # 312 of January 2011 these are some of the facts about diabetes:

- More than 220 million people worldwide have diabetes.
- In 2004, an estimated 3.4 million people died from consequences of high blood glucose.
- More than 80% of diabetes deaths occur in low- and middle-income countries.

WHO recognizes three types of diabetes:

- Type 1 diabetes is characterized by deficient insulin production and requires daily administration of insulin. The cause of type 1 diabetes is not known and it is not preventable with current knowledge. Symptoms include excessive excretion of urine, thirst, constant hunger, weight loss, vision changes and fatigue. These symptoms may occur suddenly.
- Type 2 diabetes results from the body's ineffective use of insulin. Type 2 diabetes comprises 90% of people with diabetes around the world, and is largely the result of excess body weight and physical inactivity. Symptoms may be similar to those of Type 1 diabetes, but are often less marked. As a result, the disease may be diagnosed several

years after onset, once complications have already arisen. Until recently, this type of diabetes was seen only in adults but it is now also occurring in children.

- Gestational diabetes is hyperglycaemia with onset or first recognition during pregnancy and results from the pregnancy hormones blocking the effects of insulin. Symptoms of gestational diabetes are similar to Type 2 diabetes. Gestational diabetes is most often diagnosed through prenatal screening, rather than reported symptoms.

Even though lifestyle modifications like a healthy diet, regular physical activity, maintaining a normal body weight and avoiding tobacco use can prevent or delay the onset of type 2 diabetes, the accurate measurement of the concentration of glucose in the blood is essential not only in the diagnosis but also in the long-term management of diabetes mellitus. Patients with diabetes can take a blood sample themselves and use the readings from a blood glucose meter along with their doctor's advice, to adjust their prescribed amount of insulin to administer and to make modifications in their lifestyle.

Figure 9-3 shows a commercial blood glucose meter, the patient usually inserts a disposable sensor commonly known as *test strip* into the blood glucose meter and uses a device to prick the fingertip's skin in a controlled manner known as *lancing device* in order to draw a small blood sample. The sample is then deposited into the test strip and a few seconds later the reading is displayed.



Figure 9-3 Commercial blood glucose meter

The next section will describe a blood glucose sensor typically used in most blood glucose meters.

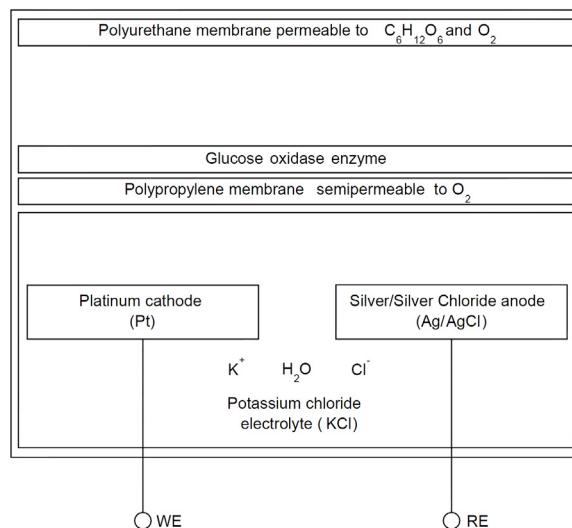
9-3 BLOOD GLUCOSE SENSOR

Many biosensors are capable of generating signals that are correlated with the concentration of glucose in the blood. Some of them employ fiber-optic, laser and infrared LEDs to take an optical approach, but most blood glucose meters today take on the *electroenzymatic* approach. This section will describe the fundamentals of the electroenzymatic approach in general, considering that many sensors today use a variation of it in an effort to optimize sensitivity, stability, response time and accuracy. Some blood glucose meters are even compatible with multiple types of sensors.

The electroenzymatic sensor described here is based on a very popular chemical reaction in nature known as *glucose oxidation* and the sensor involves a very popular electrode that measures oxygen concentration developed by Leland Clark and it is known as a *Clark-type* electrode.

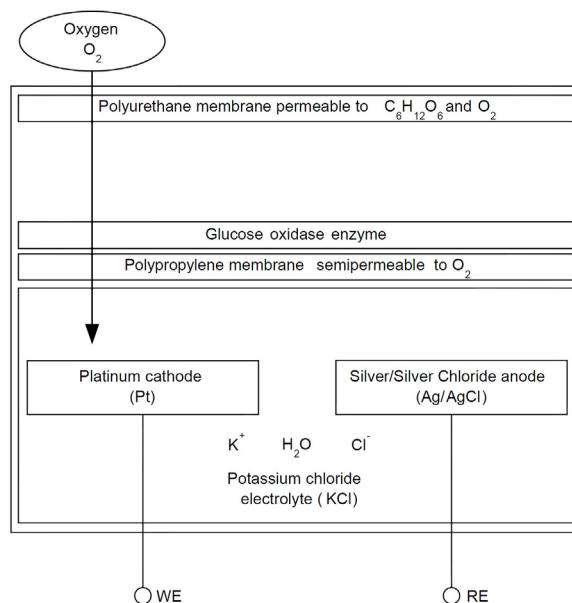
The sensor includes a series of membranes in order to isolate some molecules, a chemical that catalyzes a reaction known as *glucose oxidase enzyme* and two electrodes immersed in an *electrolyte*. Figure 9-4 shows the sequence of chemical reactions that take place in this type of sensor:

(1)

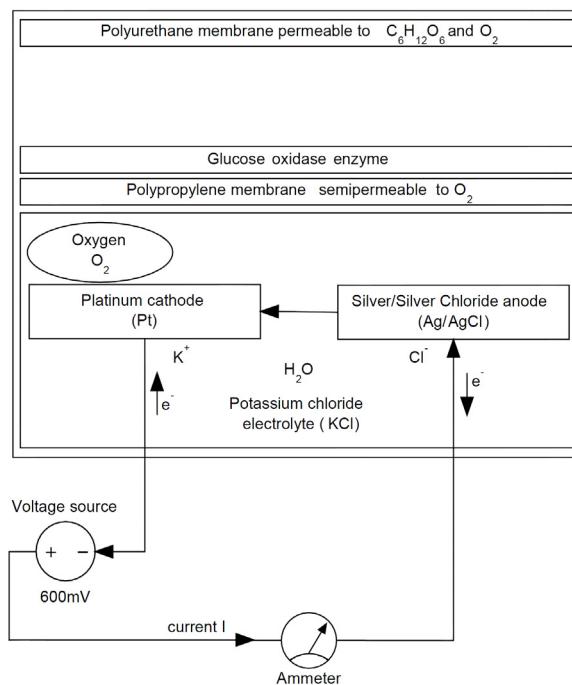


Chapter 9

(2)



(3)



(4)

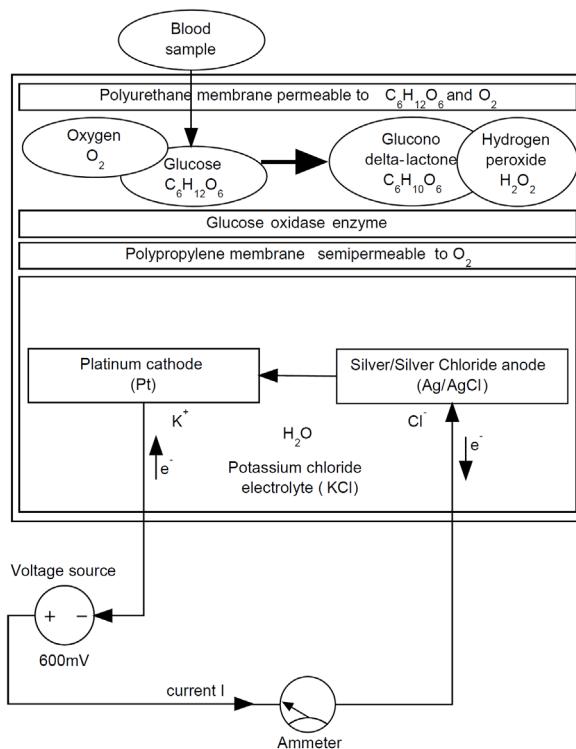
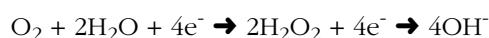


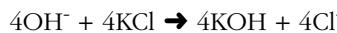
Figure 9-4 Measuring blood glucose concentration by the electroenzymatic approach

- F9-4(1) The Working Electrode (WE) made of platinum and the Reference Electrode (RE) made of silver are immersed in a solution of *potassium chloride* (KCl) and water (H_2O). When KCl dissolves in water, two different free ions will form: K^+ cations and Cl^- anions. This makes the electrolyte for the cell, which is nothing but a liquid that contains free ions that will be the carriers of electric current.
- F9-4(2) Without the blood sample being applied yet, the oxygen in the air makes its way through the polyurethane and polypropylene membranes to diffuse into the electrolyte.
- F9-4(3) A negative voltage of 600mV is applied to the platinum (Pt) electrode (cathode) referenced to the silver/silver chloride (Ag/AgCl) electrode (anode). Each electrode attracts ions that are of the opposite charge. That is, positively charged ions (K^+) move towards the negative cathode (Pt) that is providing the electrons whereas negatively charged ions (Cl^-) move towards the positive anode (Ag).

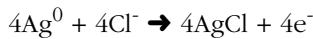
The oxygen in the vicinity of the cathode and the negative electrons supplied by the 600mV potential cause the following reaction (reduction):



The four negative ions known as *hydroxides* created by the reduction are buffered by the KCl electrolyte:

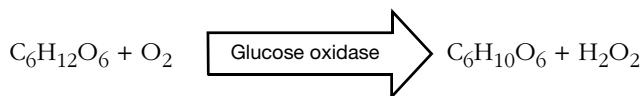


The four negative chloride ions will undergo their own oxidation reaction at the positive electrode (anode):



With this last reaction the cycle is complete, in other words, this last reaction creates the four electrons necessary in the first reaction. The cycle is complete, the circuit is closed, and the constant current is measured with an ammeter.

- F9-4(4) By using a device to prick the skin in a controlled manner known as *lancing device*, a blood sample from the fingertip is applied to the sensor. The glucose in the blood sample gets in contact with the glucose oxidase enzyme through the polyurethane membrane and reacts with the oxygen from the air to produce *glucono delta-lactone* and hydrogen peroxide:



Either the consumption of oxygen or the production of hydrogen peroxide from the chemical reaction of glucose and oxygen can be detected by the cathode depending on the type of membrane used to separate the top of the sensor and the cell at the bottom. In this case it is a polypropylene membrane that is semi-permeable to the small molecules of oxygen, which lets the oxygen diffuse into the electrolyte. The oxygen consumed in the chemical reaction between glucose and oxygen can be measured as a decrease in the current that

was running before the blood sample was applied, because less oxygen arrives at the cathode. The difference in the concentration of oxygen is directly proportional to the concentration of glucose.

Like we mentioned at the beginning of this section, many variations of this basic sensor have been developed over the years that include different chemicals, different type of membranes and number of electrodes, all with the purpose of improving the sensitivity, stability, response time and accuracy. But the vast majority of them share the same principle of glucose oxidation in the presence of an enzyme and a sensor that measures the changes in concentration of one or more of the participants during the chemical reaction.

Figure 9-5 shows a typical sensor commonly known as *test strip* featuring three electrodes. In this configuration the sensor is more stable than the one explained in Figure 9-4 because it adds an extra circuit that takes care of providing the voltage for the reaction leaving the reference electrode alone to create a second circuit to measure the voltage drop only. The extra electrode is called *Counter Electrode* (CE) and is made of platinum just like the Working Electrode (WE).

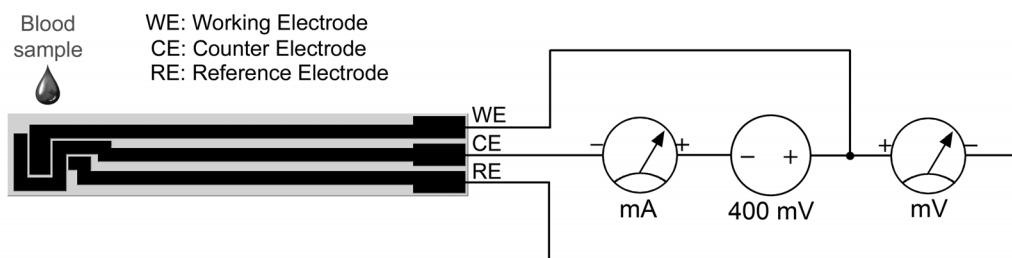


Figure 9-5 3-electrode test strip

In this test strip a voltage of 400mV is applied between the Working Electrode (WE) and the Counter Electrode (CE); and the output of the sensor is measured between the Working Electrode (WE) and the Reference Electrode (RE).

The transfer function can be calculated by applying a drop of two *stock glucose solutions* of known concentration as the transfer function is linear across the range of interest which is around 20 to 600 mg/dL (milligrams per deciliter). They are available in pharmacies and usually come in two bottles containing a low and high known concentration of glucose. The two readings from these solutions are enough to calculate the function transfer. The glucose solution comes in very handy too while testing your system without having to use your own blood.

9-4 BLOOD GLUCOSE METER DESIGN

As we learned in the last section, several sensor designs are available today but they all share the same fundamentals: a polarization voltage is applied and a voltage or current is read back. With the appropriate electronic circuits, the analog front end of this system is fairly simple.

Freescale's Kinetis TWR-K53N512 is an excellent part not only to drive this type of sensor, but also to acquire its output. The K53 microcontroller comes with a voltage reference module capable of providing an accurate voltage reference with a trim resolution of 0.5mV per step with a dedicated output pin. The K53 also comes with a 16-bit ADC that includes a temperature sensor whose output is connected to one of the ADC channel inputs. This allows the engineer to make adjustments to the transfer function in order to account for ambient temperature changes. Other nice features include a hardware averaging function, automatic compare functions and programmable gain amplifiers.

In order to read the three-electrode test strip you need a special circuit known as *transimpedance amplifier*. The transimpedance amplifier is nothing but an operational amplifier in the inverting configuration as shown in Figure 9-6 that converts current to voltage. The circuit also has a voltage divider in order to provide the 400mV to the reference electrode. A low-pass filter keeps the Working and Counter Electrode signals below 8Hz.

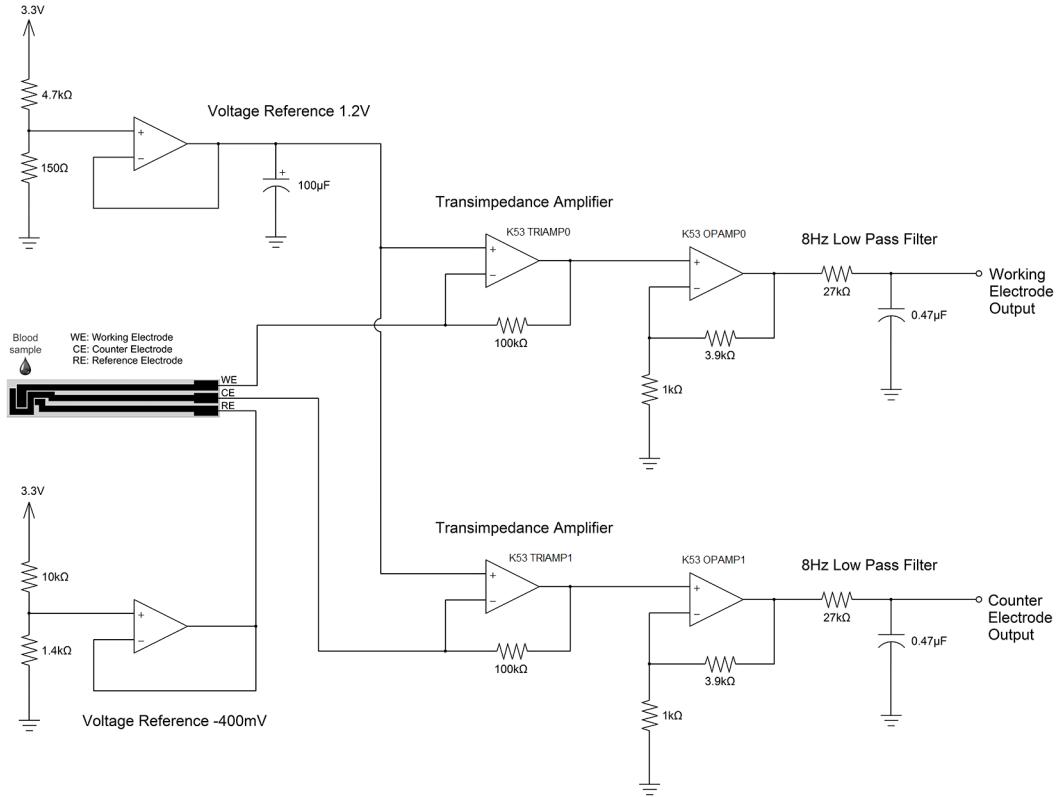


Figure 9-6 3-electrode test strip signal conditioning schematics

Freescale offers an analog module with a medical connector compatible with the TWR-K53N512 controller that implements this type of analog front end. The module is called MED-GLU.

The complete system including the TWR-K53N512 and MED-GLU is illustrated as a block diagram in Figure 9-7. The two pushbuttons onboard the TWR-K53N512 are used to either start or stop the application and the LEDs are used to indicate an error by turning on the orange LED or to indicate a successful reading by turning on the yellow LED.

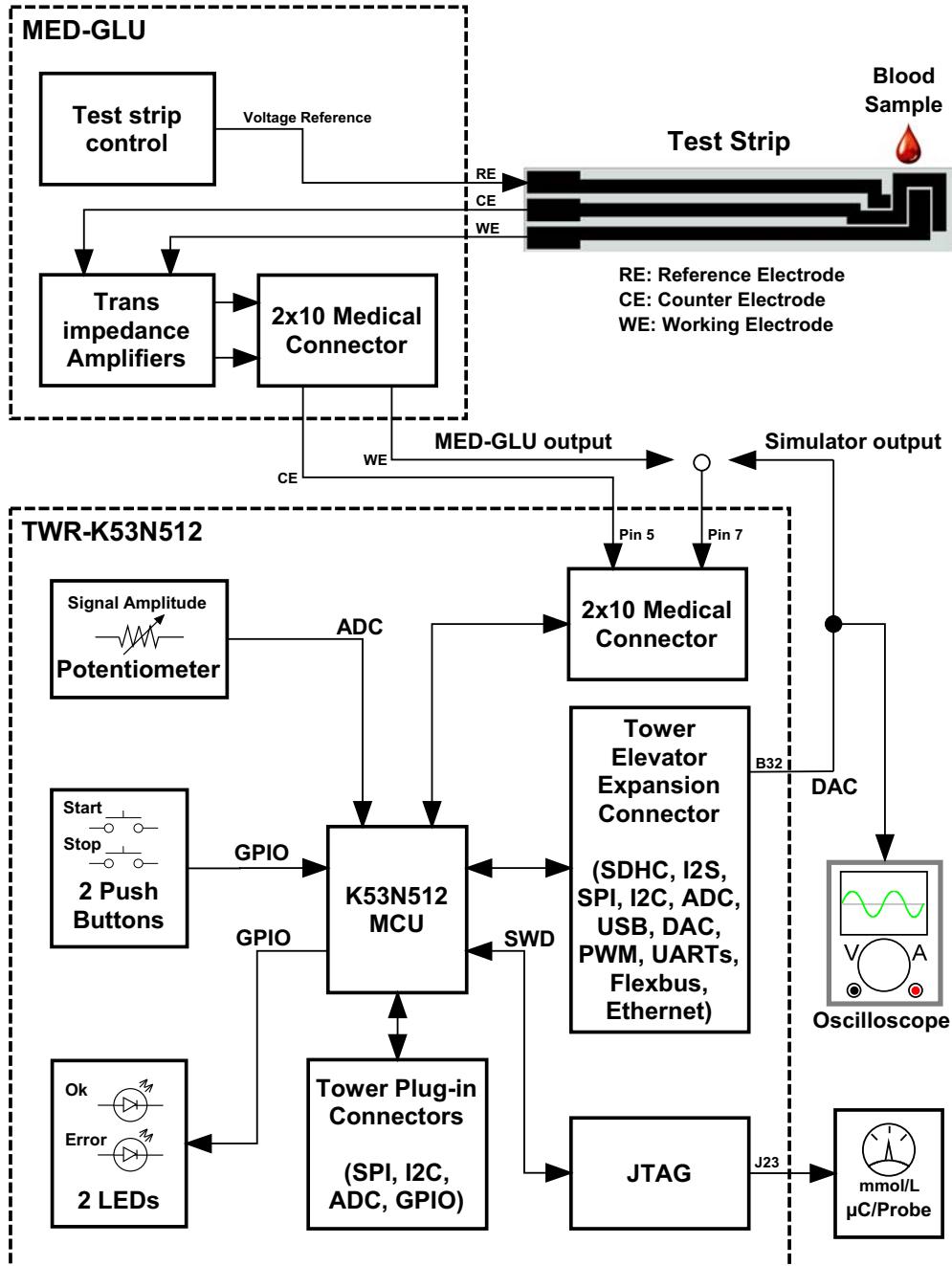


Figure 9-7 Blood Glucose Meter Block Diagram

The 2x10 medical connector of the MED-GLU not only provides connectivity with the TWR-K53N512 board to make the Working and Counter Electrode signals available to the ADC modules but also allows the use of the on-chip op-amps and transimpedance amplifiers of the Kinetis 32-bit microcontroller to implement the required signal conditioning. The rest of the signals have been omitted from the block diagram for the sake of simplicity but Table 9-1 shows the signal present in each pin of the medical connector.

MED-GLU Signal	Pin		MED-GLU Signal
VDD	1	2	GND
Not connected	3	4	Not connected
Counter Electrode Signal	5	6	Not connected
Working Electrode Signal	7	8	DAC (Voltage Reference)
OPAMP0 VOUT0	9	10	OPAMP1 VOUT1
OPAMP0 INN0-	11	12	OPAMP1 INN1-
OPAMP0 INP0+	13	14	OPAMP1 INP1+
TRIAMP0 INP0+	15	16	TRIAMP1 INP1+
TRIAMP0 INN0-	17	18	TRIAMP1 INN1-
TRIAMP0 VOUT0	19	20	TRIAMP1 VOUT1

Table 9-1 Medical Connector 2x10 Pin Header Connections

Notice from the block diagram in Figure 9-7 that in the absence of the MED-GLU module you can still run the application by using the simulated Working Electrode signal coming out of the DAC module by connecting a jumper wire between pin 7 of the medical connector and pin B32 of the elevator module in the tower system. The potentiometer is used in such situation to increase or decrease the simulated blood glucose level.

The communication with μC/Probe which will be used to display the results of the blood glucose meter is via the J-Link debug probe from Segger through the JTAG connector labeled as J23.

Figure 9-8 shows the MED-GLU module connected to the TWR-K53N512 controller in a tower system configuration.

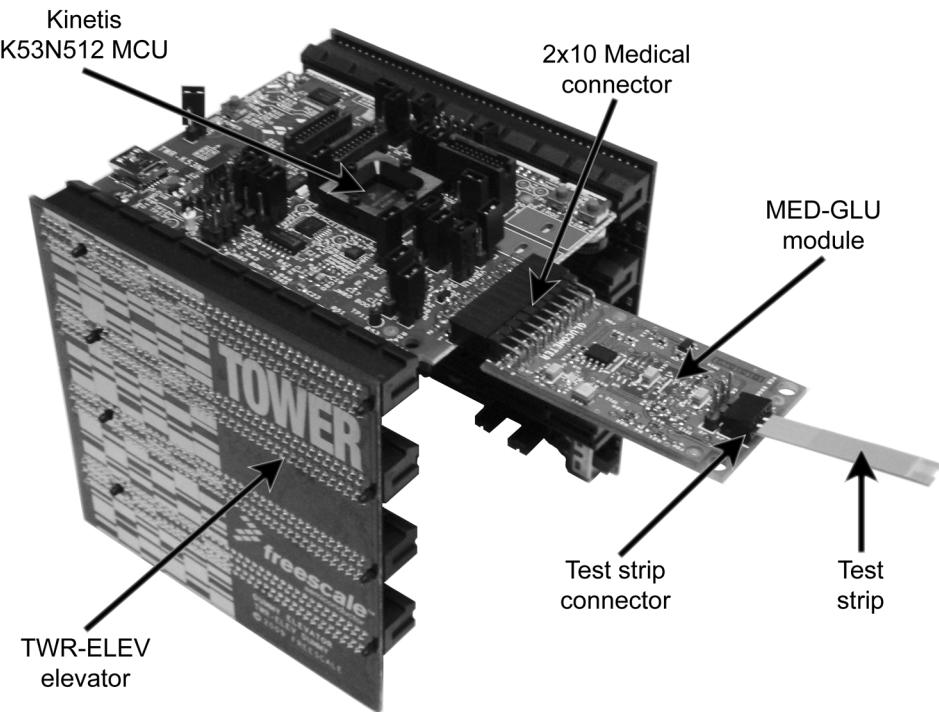


Figure 9-8 MED-GLU and TWR-K53N512

9-5 RUNNING THE EXAMPLE PROJECT

This section describes the steps you need to follow to run the blood glucose meter example. Start by connecting the MED-GLU module to the TWR-K53N512 controller through the 2x10 medical connector as shown in Figure 9-8.

The MED-GLU is compatible with the 3-contact-bars Optium blood glucose test strips from Abbott Diabetes Care. The test strips come individually wrapped in foil packets to avoid a reaction with oxygen in the air. *Do not unpack the test strip until you have completed all the following instructions.*

Assuming you have followed all the setup steps described in Chapter 7, “Setup” on page 101, proceed to connect one end of the Segger’s J-Link for ARM debug probe to the JTAG connector on J23 of the TWR-K53N512 and the other end of the probe to any USB port available in your PC.

Run IAR's Embedded Workbench for ARM (EWARM) and open the workspace at:

\$\Micrium\Examples\Freescale\TWR-K53N512\OS2-TCPIP-BGM\IAR\
OS2-TCPIP-BGM.eww

The workspace window shows all of the files in the project which are sorted into groups represented by folder icons. Figure 9-9 shows the project files for OS2-TCPIP-BGM in the workspace explorer window.

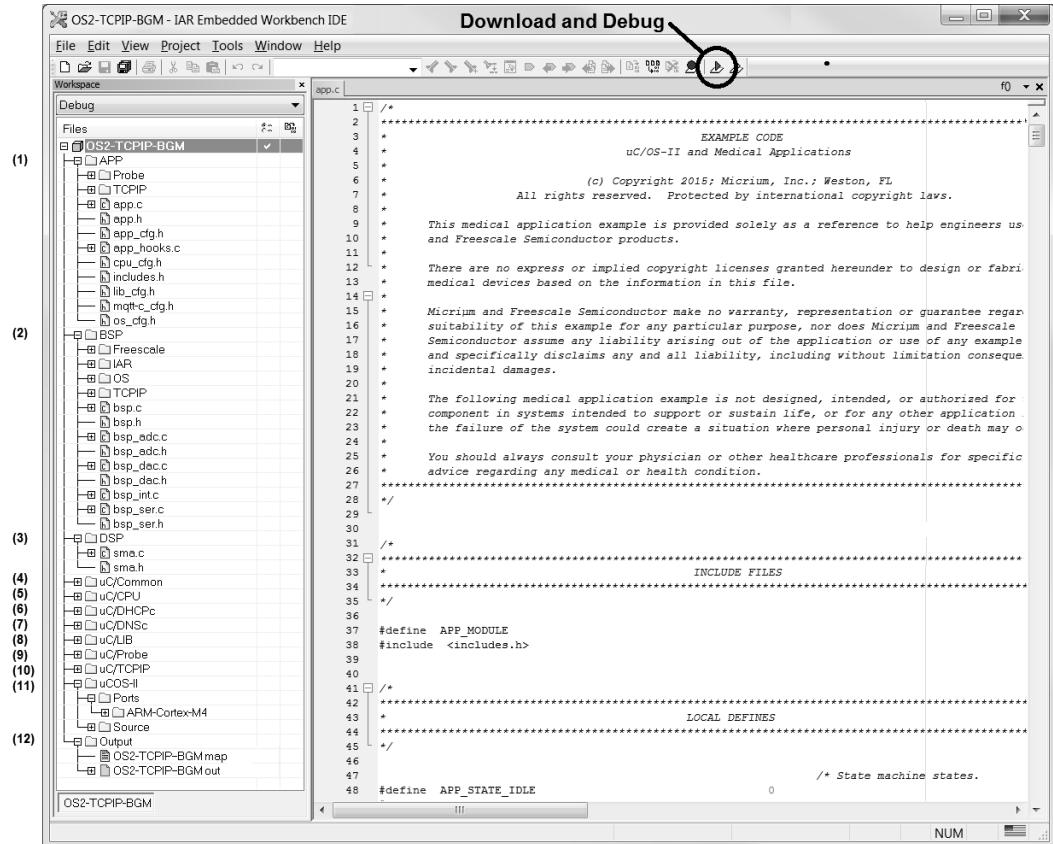


Figure 9-9 Running the project with EWARM

- F9-9(1) The **APP** group includes all of the application files for this example, including the header files used to configure the application.

- F9-9(2) The **BSP** group contains the files that comprise the board support package. The board support package includes the code used to control the peripherals on the board. For this example, the software to control the LEDs, pushbuttons, ADCs, DACs, operational amplifiers and transimpedance amplifiers will be used. The subgroup Freescale includes the header files for the Kinetis K50 family of microcontrollers.
- F9-9(3) The **DSP** group contains the files that define some of the Digital Signal Processing functions called by the application.
- F9-9(4) The **uC/Common** group contains the kernel abstraction layer.
- F9-9(5) The **uC/CPU** group contains the µC/CPU source code files.
- F9-9(6) The **uC/DHCPC** group contains the header files for the DHCP client that allows you to negotiate an IP address with your network's router.
- F9-9(7) The **uC/DNSc** group contains the header files for the DNS client that allows you to translate domain names into IP addresses.
- F9-9(8) The **uC/LIB** group contains the µC/LIB source code files.
- F9-9(9) The **uC/Probe** group contains the source code to enable communication with µC/Probe via TCP/IP.
- F9-9(10) The **uC/TCP-IP** group contains the basic header files for the TCP/IP stack.
- F9-9(11) The **uC/OS-II** group contains the µCOS-II source code.
- F9-9(12) The **Output** group contains the files generated by the compiler/linker.

Start the debugger by clicking the “Download and Debug” icon as shown in Figure 9-9. The application is programmed to the internal Flash of the K53N512 microcontroller and the debugger starts. The code automatically starts executing and stops at the `main()` function in the `app.c` file.

Click the “Go” icon in IAR Embedded Workbench’s debugging tools to run the application as shown in Figure 9-10.

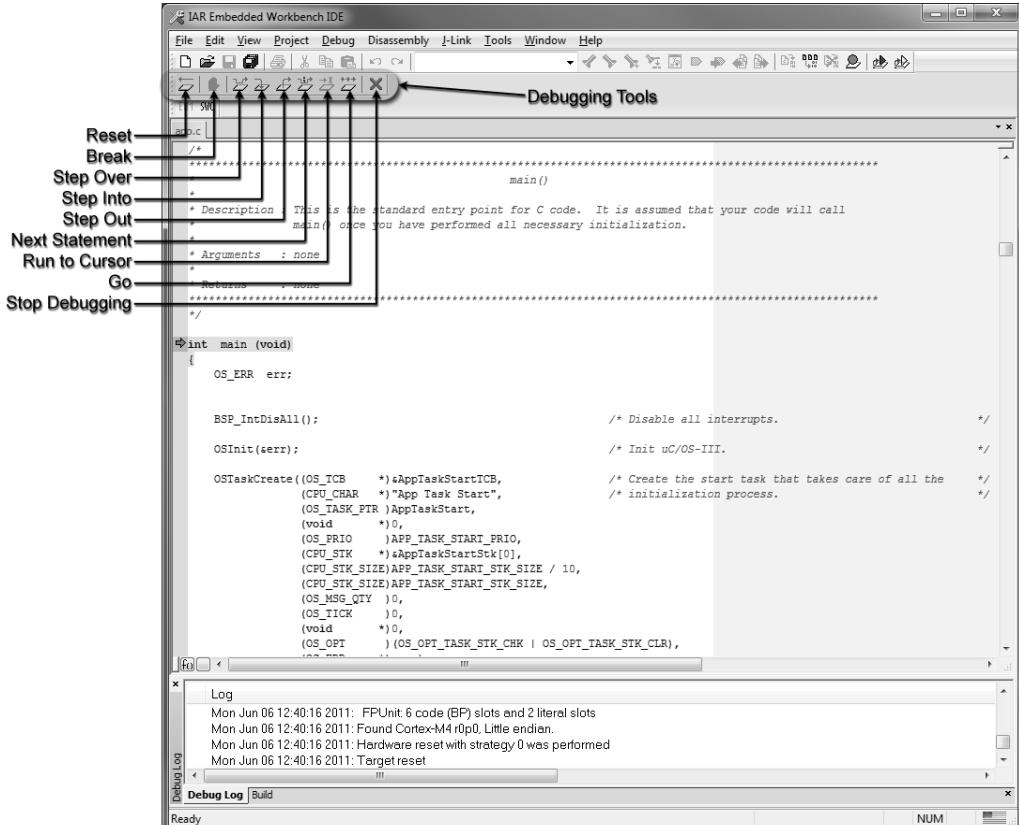


Figure 9-10 Downloading the Application and Starting the Debugger

You can test the application by starting μC/Probe and open the workspace for the Blood Glucose Meter at:

`$\Micrium\Examples\Freescale\TWR-K53N512\OS2-TCPIP-BGM\IAR\OS2-TCPIP-BGM.wspx`

After μC/Probe starts, click the run button (green triangle). Once μC/Probe is running, you will see a screen similar to the one shown in Figure 9-11

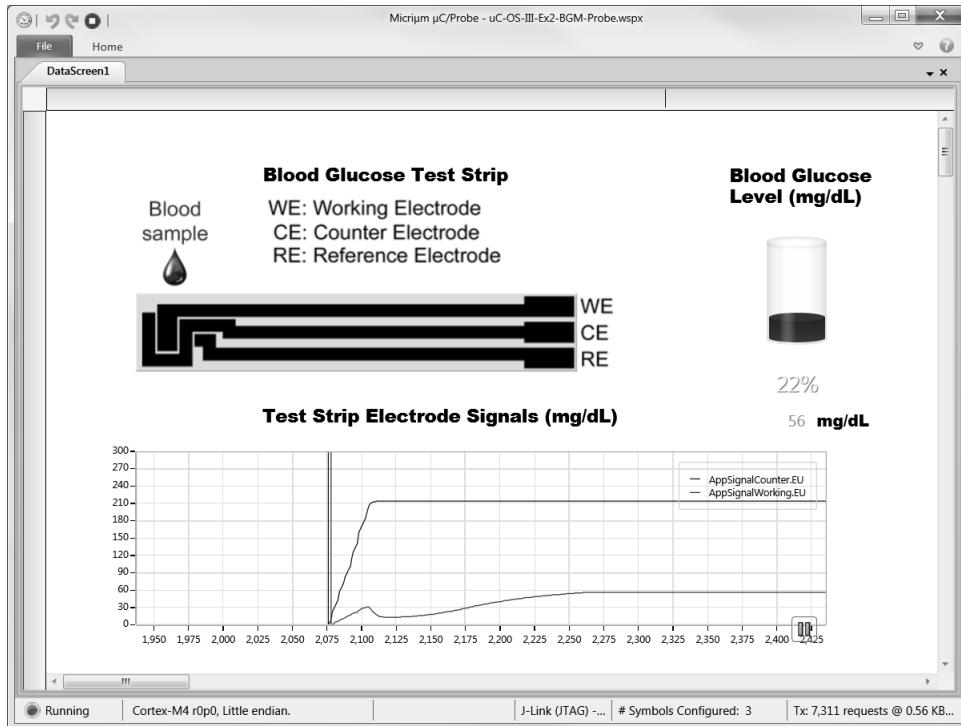


Figure 9-11 µC/Probe Blood Glucose Meter Dashboard

Press the pushbutton on the TWR-K53N512 board labeled as SW1 to start the application.

The application will wait for the detection of a valid test strip and a valid blood sample.

Unpack one of the test strips and insert it into the 3-contact connector in J4 of the MED-GLU as shown in Figure 9-8.

If you don't feel like lancing your finger to obtain a blood sample, you may use a control solution with a known glucose level. If you decide to test it with your own blood then first make sure your finger is clean, dry and warm. In order to help the blood flow, hang your arm down before lancing your finger and obtain the blood drop by following the instructions of the lancing device you were able to purchase. Get the blood drop to touch the white target area at the end of the test strip and the blood drop will be drawn into the test strip by capillary motion.

You will start seeing the Working and Counter Electrodes signal tracings in real time on the dashboard. The blood glucose level will be displayed in less than five seconds as shown in Figure 9-11.

According to the World Health Organization (WHO) the expected blood glucose range for a non-diabetic, non-pregnant fasting adult is 74-106 mg/dL and between one and two hours after a meal, the levels should be less than 160 mg/dL. *You should always consult your healthcare professional to determine the range that is appropriate for you.*

9-6 HOW THE CODE WORKS

The blood glucose meter application consists of three different tasks:

- User IF task: The user interface task monitors the state of the two pushbuttons onboard the TWR-K53N512. The pushbuttons are used to either start or stop the application. The task is defined by `AppTaskUserIF()` in `app.c`.
- Sim task: The simulator task monitors the state of the potentiometer and updates the amplitude of a Working Electrode simulated signal generated by the DAC according to the position of the potentiometer. The task is defined by `AppTaskSim()` in `app.c`.
- DAQ task: The data acquisition task is the main task in the application and implements the state machine that processes the analog input samples to calculate the blood glucose level. The task is defined by `AppTaskDAQ()` in `app.c`.

The interaction among the different tasks is facilitated by the use of µC/OS-II's semaphores and message queues and it is illustrated in Figure 9-12.

Chapter 9

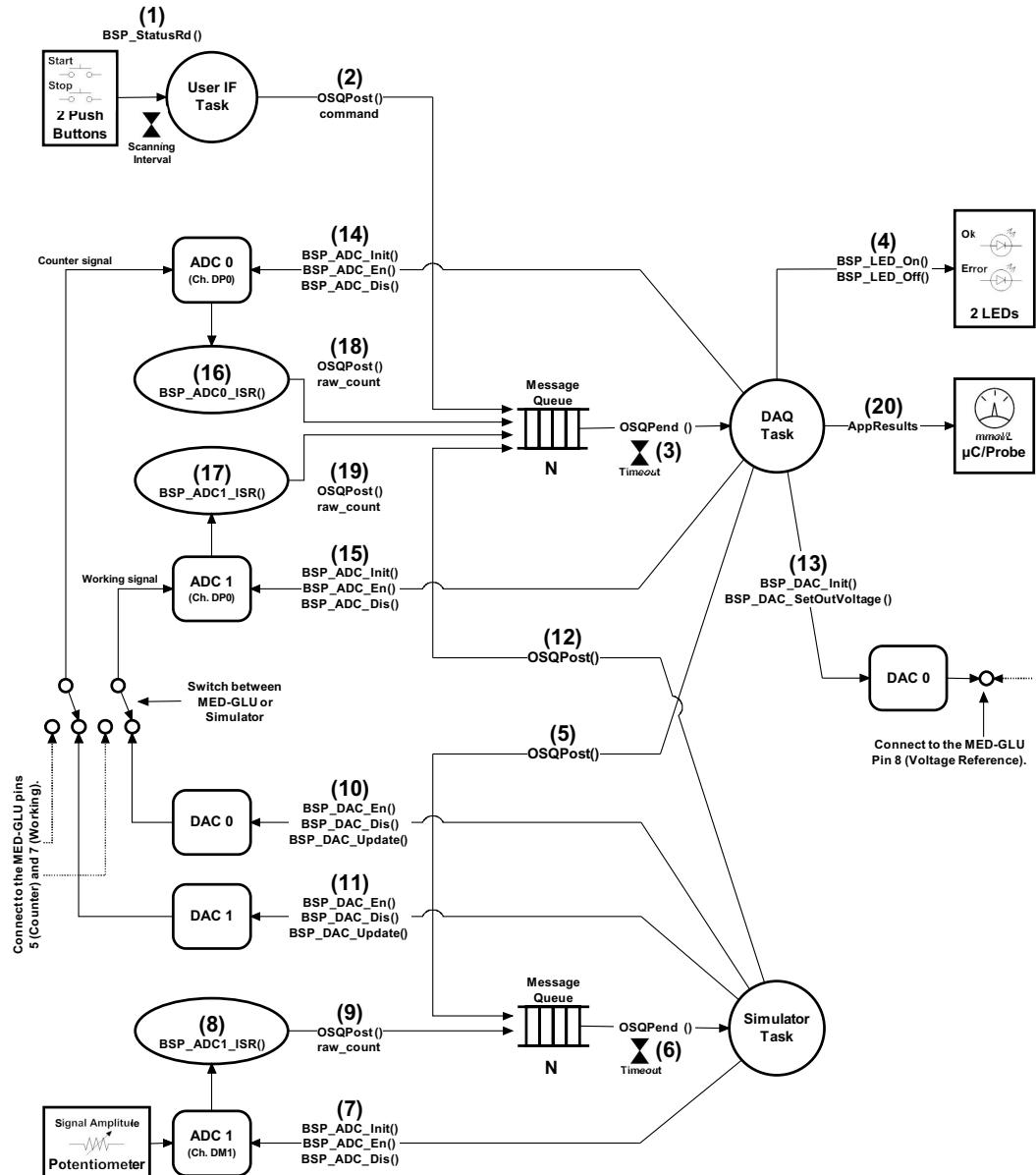


Figure 9-12 Interaction of Application Tasks

-
- F9-12(1) The state of the push buttons is monitored by the user IF task at a scanning interval defined by `BSP_STATUS_CHECK_INTERVAL`. See `AppTaskUserIF()` in `app.c` and `BSP_StatusRd()` in `bsp.c`.
- F9-12(2) Depending on the pushbutton pressed by the user a message from the user IF task to either start or stop the application is posted to the DAQ task's built-in message queue by calling `OSTaskQPost()`. See `AppTaskUserIF()` in `app.c`.
- F9-12(3) The DAQ task can receive data messages directly from the ADCs ISRs or command messages from the other two tasks. When `OSTaskQPend()` is called, the message is retrieved and processed in two different ways depending on the sender and the type of message. If the message comes from one of the ADC's ISR then the message is processed by calling the function `AppTaskDAQ_ProcessData()` and if the message comes from any other task then the message is processed by calling the function `AppTaskDAQ_ProcessCmd()` in `app.c`.
- F9-12(4) If the message retrieved by the DAQ task is a command to start the blood glucose measurement then the DAQ task turns off the LEDs by calling the function `BSP_LED_Off()` in `bsp.c`.
- F9-12(5) If the message retrieved by the DAQ task is a command to start the blood glucose measurement in simulation mode then the DAQ task posts a message to the sim task's built-in message queue to start the simulator. See `AppTaskDAQ()` in `app.c` and `AppTaskDAQ_ProcessCmd()` in `app.c`.
- F9-12(6) In similar fashion, the sim task can receive messages directly from the ADC1's ISR or from the DAQ task. When `OSTaskQPend()` is called, the message is retrieved and processed in different ways depending on the sender. The sim task retrieves the received message from its message queue and proceeds according to the command issued by the DAQ task as follows:
- F9-12(7) If the message contains a start command then the sim task starts ADC1 channel DM1 in order to read the potentiometer. See `AppTaskSim()` in `app.c`.
- F9-12(8) The CPU is interrupted by the completion of a conversion of ADC1 Channel DM1 and the interrupt is handled by `BSP_ADC1_ISR` in `bsp_adc.c`.

-
- F9-12(9) The ADC1 ISR posts the result of the conversion in the form of a 16-bit raw count to the sim task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC1_ISR()` in `bsp_adc.c`.
 - F9-12(10) The sim task retrieves the message from the queue and configures the DAC0 to output a simulated Working Electrode waveform at an amplitude proportional to the raw count of the ADC1 channel DM1. See `AppTaskSim()` in `app.c`.
 - F9-12(11) The sim task retrieves the message from the queue and configures the DAC1 to output a simulated Counter Electrode waveform. See `AppTaskSim()` in `app.c`.
 - F9-12(12) The sim task posts a message to the DAQ task's built-in message queue notifying the DAQ task that the simulator is running. See `AppTaskSim()` in `app.c`.
 - F9-12(13) The DAQ task retrieves the message from the queue and initializes DAC0 to output a voltage reference to drive the test strip and trigger the chemical reaction. See `AppTaskDAQ()` and `AppTaskDAQ_ProcessCmd()` in `app.c`.
 - F9-12(14) The DAQ task retrieves the message from the queue and enables ADC0 channel DP0 to convert the Counter Electrode signal. See `AppTaskDAQ()`, `AppTaskDAQ_ProcessCmd()` and `AppTaskDAQ_ExecCmd()` in `app.c`.
 - F9-12(15) The DAQ task retrieves the message from the queue and enables ADC1 channel DP0 to convert the Working Electrode signal. See `AppTaskDAQ()`, `AppTaskDAQ_ProcessCmd()` and `AppTaskDAQ_ExecCmd()` in `app.c`.
 - F9-12(16) The CPU is interrupted by the completion of a conversion of ADC0 channel DP0 and the interrupt is handled by `BSP_ADC0_ISR` in `bsp_adc.c`.
 - F9-12(17) About the same time, the CPU is interrupted by the completion of a conversion of ADC1 channel DP0 and the interrupt is handled by `BSP_ADC1_ISR()` in `bsp_adc.c`.
 - F9-12(18) The ADC0 ISR posts the result of the conversion in the form of a 16-bit raw count to the DAQ task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC0_ISR()` in `bsp_adc.c`.

- F9-12(19) The ADC1 ISR posts the result of the conversion in the form of a 16-bit raw count to the DAQ task's built-in message queue by calling the function OSTaskQPost(). See `BSP_ADC1_ISR()` in `bsp_adc.c`.
- F9-12(20) The DAQ task retrieves the message from the queue and uses the raw counts from ADC0 and ADC1 to calculate the blood glucose level and display the value in μ C/Probe by calling the function `AppTaskDAO_ProcessData()` and `AppCalcResults()` in `app.c`.

The state machine that processes every sample in `AppTaskDAO_ProcessData()` is illustrated in Figure 9-13.

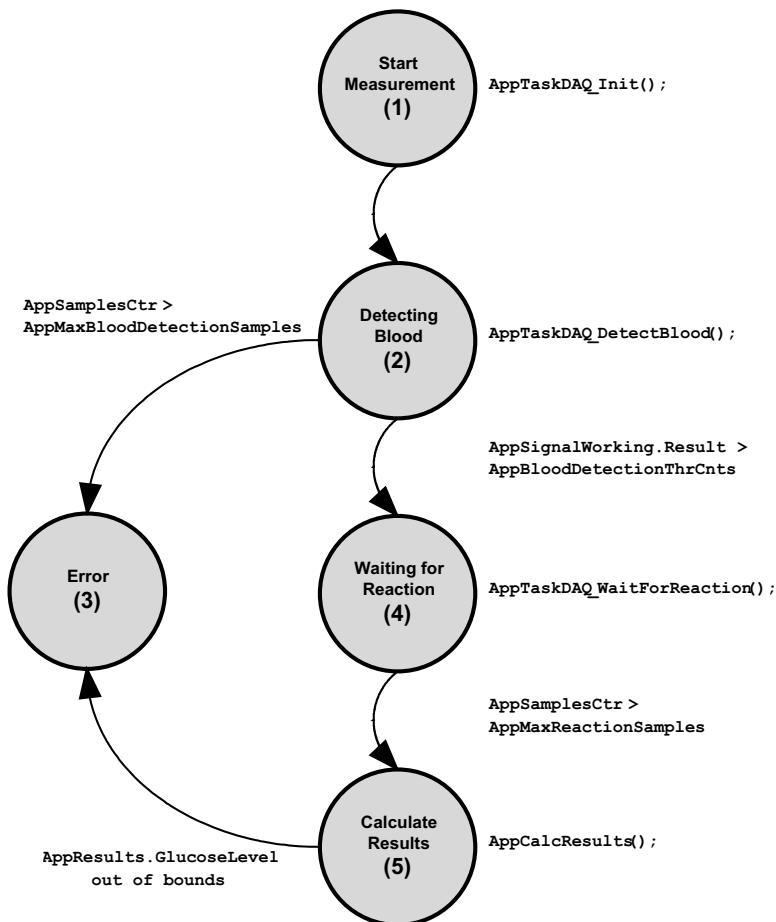


Figure 9-13 Data Processing State Machine

-
- F9-13(1) Once the start button is pressed, the application starts the blood glucose level measurement by calling `AppTaskDAQ_Init()` to initialize some global variables.
 - F9-13(2) The next state keeps monitoring the amplitude of the signal coming out of the Working Electrode until the amplitude is above a threshold that indicates the presence of a valid test strip and a valid blood sample. This threshold is defined in millivolts by `APP_BLOOD_DETECT_THR_MVOLTS`. The function that smooths the signal and detects the blood sample is called `AppTaskDAQ_DetectBlood()`.
 - F9-13(3) An error occurs in case the test strip is not valid or the blood sample is not enough. Such conditions can be detected by monitoring the amplitude of both, the Working and Counter Electrodes. The same error condition occurs if the calculated blood glucose level is out of the valid ranges defined in mg/dL by `APP_MIN_GLUCOSE_LEVEL_MGDL` and `APP_MAX_GLUCOSE_LEVEL_MGDL`.
 - F9-13(4) The chemical reaction takes place in about five seconds and in this state, every sample from the Working Electrode is inserted into a simple moving average filter with a window size of about one second defined in number of samples by `APP_SMA_WIN_SIZE`.
 - F9-13(5) The algorithm to calculate the actual blood glucose level is quite simple. A Simple Moving Average (SMA) of the voltage at the Working Electrode with a window sized at `APP_SMA_WIN_SIZE` samples is calculated over the five seconds following the beginning of the chemical reaction. The result of the SMA filter is stored in `AppSignalWorking.Result`. The blood glucose level in mg/dL is calculated in terms of the voltage at the Working Electrode by the following transfer function:

$$\text{Blood Glucose Level} = 90.2643 * \text{Working Electrode Voltage} - 75.2758$$

9-7 SUMMARY

This chapter explained the anatomy and physiology of the organs involved in the regulation of glucose in the blood and the theory of operation of a blood glucose meter including the hardware and software.

This example application demonstrated how simple it is to implement a blood glucose meter using a combination of Micrium's µC/OS-II and Freescale hardware. It also demonstrated one of the most important features of µC/OS-II: *Message Queues*.

Message queues are built into each task and the user can send messages directly to a task from another task or an ISR. You will notice that task message queues are used not only in this example but all the rest of examples because they allow the developer to encapsulate each task's functionality into a clean and simple message-based API. For example, we used the DAQ task's built-in message queue as a means to receive different types of commands from other tasks to perform an action like start or stop the simulator, start or stop the data acquisition and process samples from the ADC's ISRs.

Thanks to the task-oriented approach of this example, adding more features to the blood glucose meter is easy to accomplish. The first obvious feature to add is USB or bluetooth connectivity to make the blood glucose meter act as an accessory of computer-like devices such as smartphones or tablet PCs. This configuration takes advantage of the GUI and internet capabilities of the smartphone or tablet PC.

For a more stand-alone approach, LCD or any other type of display control is also a great candidate to encapsulate into a task and use the task message queue to receive commands to update the display.

If you want to take the design to the ultimate level you can imagine closing the control loop by not only reading the blood glucose level but also controlling an insulin infusion pump.

Freescale and Micrium support the addition of other functionality to this blood glucose meter by offering more tower system compatible peripherals and software stacks. Visit the Freescale and Micrium websites to learn more about other products that can help you take your design to the next level.

Chapter

10

Pulse Oximeter

A pulse oximeter is a device that measures the amount of oxygen transported in the arterial blood. This relative measure is known as *Oxygen Saturation* (SpO_2) and is given as a percentage. Figure 10-1 shows an example of a commercial pulse oximeter displaying the $\text{SpO}_2\%$ at the top and the heart rate in beats-per-minute at the bottom.

This chapter demonstrates a basic implementation of a pulse oximeter built using $\mu\text{C}/\text{OS-II}$ and Freescale products.

The chapter starts with an illustrated description of the mechanism of transport of oxygen from the air to the tissue cells known as respiration. The next section continues the introduction, but with emphasis on the acquisition and calculation of the SpO_2 and presents the design of a pulse oximeter (POX). The last part of the chapter deals with using the tools to run the example and a description of how the code works.



Figure 10-1 Commercial pulse oximeter

10-1 RESPIRATION

Every cell in the body requires oxygen to perform their function. The main role of the respiratory system is to replenish the blood with oxygen from the air in order for the blood to carry the oxygen to all the cells in the body. This is done through the exchange of two gases: oxygen and carbon dioxide. We inhale oxygen and exhale carbon dioxide by breathing as we will explain in the next paragraphs.

Figure 10-2 shows that breathing is accommodated by the muscular contraction of the *diaphragm* and other muscles in the chest and abdomen. The diaphragm is a sheet of muscles that extends across the bottom of the chest cavity. When the diaphragm contracts, it actually pulls down, lowering the internal air pressure and leaving more space for the lungs to fill with air. Since the air pressure in the exterior is higher, the air gets sucked in through the *nasal passage* (nose) and *oral cavity* (mouth) where it gets conditioned for temperature and moisture and filtered from any particles like dust.

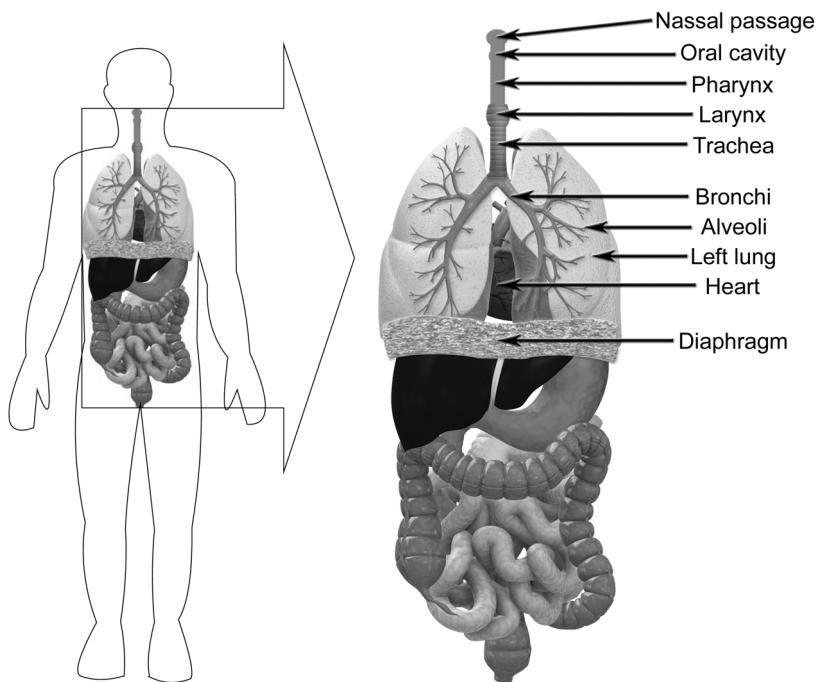


Figure 10-2 The respiratory system

Figure 10-3 below shows that the air then passes through a part of the throat known as *pharynx* and then to an organ in the neck known as the *larynx* which among other functions it is involved in protecting the rest of the respiratory system against food aspiration. The air continues to go down a tube that enters the chest cavity known as the *trachea*. There, the trachea divides into two smaller tubes that lead to the left and right *lungs* known as the *bronchi*. Inside the lungs the bronchi branch into millions of even smaller tubes which connect to tiny air filled sacs called *alveoli*. Each alveolus is wrapped around a mesh of *capillaries* which are the smallest blood vessels and are the point where deoxygenated and oxygenated blood meet.

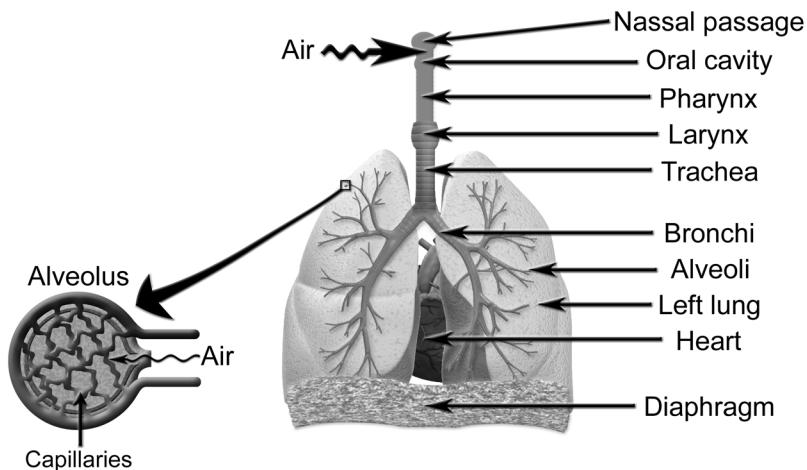


Figure 10-3 The respiratory system, inhalation.

Figure 10-4 shows that the air fills the alveoli and since the partial pressure of oxygen in the air is higher than the one in the blood, the oxygen diffuses into the blood and binds to a protein molecule that is carried by the *red blood cells* known as *hemoglobin* (Hb). When the hemoglobin binds to a molecule of oxygen, it forms what is known as *oxyhemoglobin* (Hb-O_2) which has a bright red color. At the same time, the partial pressure of carbon dioxide in the air in the alveoli is less than the partial pressure of carbon dioxide in the blood, so carbon dioxide diffuses out from the red blood cells and into the air in the alveoli.

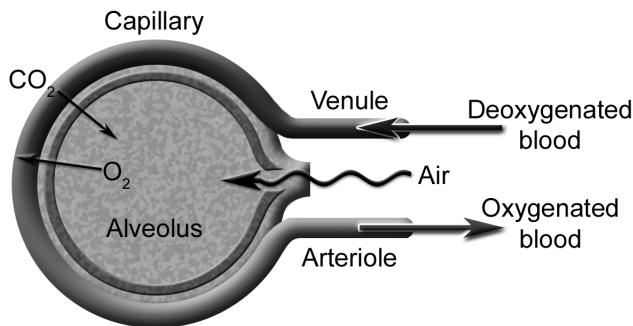


Figure 10-4 Exchange of gases at the alveolus.

Figure 10-5 shows how the oxygenated blood flows through the pulmonary veins into the heart where it gets handled by the left atrium and left ventricle to be pumped through the aorta to all the rest of the body. We do not mention the right atrium and ventricle on purpose because this explanation only focuses on the trip taken by the oxygen from the air to the tissue cells. But as we saw in section 8-1 “The Heart” on page 111, both pumping actions happen at the same time.

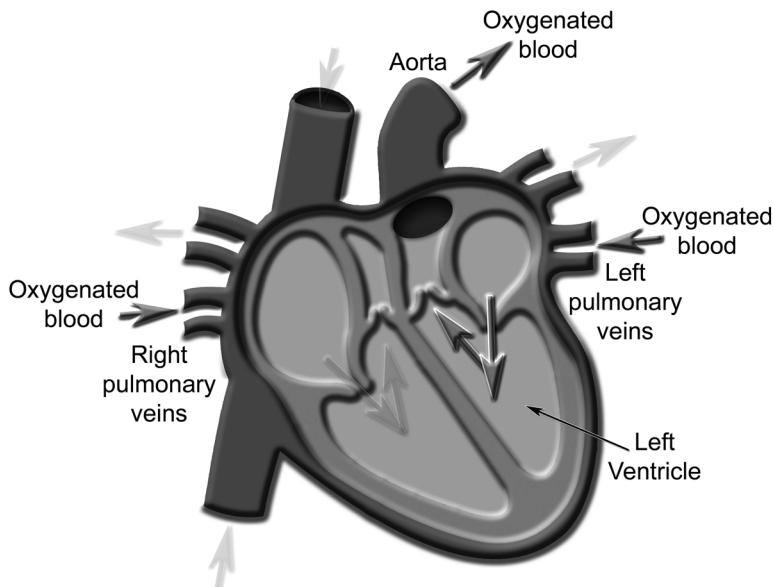


Figure 10-5 Oxygenated blood transportation by the heart.

Figure 10-6 shows how the blood makes its way to the body cells through the capillaries where it exchanges oxygen and carbon dioxide by diffusion; that is, the partial pressure of oxygen in the blood is higher than the one in the body tissues so the oxyhemoglobin releases its oxygen and the oxygen diffuses into the body tissues. When the oxyhemoglobin loses its bind to oxygen, it forms what is called *deoxyhemoglobin* (Hb) which has a color between blue and purple.

At the same time, the partial pressure of carbon dioxide in the blood is lower than the partial pressure of carbon dioxide in the body tissues, so carbon dioxide diffuses into the blood and binds to the hemoglobin to form *carbaminohemoglobin*.

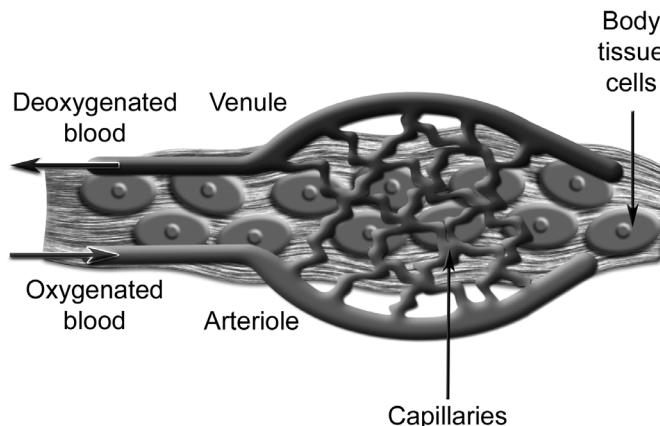


Figure 10-6 Exchange of gases at the tissues.

As shown in Figure 10-7, the deoxygenated blood is carried through the veins back to the heart where it gets handled by the right side chambers and pumped through the left and right pulmonary arteries back into the lungs.

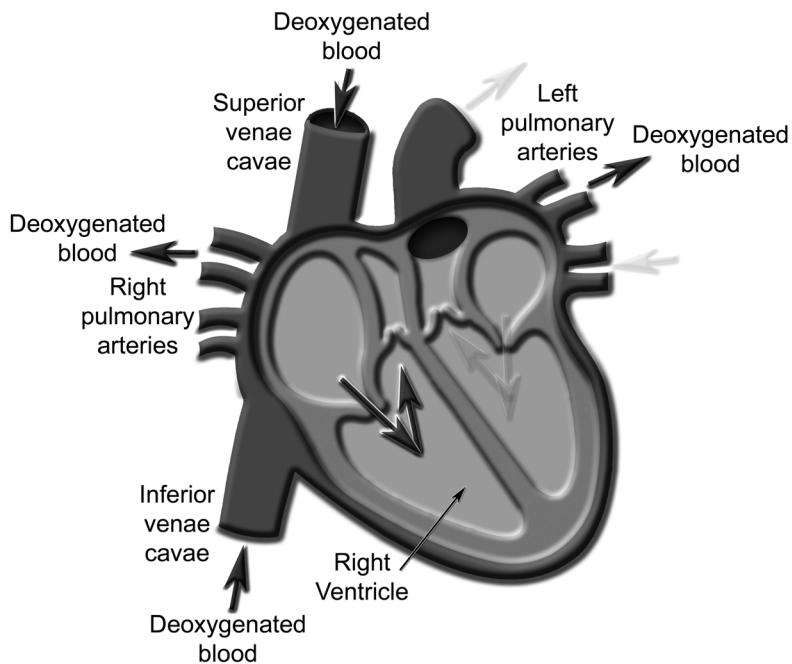


Figure 10-7 Deoxygenated blood transportation by the heart.

Figure 10-8 shows that the cycle is completed by the relaxation of the diaphragm which makes the chest cavity smaller again. The muscles squeeze the lungs and the carbonated air is pushed out of the body through the same passages it came in. The breathing cycle is repeated about 20 times per minute and the frequency is controlled by a complex system that makes sure the levels of carbon dioxide in the blood are maintained below a certain threshold. Even though the breathing frequency can also be changed voluntarily, the system is considered mostly autonomous.

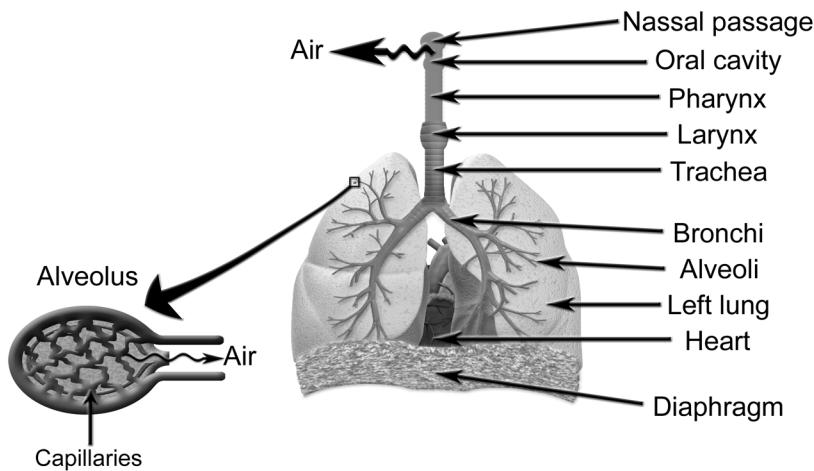


Figure 10-8 The respiratory system, exhalation.

10-2 PULSE OXIMETRY

As we learned in the last section, the blood, and more specifically the hemoglobin, can be saturated with oxygen in the form of oxyhemoglobin (Hb-O_2) or desaturated of oxygen in the form of deoxyhemoglobin (Hb). The determination of the oxygen saturation level provides very important information about the efficiency of the respiratory system function in the exchange of gases and the performance of the circulatory system to carry the gases to and from the tissues. Healthy individuals show arterial oxygen saturation levels between 97% and 99% and the values can become unstable as side effects of anesthesia and / or trauma in clinical settings like an intensive care unit, operating room and emergency room. In such situations the readings off a pulse oximeter help the doctors respond appropriately like for example controlling the concentration of oxygen to be administered to the patient. For that reason, pulse oximetry is considered one of the most important monitoring tools during any procedure that involves the use of general or regional anesthesia.

A pulse oximeter is a device with two LEDs and a photodiode that analyzes the absorption of the light transmitted through the finger or the earlobe as shown in Figure 10-9.

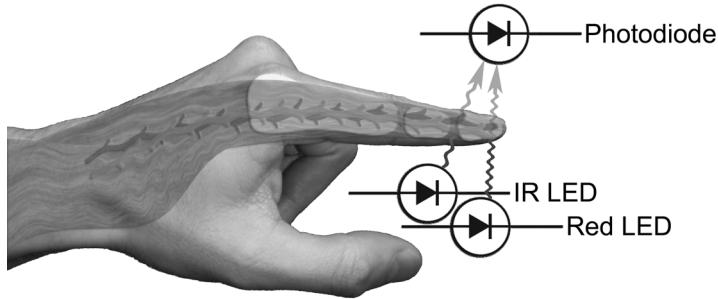
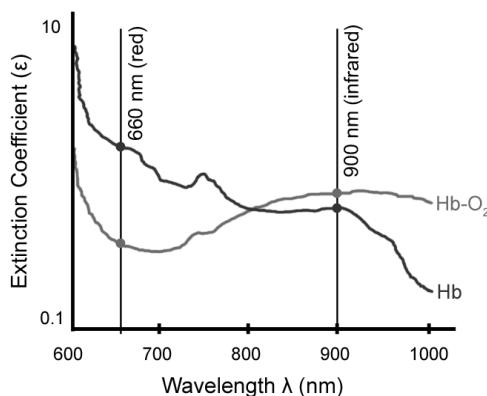


Figure 10-9 Two-wavelength pulse oximetry in the finger

Pulse oximetry is a technique that relies on the measurement of the absorption of electromagnetic radiation (light) due to its interaction with the hemoglobin. The *extinction coefficient* is a measure of the absorption of a solution per unit path length and concentration. In the case of oxyhemoglobin (Hb-O_2) and deoxyhemoglobin (Hb), the extinction coefficient is different at specific wavelengths as shown in Figure 10-10.

In pulse oximetry, when optimizing sensitivity, the best wavelengths to use are the 660 nm (red) and the 900 nm (infrared) not only because of the large difference in the extinction coefficient of oxyhemoglobin and deoxyhemoglobin at 660 nm but also because the extinction coefficients at 900 nm are about the same. Figure 10-10 clearly shows that when the blood is saturated with oxygen (Hb-O_2), it presents a bright red color because there is not much absorption at 660 nm. Similarly, when the blood is deoxygenated (Hb) it turns into a dark color because there is a lot of absorption at 660 nm. Meanwhile, the absorption of both oxyhemoglobin and deoxyhemoglobin at 900 nm is about the same.

Figure 10-10 Extinction coefficient of Hb-O_2 and Hb at different wavelengths

The value calculated by a pulse oximeter is known as oxygen saturation (SpO_2) and by definition is calculated as the oxygen concentration expressed as a percentage of the maximum concentration of oxygen that can be carried by the blood:

$$\text{SpO}_2 = \frac{\text{Concentration of Oxyhemoglobin}}{\text{Total concentration of Hemoglobin}} \times 100$$

$$\text{SpO}_2 = \frac{c_{\text{Hb-O}_2}}{c_{\text{Hb-O}_2} + c_{\text{Hb}}} \times 100$$

Figure 10-11 shows that as the beam of light from the LEDs travels through the finger tissues, the light intensity for the most part decreases logarithmically with the path length and the concentration of the molecule in question (Hb and Hb-O₂) according to the *Beer-Lambert law*:

$$\epsilon * c * l = -\log_{10} (I_{\text{final}} / I_{\text{init}})$$

Where:

ϵ : Extinction coefficient (see Figure 10-10).

c: Concentration (more molecules of Hb absorbing will result in more photons absorbed).

l: Path length (longer path lengths will result in more photons absorbed).

I_{init} : Initial intensity of the light (incident light).

I_{final} : Final intensity of the light (transmitted light).

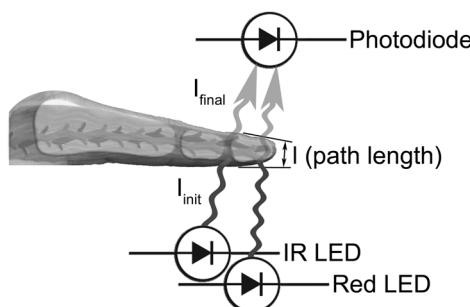


Figure 10-11 **Beer Lambert law**

Because $10^c / 10^d = 10^{c-d}$ then $\log_{10}(x/y) = \log_{10}(x) - \log_{10}(y)$ and:

$$\varepsilon * c * l = -\log_{10}(I_{final}) + \log_{10}(I_{init})$$

$$c = \frac{-\log_{10}(I_{final}) + \log_{10}(I_{init})}{\varepsilon * l}$$

Thus the concentrations for the hemoglobin and oxyhemoglobin being transilluminated by the red and infrared light respectively are:

$$c_{Hb} = \frac{-\log_{10}(I_{red-final}) + \log_{10}(I_{red-init})}{\varepsilon_{Hb} * l}$$

$$c_{HbO_2} = \frac{-\log_{10}(I_{ir-final}) + \log_{10}(I_{ir-init})}{\varepsilon_{Hb-O_2} * l}$$

If the oxygen saturation (SpO_2) is defined as the ratio:

$$SpO_2 = \frac{\text{Concentration of Oxyhemoglobin}}{\text{Total concentration of Hemoglobin}} * 100 \%$$

Then SpO_2 expressed in terms of the light intensities, path lengths and extinction coefficients is:

$$SpO_2 = \frac{\frac{-\log_{10}(I_{red-final}) + \log_{10}(I_{red-init})}{\varepsilon_{Hb-O_2} * l}}{\frac{-\log_{10}(I_{red-final}) + \log_{10}(I_{red-init})}{\varepsilon_{Hb-O_2} * l} - \frac{-\log_{10}(I_{ir-final}) + \log_{10}(I_{ir-init})}{\varepsilon_{Hb} * l}}$$

The extinction coefficients are constants, the path length is the same, the final intensities are those measured at the output of the photodiode and the initial intensities can be assumed to be the level of the DC component just because over 90% of the absorption is due to the skin and tissues as we will explain in the next paragraph.

Let's assume we measure the final intensity of an LED that is turned on for at least two seconds and plot the initial and final intensities over time as shown in Figure 10-12. Notice that most of the absorption is due to the interaction of the light with the skin and tissue, while only a small part is due to the venous and arterial blood. Pulse oximetry relies on the

AC component of the final intensity of the light which is due to the absorption by the oxyhemoglobin that gets added at every heartbeat. This is the reason why the initial intensities can be assumed to be the level of the DC component.

That and all the assumptions mentioned before, specially the fact that the SpO_2 percentage is just a ratio can simplify the formula to:

$$\text{SpO}_2 = \frac{\log_{10}(I_{\text{red-final}})}{\log_{10}(I_{\text{ir-final}})} * 100 \%$$

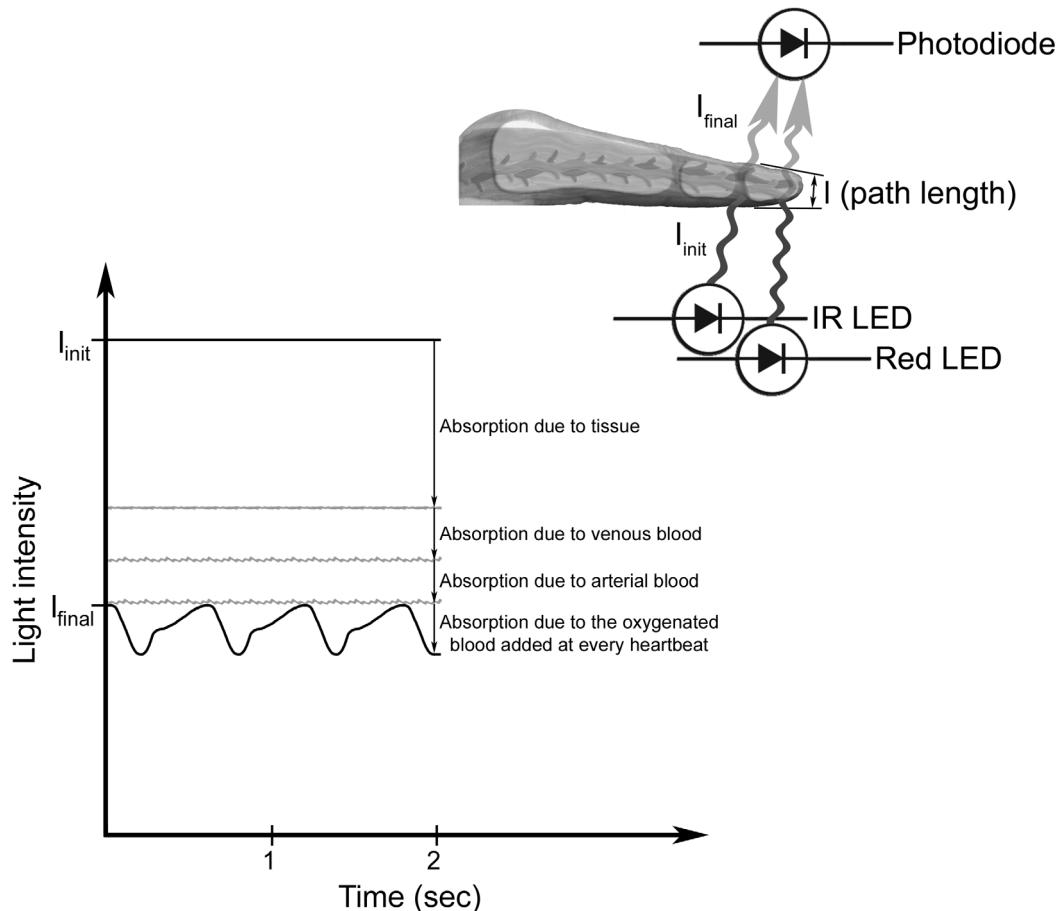


Figure 10-12 Light intensity over time

10-3 PULSE OXIMETER DESIGN

The analog front end of a pulse oximeter is fairly simple and includes an instrumentation circuit similar to the one used in the previous chapter which converted current to voltage known as transimpedance amplifier. The output of the photodiode (anode and cathode) is connected to the input of the transimpedance amplifier and an analog signal multiplexor routes the output to one of two channels (Red or IR LED).

Each analog channel has a low pass filter with a cutoff frequency of 6Hz in order to extract the DC component as shown in Figure 10-13.

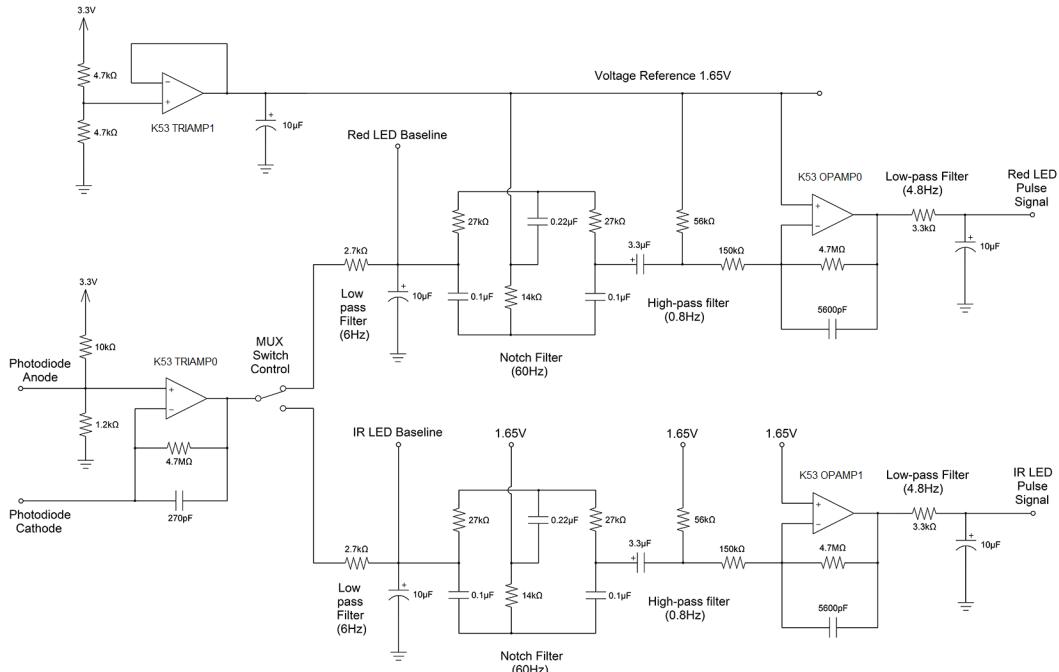


Figure 10-13 Pulse Oximeter Signal Conditioning

In similar fashion to the heart rate monitor's processing of the baseline signal in Chapter 8, “ECG / Heart Rate Monitor” on page 111, the signal out of the low-pass filter needs to be monitored and the intensity of the LEDs needs to be adjusted depending on the level of this baseline signal.

A notch filter attenuates any 60Hz noise and a band-pass filter not only keeps the signal within the bandwidth of interest (0.8Hz - 4.8Hz) but also provides the last stage of amplification.

The other part of the analog front includes an H-Bridge based LED driver circuit shown in Figure 10-14.

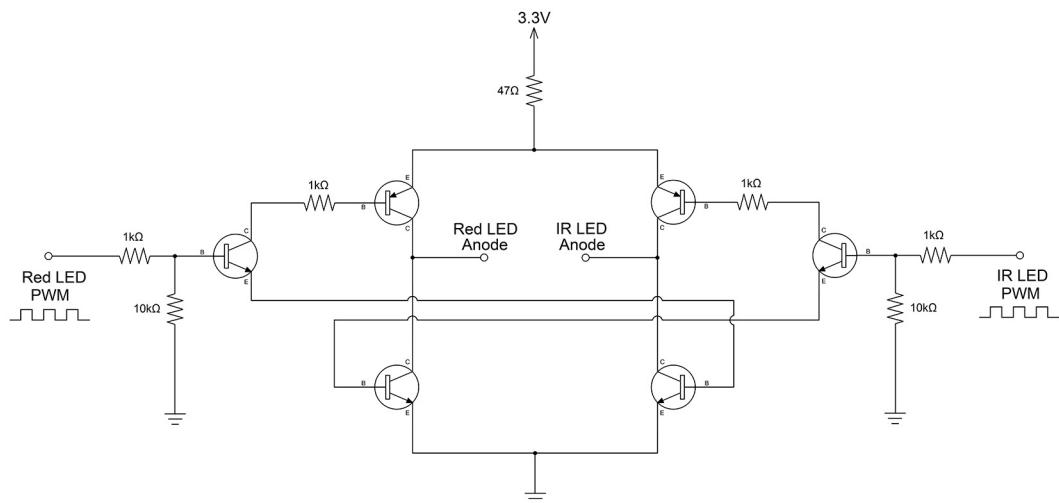


Figure 10-14 H-Bridge LED driver

The intensity of the LEDs is controlled by the duty cycle of a PWM signal that feeds this H-Bridge circuit. This set of bipolar transistors is arranged in such a way that a change in the PWM signal's duty cycle results in a proportional change in the current sourced to the anode of the LEDs.

Freescale offers an analog module with a medical connector compatible with the TWR-K53N512 medical board that implements this type of analog front end called MED-SPO2.

The complete system including the TWR-K53N512 and MED-SPO2 is illustrated as a block diagram in Figure 10-15. The two pushbuttons onboard the TWR-K53N512 are used to either start or stop the application and the LEDs are used to indicate an error by turning on the orange LED and to indicate the detection of the heartbeat by blinking the yellow LED.

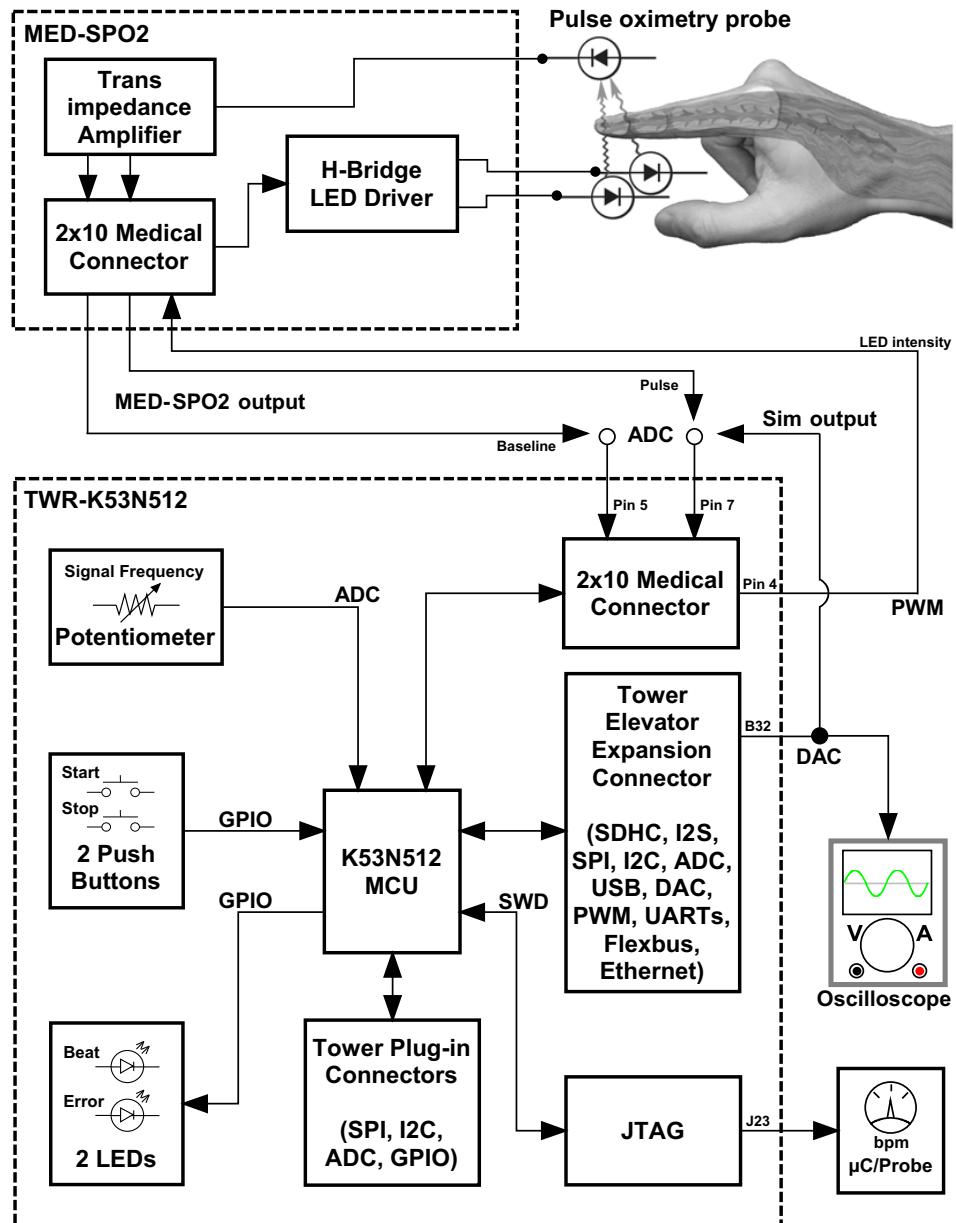


Figure 10-15 Pulse Oximeter Block Diagram

The 2x10 medical connector of the MED-SPO2 not only provides connectivity with the TWR-K53N512 board to make the Baseline and Pulse signals available to the ADC modules but also allows the use of the on-chip op-amps and transimpedance amplifiers of the Kinetis 32-bit microcontroller to implement the required signal conditioning. The rest of the signals have been omitted from the block diagram for the sake of simplicity but Table 10-1 shows the signal present in each pin of the medical connector.

MED-SPO2 Signal	Pin		MED-SPO2 Signal
VDD	1	2	GND
MUX Switch Control	3	4	PWM Signal
Baseline Signal	5	6	Not connected
Pulse Signal	7	8	Not connected
OPAMP0 VOUT0	9	10	OPAMP1 VOUT1
OPAMP0 INN0-	11	12	OPAMP1 INN1-
OPAMP0 INP0+	13	14	OPAMP1 INP1+
TRIAMP0 INP0+	15	16	TRIAMP1 INP1+
TRIAMP0 INN0-	17	18	TRIAMP1 INN1-
TRIAMP0 VOUT0	19	20	TRIAMP1 VOUT1

Table 10-1 Medical Connector 2x10 Pin Header Connections

Notice from the block diagram in Figure 10-15 that in the absence of the MED-SPO2 module you can still run the application by using the simulated pulse signal coming out of the DAC module by connecting a jumper wire between pin 7 of the medical connector and pin B32 of the elevator module in the tower system. The potentiometer is used in such situation to increase or decrease the simulated heart rate.

The communication with μC/Probe which will be used to display the results of the pulse oximeter is via the J-Link debug probe from Segger through the JTAG connector labeled as J23.

Figure 10-16 shows the MED-SPO2 module connected to the TWR-K53N512 controller in a tower system configuration.

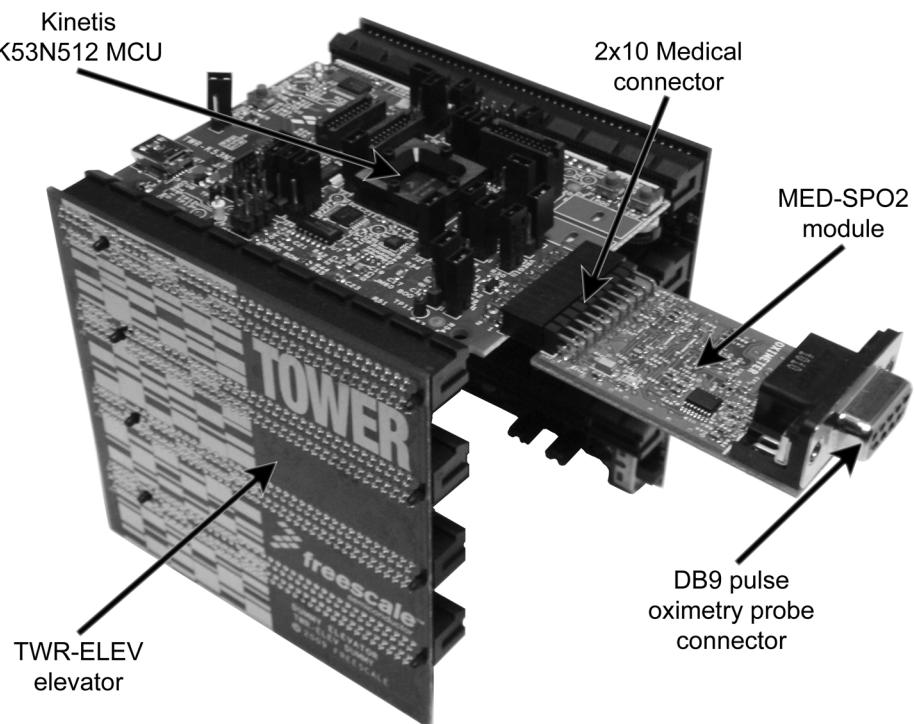


Figure 10-16 MED-SPO2 and TWR-K53N512

10-4 RUNNING THE EXAMPLE PROJECT

This section describes the steps you need to follow to run the pulse oximeter example. Start by connecting the MED-SPO2 module to the TWR-K53N512 controller through the 2x10 medical connector as shown in Figure 10-16.

The MED-SPO2 module is compatible with the 9-pin D-SUB male connector pulse oximeter probe from Nellcor shown in Figure 10-17. The wavelengths of the light emitted by this sensor are 660 nm for the Red LED and 900 nm for the IR LED just as previously illustrated in Figure 10-10.

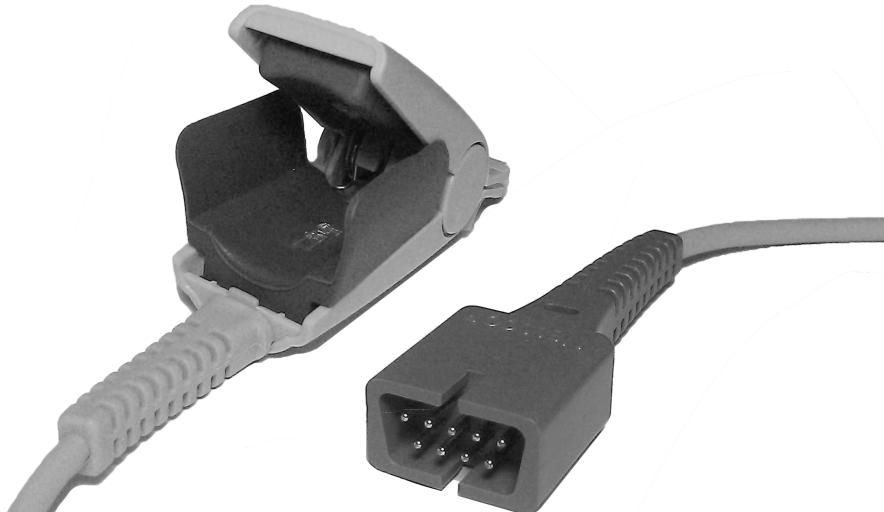


Figure 10-17 9-pin connector Pulse Oximetry Probe from Nellcor

Assuming you have followed all the setup steps described in Chapter 7, “Setup” on page 101, proceed to connect one end of the Segger's J-Link for ARM debug probe to the JTAG connector on J23 of the TWR-K53N512 and the other end of the probe to any USB port available in your PC.

Run IAR's Embedded Workbench for ARM (EWARM) and open the workspace at:

\$\Micrium\Examples\Freescale\TWR-K53N512\OS2-TCPIP-POX\IAR\
OS2-TCPIP-POX.eww

Chapter 10

The workspace window shows all of the files in the project which are sorted into groups represented by folder icons. Figure 10-18 shows the project files for OS2-TCPIP-POX in the workspace explorer window.

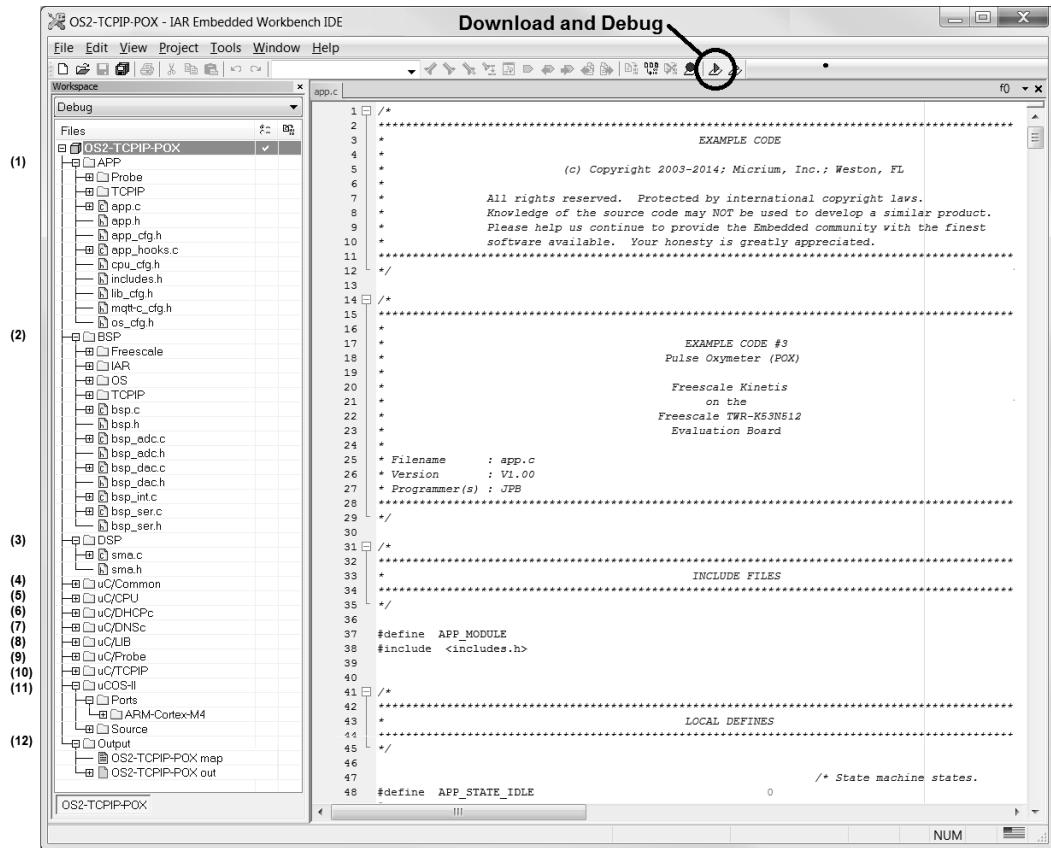


Figure 10-18 Running the project with EWARM

- F10-18(1) The **APP** group includes all of the application files for this example, including the header files used to configure the application.
- F10-18(2) The **BSP** group contains the files that comprise the board support package. The board support package includes the code used to control the peripherals on the board. For this example, the software to control the LEDs, pushbuttons, ADCs, DACs, operational amplifiers and transimpedance amplifiers will be used. The subgroup **Freescale** includes the header files for the Kinetis K50 family of microcontrollers.

-
- F10-18(3) The **DSP** group contains the files that define some of the Digital Signal Processing functions called by the application.
 - F10-18(4) The **uC/Common** group contains the kernel abstraction layer.
 - F10-18(5) The **uC/CPU** group contains the μ C/CPU source code files.
 - F10-18(6) The **uC/DHCPC** group contains the header files for the DHCP client that allows you to negotiate an IP address with your network's router.
 - F10-18(7) The **uC/DNSc** group contains the header files for the DNS client that allows you to translate domain names into IP addresses.
 - F10-18(8) The **uC/LIB** group contains the μ C/LIB source code files.
 - F10-18(9) The **uC/Probe** group contains the source code to enable communication with μ C/Probe via TCP/IP.
 - F10-18(10) The **uC/TCP-IP** group contains the basic header files for the TCP/IP stack.
 - F10-18(11) The **uC/OS-II** group contains the μ COS-II source code.
 - F10-18(12) The **Output** group contains the files generated by the compiler/linker.

Start the debugger by clicking the “Download and Debug” icon as shown in Figure 10-18. The application is programmed to the internal Flash of the K53N512 microcontroller and the debugger starts. The code automatically starts executing and stops at the `main()` function in the `app.c` file.

Click the “Go” icon in IAR Embedded Workbench’s debugging tools to run the application as shown in Figure 10-19.

Chapter 10

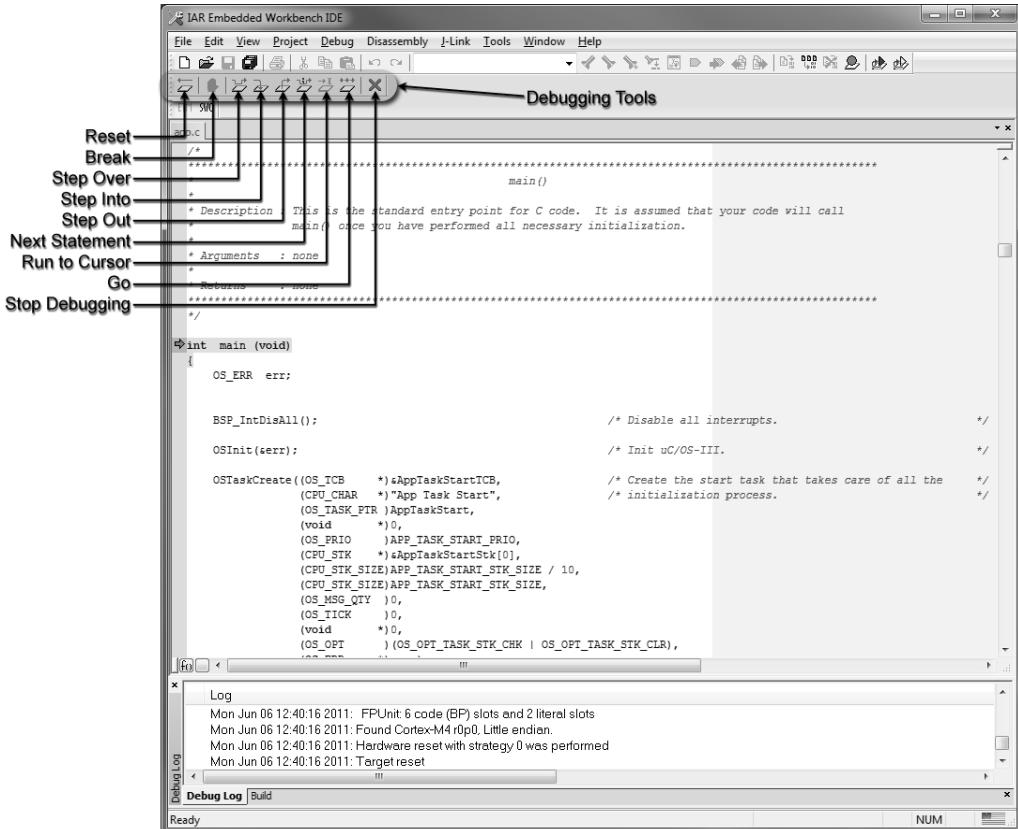


Figure 10-19 Downloading the Application and Starting the Debugger

You can test the application by starting μC/Probe and open the workspace for the Pulse Oximeter at:

```
$\Micrium\Examples\Freescale\TWR-K53N512\OS2-TCPIP-POX\IAR\
OS2-TCPIP-POX.wspx
```

After μC/Probe starts, click the run button (green triangle). Once μC/Probe is running, you will see a screen similar to the one shown in Figure 10-20.

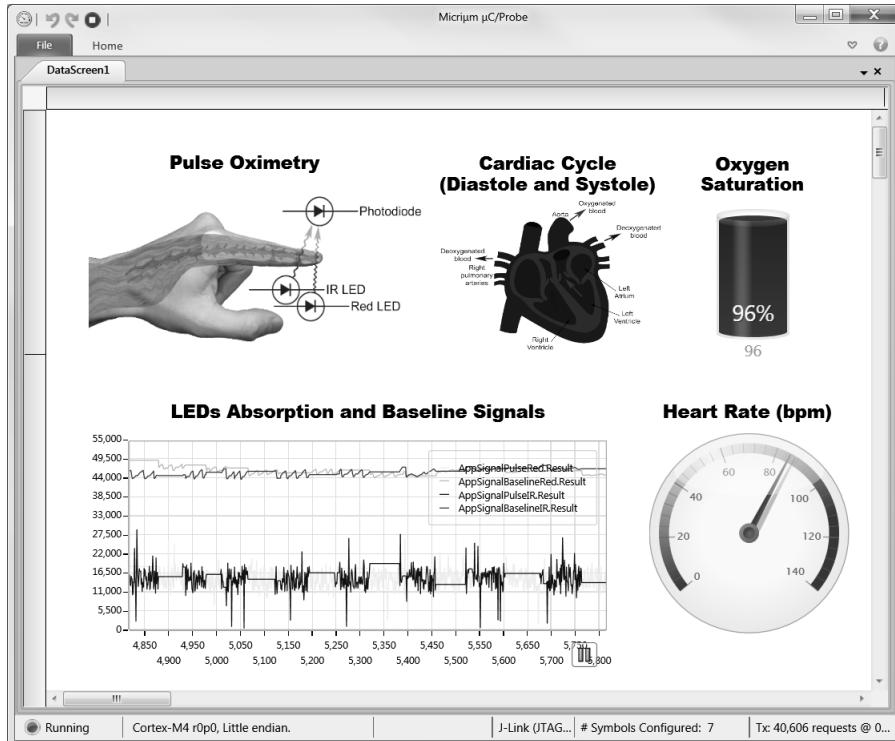


Figure 10-20 µC/Probe Pulse Oximeter Dashboard

Press the pushbutton on the TWR-K53N512 board labeled as SW1 to start the application.

The application will run indefinitely until you press the pushbutton on the TWR-K53N512 board labeled as SW2 to stop the application.

Place your finger in the pulse oximetry probe and watch in real time how the baseline signals for the Red and IR LEDs move around in an effort to optimize the intensity of the LEDs for the color of your skin and the size of your finger. It takes a few seconds for the system to adjust the intensities and then you should be able to see your heart rate displayed in the gauge and your oxygen saturation in the vertical meter. An animation of the heart during the systole and diastole phases of your own heart is shown to demonstrate one of the features of µC/Probe.

10-5 HOW THE CODE WORKS

The pulse oximeter application consists of four different tasks:

- User IF task: The user interface task monitors the state of the two pushbuttons onboard the TWR-K53N512. The pushbuttons are used to either start or stop the application. The task is defined by `AppTaskUserIF()` in `app.c`.
- Sim task: The simulator task monitors the state of the potentiometer and updates the frequency of a pulse simulated signal generated by the DAC according to the position of the potentiometer. The task is defined by `AppTaskSim()` in `app.c`.
- DAQ task: The data acquisition task is the main task in the application and implements the state machine that processes the analog input samples to calculate the heart rate and the oxygen saturation. The task is defined by `AppTaskDAQ()` in `app.c`.
- Heartbeat task: The heartbeat task is responsible for controlling the heart animation in µC/Probe by updating the value of the global variable `AppCardiacCycleState` that can take one of three states: Diastole, Systole or None. The heartbeat task is defined by `AppTaskHeartbeat()` in `app.c`

The interaction among the different tasks is facilitated by the use of µC/OS-II's semaphores and message queues and it is illustrated in Figure 10-21.

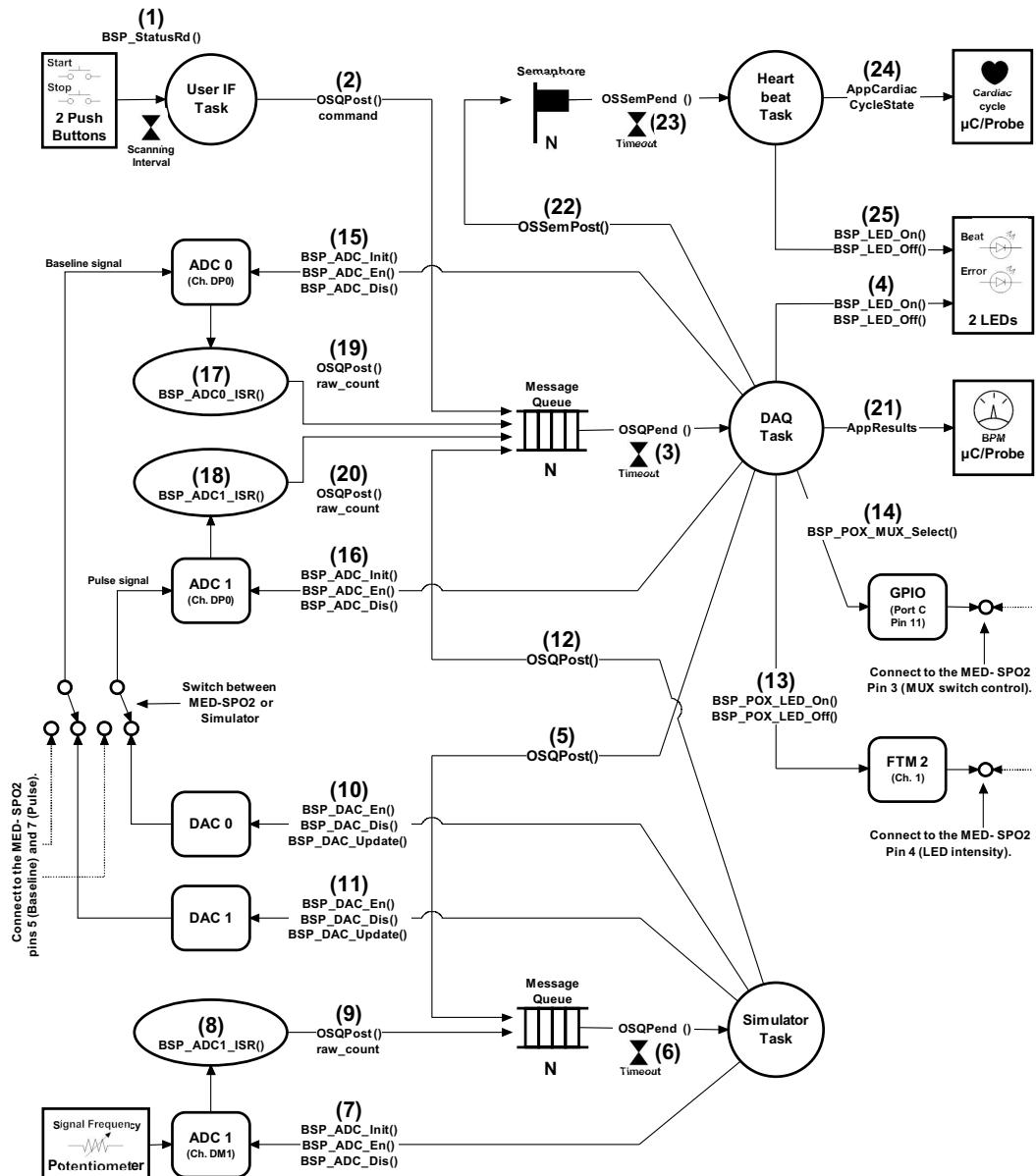


Figure 10-21 Interaction of Application Tasks

-
- F10-21(1) The state of the push buttons is monitored by the user IF task at a scanning interval defined by `BSP_STATUS_CHECK_INTERVAL`. See `AppTaskUserIF()` in `app.c` and `BSP_StatusRd()` in `bsp.c`.
 - F10-21(2) Depending on the pushbutton pressed by the user a message from the user IF task to either start or stop the application is posted to the DAQ task's built-in message queue by calling `OSTaskQPost()`. See `AppTaskUserIF()` in `app.c`.
 - F10-21(3) The DAQ task can receive data messages directly from the ADCs ISRs or command messages from the other two tasks. When `OSTaskQPend()` is called, the message is retrieved and processed in two different ways depending on the sender and the type of message. If the message comes from one of the ADC's ISR then the message is processed by calling the function `AppTaskDAQ_ProcessData()` and if the message comes from any other task then the message is processed by calling the function `AppTaskDAQ_ProcessCmd()` in `app.c`.
 - F10-21(4) If the message retrieved by the DAQ task is a command to start the pulse oximetry monitoring then the DAQ task turns off the LEDs by calling the function `BSP_LED_Off()` in `bsp.c`.
 - F10-21(5) If the message retrieved by the DAQ task is a command to start the pulse oximetry monitoring in simulation mode then the DAQ task posts a message to the sim task's built-in message queue to start the simulator. See `AppTaskDAQ()` in `app.c` and `AppTaskDAQ_ProcessCmd()` in `app.c`.
 - F10-21(6) In similar fashion, the sim task can receive messages directly from the ADC1's ISR or from the DAQ task. When `OSTaskQPend()` is called, the message is retrieved and processed in different ways depending on the sender. The sim task retrieves the received message from its message queue and proceeds according to the command issued by the DAQ task as follows:
 - F10-21(7) If the message contains a start command then the sim task starts ADC1 channel DM1 in order to read the potentiometer. See `AppTaskSim()` in `app.c`.
 - F10-21(8) The CPU is interrupted by the completion of a conversion of ADC1 Channel DM1 and the interrupt is handled by `BSP_ADC1_ISR()` in `bsp_adc.c`.

-
- F10-21(9) The ADC1 ISR posts the result of the conversion in the form of a 16-bit raw count to the sim task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC1_ISR()` in `bsp_adc.c`.
 - F10-21(10) The sim task retrieves the message from the queue and configures the DAC0 to output a simulated pulse waveform. See `AppTaskSim()` in `app.c`.
 - F10-21(11) The sim task retrieves the message from the queue and configures the DAC1 to output a simulated baseline waveform at a rate proportional to the raw count of the ADC1 channel DM1. See `AppTaskSim()` in `app.c`.
 - F10-21(12) The sim task posts a message to the DAQ task's built-in message queue notifying the DAQ task that the simulator is running. See `AppTaskSim()` in `app.c`.
 - F10-21(13) The DAQ task retrieves the message from the queue and initializes FlexTimer FTM2 to output a PWM signal to drive the Red and IR LEDs. See `AppTaskDAQ()` and `AppTaskDAQ_ProcessCmd()` in `app.c`.
 - F10-21(14) The DAQ task retrieves the message from the queue and sets/clears pin 11 of the GPIO in Port C to control the selection of either the Red or IR analog channel in the MUX onboard the MED-SPO2 module. See `AppTaskDAQ()` and `AppTaskDAQ_ProcessCmd()` in `app.c`.
 - F10-21(15) The DAQ task retrieves the message from the queue and enables ADC0 channel DP0 to convert the baseline signal. See `AppTaskDAQ()`, `AppTaskDAQ_ProcessCmd()` and `AppTaskDAQ_ExecCmd()` in `app.c`.
 - F10-21(16) The DAQ task retrieves the message from the queue and enables ADC1 channel DP0 to convert the pulse signal. See `AppTaskDAQ()`, `AppTaskDAQ_ProcessCmd()` and `AppTaskDAQ_ExecCmd()` in `app.c`.
 - F10-21(17) The CPU is interrupted by the completion of a conversion of ADC0 channel DP0 and the interrupt is handled by `BSP_ADC0_ISR` in `bsp_adc.c`.
 - F10-21(18) About the same time, the CPU is interrupted by the completion of a conversion of ADC1 channel DP0 and the interrupt is handled by `BSP_ADC1_ISR()` in `bsp_adc.c`.

-
- F10-21(19) The ADC0 ISR posts the result of the conversion in the form of a 16-bit raw count to the DAQ task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC0_ISR()` in `bsp_adc.c`.
 - F10-21(20) The ADC1 ISR posts the result of the conversion in the form of a 16-bit raw count to the DAQ task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC1_ISR()` in `bsp_adc.c`.
 - F10-21(21) The DAQ task retrieves the message from the queue and uses the raw counts from ADC0 and ADC1 to calculate the heart rate, the oxygen saturation and display the values in μ C/Probe by calling the function `AppTaskDAQ_ProcessData()` and `AppCalcResults()` in `app.c`.
 - F10-21(22) The DAQ task sends a signal to the heartbeat task through its local semaphore by calling the function `OSTaskSemPost()`. The signal is to notify the heartbeat task that a new heart rate has been calculated. See `AppCalcResults()` in `app.c`.
 - F10-21(23) The heartbeat task normally waits for a signal from the DAQ task in order to update a timer that drives the cardiac cycle animation in μ C/Probe. See `AppTaskHeartbeat()` in `app.c`.
 - F10-21(24) The heartbeat task updates the global variable `AppCardiacCycleState` toggling between diastolic and systolic states at the same rate calculated by the DAQ task. See `AppTaskHeartbeat()` in `app.c`.
 - F10-21(25) The heartbeat task toggles the LED at the same rate of the heartbeat by calling the function `BSP_LED_On()` and `BSP_LED_Off()` in `bsp.c`.

10-5-1 BIOMEDICAL SIGNAL ANALYSIS

The algorithm to detect pulses relies on the presence in the signals coming out of the Red and Infrared pulse of consecutive summit and valley peaks that represent every heartbeat (pulse).

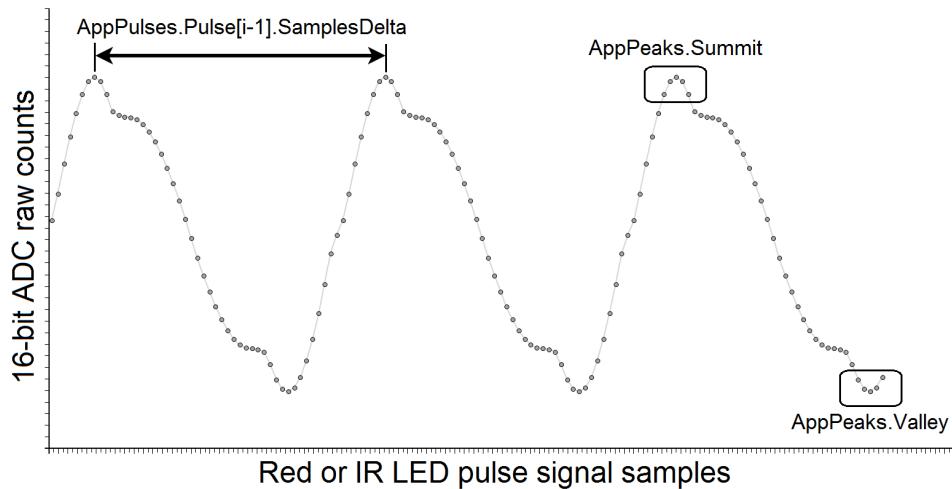


Figure 10-22 **Pulse Detection Algorithm**

As illustrated in Figure 10-22, the DAQ task keeps two sliding windows of five samples:

```
AppPeaks.Summit.Point[5]
AppPeaks.Valley.Point[5]
```

Listing 10-1 **DAQ task's sliding windows for peak detection**

The DAQ tasks keeps monitoring the first 5-point sliding window until a summit is detected. The number of samples between pulses counter at `AppPulses.Pulse[i-1].SamplesDelta` gets reset every time a summit is detected. The counter keeps counting samples and monitoring the second 5-point sliding window until a valley gets detected. At that point, it is assumed that a pulse has been detected and the time-domain features of the pulse are stored into an array of pulses for each signal (Red and Infrared):

```
AppPulses.Pulse[i].Pk2PkAmplitudeCnt  
AppPulses.Pulse[i].SamplesDelta
```

Listing 10-2 DAQ task's arrays of Red and IR pulses

Whenever a minimum number of pulses threshold defined by APP_MIN_HEARTBEATS_PER_CALC is reached, the heart rate and oxygen saturation is calculated by inserting all the time domain features into a histogram and calculating the *mode*, which in statistics it is defined as the value that occurs most frequently in a data set.

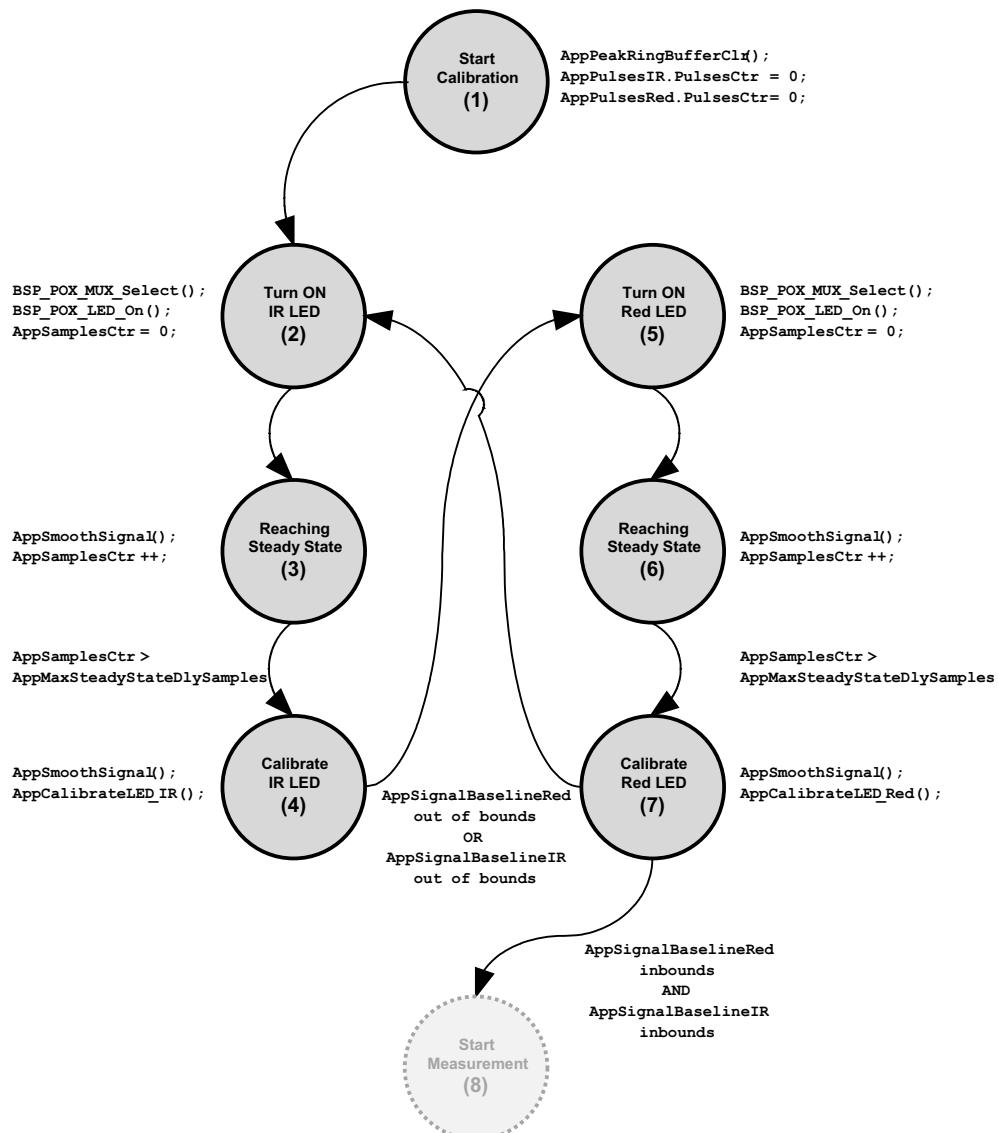
The calculation of the most frequent peak-to-peak amplitude and the most frequent number of samples between pulses eliminates any errors due to artifacts and the heart rate and oxygen saturation can be calculated by:

```
AppResults.HeartRate = 60 * (APP_SAMPLING_RATE) / SamplesDeltaMode;  
  
AppResults.OxygenSaturation = log10(Peak2PeakRedMode) / log10(Peak2PeakIRMode);
```

Listing 10-3 Heart Rate and SpO₂ Calculation

10-5-2 DATA PROCESSING STATE MACHINE

The state machine that processes every sample in AppTaskDAQ_ProcessData() is illustrated in Figure 10-23:



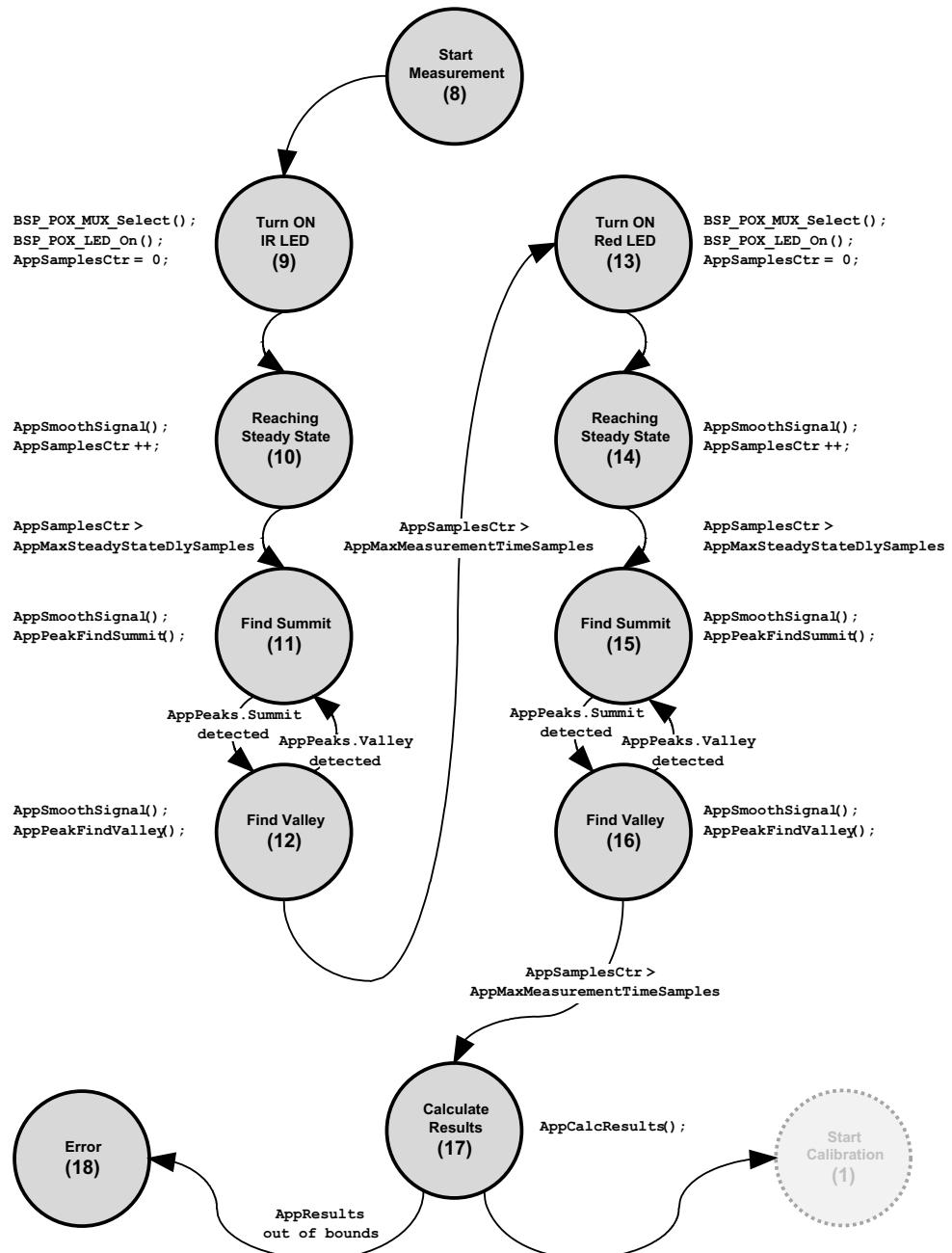


Figure 10-23 Data Processing State Machine

-
- F10-23(1) The first set of states from (1) to (7) are meant for calibration of the intensity of the IR and Red LEDs. In this state, the calibration process gets started by calling `AppPeakRingBufferClr()` with the appropriate arguments to initialize the global variables used by the peak detection algorithm.
 - F10-23(2) In this state, the IR LED analog channel illustrated in Figure 10-13 is selected from the multiplexor by calling `BSP_MUX_Select()` with the appropriate arguments and the IR LED is turned on by calling `BSP_POX_LED_On()` with the right arguments.
 - F10-23(3) It takes some time for the signal conditioning circuit shown in Figure 10-13 to reach a steady state condition and in this state, both the Baseline and Pulse signals for the IR LED are smoothed by applying a simple moving average filter. The number of samples processed is kept in `AppSamplesCtr` in order to count the number of samples until it reaches the steady state threshold time.
 - F10-23(4) In this state, the intensity of the IR LED is optimized for the skin color and the size of the finger by calling the function `AppCalibrateLED_IR()`. The function makes adjustments to the duty cycle of the PWM signal that drives the IR LED in such a way that the Baseline signal falls in the middle of a range of amplitudes defined in volts by `APP_BASELINE_UPPER_LIMIT_MIN_VOLTS` and `APP_BASELINE_UPPER_LIMIT_MAX_VOLTS`. That ensures that the Pulse signal from the IR LED does not get clipped due to over amplification.
 - F10-23(5) In a similar fashion to the IR LED, in this state, the Red LED analog channel is selected from the multiplexor by calling `BSP_MUX_Select()` with the appropriate arguments and the Red LED is turned on by calling `BSP_POX_LED_On()` with the corresponding arguments.
 - F10-23(6) In order to reach a steady state condition, in this state, both the Baseline and Pulse signals for the Red LED are smoothed by applying a simple moving average filter. The number of samples processed is kept in `AppSamplesCtr` in order to count the number of samples until it reaches the steady state threshold time.
 - F10-23(7) In this state, the intensity of the Red LED is optimized for the skin color and the size of the finger by calling the function `AppCalibrateLED_Red()`. The function makes adjustments to the duty cycle of the PWM signal that drives the

Red LED in such a way that the Baseline signal falls in the middle of a range of amplitudes defined in volts by APP_BASELINE_UPPER_LIMIT_MIN_VOLTS and APP_BASELINE_UPPER_LIMIT_MAX_VOLTS. That ensures that the Pulse signal from the Red LED does not get clipped due to over amplification. The process continues to calibrate the intensities by going back and forth between the IR and Red LEDs until both are within the appropriate thresholds.

- F10-23(8) The next state after the intensities of the LEDs are optimized is to start the measurement.
- F10-23(9) In this state, the IR LED analog channel illustrated in Figure 10-13 is selected from the multiplexor by calling `BSP_MUX_Select()` with the appropriate arguments and the IR LED is turned on by calling `BSP_POX_LED_On()` with the right arguments.
- F10-23(10) It takes some time for the signal conditioning circuit illustrated in Figure 10-13 to reach a steady state condition and in this state, both the Baseline and Pulse signals for the IR LED are smoothed by applying a simple moving average filter. The number of samples processed is kept in `AppSamplesCtr` in order to count the number of samples until it reaches the steady state threshold time.
- F10-23(11) Because a pulse is defined as a consecutive summit and valley as illustrated in Figure 10-22, in this state, every sample is inserted into the 5-sample sliding window in order to detect a summit.
- F10-23(12) In this state, every sample is inserted into the 5-sample sliding window in order to detect a valley. If the consecutive summit and valley form a valid pulse as illustrated in Figure 10-22, then the time-domain features of the pulse are inserted into the array of pulses `AppPulsesIR`. The process goes back and forth between finding summits and valleys for the IR LED until the maximum measurement time is reached.
- F10-23(13) In this state, the Red LED analog channel is selected from the multiplexor by calling `BSP_MUX_Select()` with the appropriate arguments and the Red LED is turned on by calling `BSP_POX_LED_On()` with the right arguments.

-
- F10-23(14) It is necessary to wait some time in order to reach a steady state condition and in this state, both the Baseline and Pulse signals for the Red LED are smoothed by applying a simple moving average filter. The number of samples processed is kept in `AppSamplesCtr` in order to count the number of samples until it reaches the steady state threshold time.
 - F10-23(15) In a similar fashion to the IR LED, in this state, every sample is inserted into the 5-sample sliding window in order to detect a summit.
 - F10-23(16) In this state, every sample is inserted into the 5-sample sliding window in order to detect a valley. If the consecutive summit and valley form a valid pulse as illustrated in Figure 10-22, then the time-domain features of the pulse are inserted into the array of pulses for the Red LED `AppPulsesRed`. The process goes back and forth between finding summits and valleys for the Red LED until the maximum measurement time is reached.
 - F10-23(17) The results are calculated based on the two sets of pulses stored in `AppPulsesIR` and `AppPulsesRed` by calling `AppCalcResults()` which implements the algorithm described in section 10-5-1 “Biomedical Signal Analysis” on page 205. The results are stored in `AppResults` and the whole process is repeated over again starting from the calibration states.
 - F10-23(18) An error occurs if the results are out of the valid ranges defined in beats-per-minute by `APP_MIN_HEARTRATE_BPM` and `APP_MAX_HEARTRATE_BPM` for the heart rate and the ranges defined by `APP_MIN_OXYGEN_SATURATION` and `APP_MAX_OXYGEN_SATURATION` for the oxygen saturation.

10-6 SUMMARY

Similar to the examples presented in Chapter 8, “ECG / Heart Rate Monitor” on page 111 and Chapter 9, “Blood Glucose Meter” on page 149 this chapter provides an easy to follow explanation of the anatomy and physiology of the respiratory system as far as the exchange of gases is concerned. In a similar fashion, it explained the theory of operation of a pulse oximeter including the hardware and software.

This example application demonstrated how simple it is to implement a pulse oximeter using a combination of Micrium's µC/OS-II and Freescale hardware. It also demonstrated two of the most important features of µC/OS-II: *Semaphores* and *Message Queues*.

Signaling a task through its built-in semaphore is a typical method of synchronization, and in this example we used the heartbeat task's built-in semaphore to synchronize the activities of the DAQ task and the heartbeat task in the same way we did in Chapter 8, "ECG / Heart Rate Monitor" on page 111. The heartbeat task is signaled by the DAQ task when a new heart rate result has been calculated and depending on the heartbeat task priority, the scheduler is run. The heartbeat task may then update the cardiac cycle animation with the new heart rate result.

Message queues are built into each task and the user can send messages directly to a task from another task or an ISR. Task message queues allow the developer to encapsulate each task's functionality into a clean and simple message-based API. For example, we used the DAQ task's built-in message queue as a means to receive different types of commands from other tasks to perform an action like start or stop the simulator, start or stop the data acquisition and process samples from the ADC's ISRs.

This example delivers a solid platform to create a commercial medical product based on a task-oriented approach. You can add as many tasks as memory resources permit in order to support new features. Imagine you want to be able to send the oxygen saturation and heart rate readings to a fitness equipment, smartphone or tablet PC for display and monitoring purposes. Whether it is USB or bluetooth, chances are that you may need at least two more tasks to handle the reception and transmission of packets and one more task to handle all time-outs related to the communication protocol.

Most commercial pulse oximeters are stand-alone and come with a built-in OLED display. This type of display control is also a great candidate to encapsulate into a task and use the task message queue to receive commands to update the display.

Freescale and Micrium support the addition of other functionality to this pulse oximeter by offering more tower system compatible peripherals and software stacks. Visit the Freescale and Micrium websites to learn more about other products that can help you take your design to the next level.

Chapter

11

Blood Pressure Monitor

A blood pressure monitor is a device that measures the arterial blood pressure at the phase during the cardiac cycle when the blood is pumped out of the heart (*systolic pressure*) and when the blood fills the heart (*diastolic pressure*). The device is also capable of measuring the heart rate as a byproduct of the blood pressure measurement process. Figure 11-1 shows an example of a commercial blood pressure monitor displaying the systolic and diastolic blood pressures in millimeters of mercury (mmHg) at the top and the heart rate in beats per minute at the bottom.

This chapter demonstrates a basic implementation of a blood pressure monitor built using µC/OS-II and Freescale products.

In a similar manner to the previous examples, this chapter starts with an illustrated description of the natural mechanism of blood pressure regulation. The next section continues the introduction, but with emphasis on the acquisition and calculation of the arterial blood pressure and presents the design of a blood pressure monitor (BPM). The last part of the chapter deals with using the tools to run the example and a description of how the code works.



Figure 11-1 Commercial blood pressure monitor

11-1 BLOOD PRESSURE

Section 4-01 described the heart as a four-chambered pump responsible for pumping blood throughout the body. A series of coordinated and repeating electrical and mechanical events known as the cardiac cycle include two phases known as *systole* and *diastole*.

Figure 11-2 shows the heart during systole. The pressure generated by the ventricular contraction forces blood out of the heart through the aortic valve and into the aorta. The pressure gets distributed throughout the walls of the arteries as the blood flows through them.

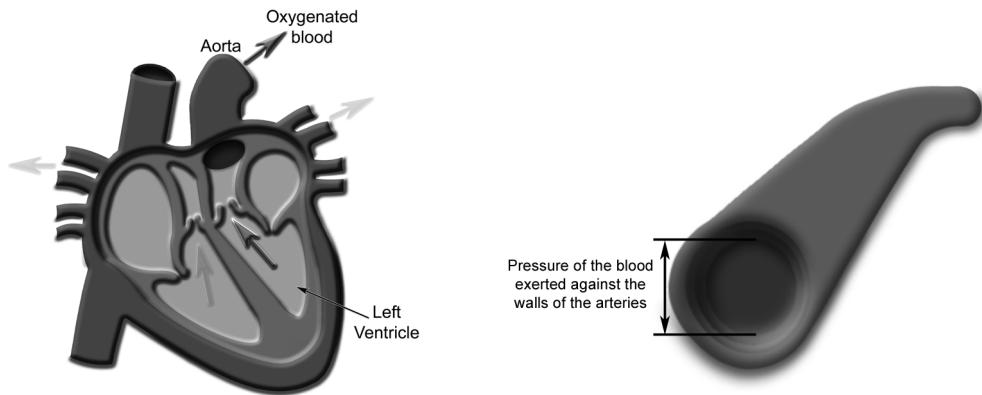


Figure 11-2 Heart during systole

Figure 11-3 shows the heart during diastole. The pressure inside the ventricles decreases as the ventricles relax and start filling with blood.

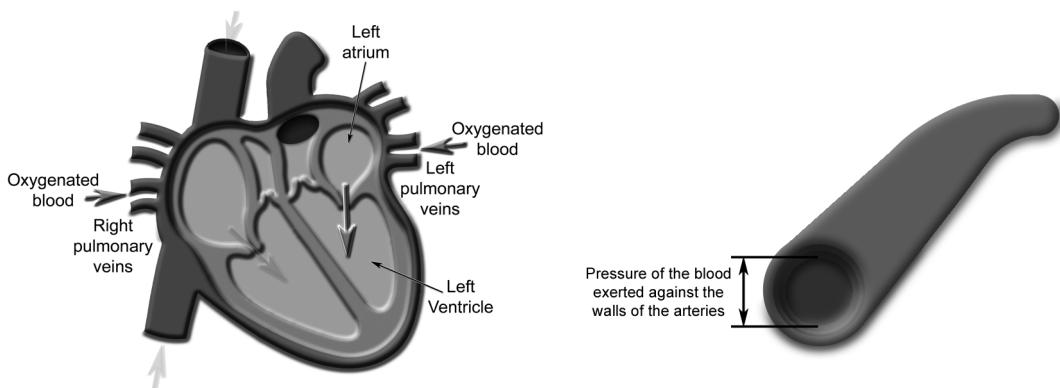


Figure 11-3 Heart during diastole

Systole marks the time of maximum arterial pressure while diastole marks the time of minimum arterial pressure. Determining an individual's systolic and diastolic arterial blood pressures is a standard clinical measurement and the values help to determine the functional integrity of the cardiovascular system.

Furthermore, a third important metric is called *Mean Arterial Pressure* (MAP) which is defined as the average arterial pressure over a cardiac cycle (heartbeat) and can be estimated from the systolic and diastolic pressures as:

$$\text{MAP} \approx \frac{\text{DIA} + (\text{SYS} - \text{DIA})}{3}$$

Fluctuations in the Mean Arterial Pressure (MAP) are due to many factors including the heart rate, blood volume, blood viscosity and the resistance to the flow of blood by the blood vessels. These factors are affected by age, diet, obesity, physical activity, alcohol, drugs, stress and diseases among others.

Some of the key players in the natural regulation of the arterial blood pressure in the body are shown in Figure 11-4.

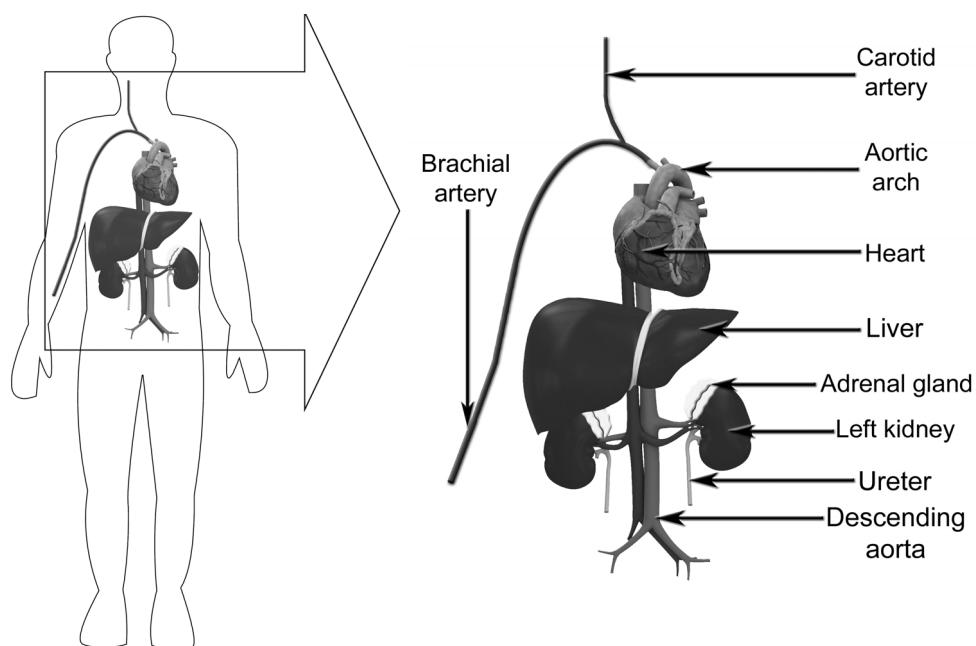


Figure 11-4 Key players in the regulation of the arterial blood pressure

The arterial blood pressure in the body is controlled by two mechanisms known as the baroreceptor reflex and the *Renin-Angiotensin-Aldosterone System* (RAAS) that work in parallel as briefly described in the next two sections.

11-2 THE BARORECEPTOR REFLEX

The baroreceptor reflex is a negative feedback control system that regulates short-term changes in the mean arterial pressure (MAP) by detecting them with a set of pressure sensors known as arterial *baroreceptors* located in the *aortic arch* and the *carotid sinuses* as shown in Figure 11-5.

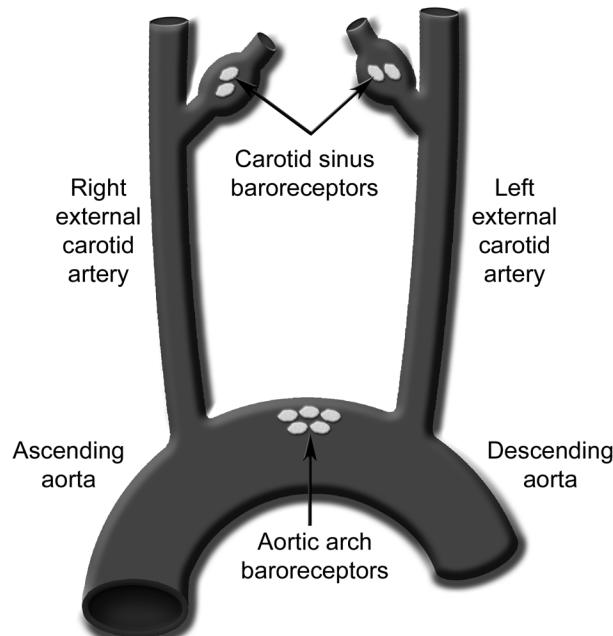


Figure 11-5 Baroreceptors at the aortic arch and carotid sinuses

These baroreceptors respond to the stretch caused by the force of the blood exerted against them. The baroreceptors generate an electrical signal with a frequency that is directly proportional to the stretch (blood pressure) and the signal activates and inhibits two of the main divisions of the *autonomic nervous system* known as *sympathetic* and *parasympathetic* nervous system.

If the blood pressure decreases, the frequency of the signal generated by the baroreceptors decreases proportionally and the signal activates the sympathetic nervous system and inhibits the parasympathetic nervous system. The activation of the sympathetic nervous system has the effect on the heart of increasing the heart rate and increasing the force of the ventricular contraction, and the effect on the arteries of narrowing their walls also known as *vasoconstriction*, which in turn, raises the blood pressure as illustrated in Figure 11-6.

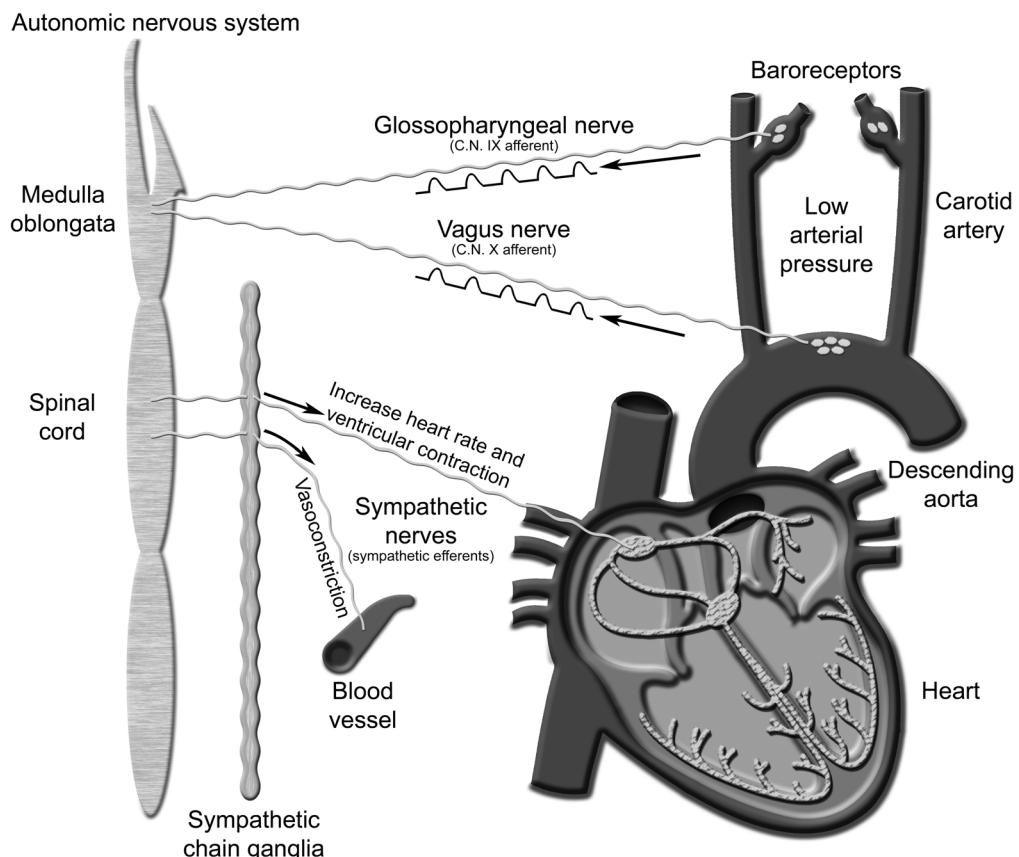


Figure 11-6 Sympathetic nervous system effect when the blood pressure decreases

In a similar manner, when the blood pressure increases, the frequency of the signal generated by the baroreceptors increases proportionally and the signal activates the parasympathetic nervous system and inhibits the sympathetic nervous system. The activation of the parasympathetic nervous system has the effect on the heart of decreasing the heart rate and decreasing the force of the ventricular contraction, which in turn, lowers the blood pressure as illustrated in Figure 11-7.

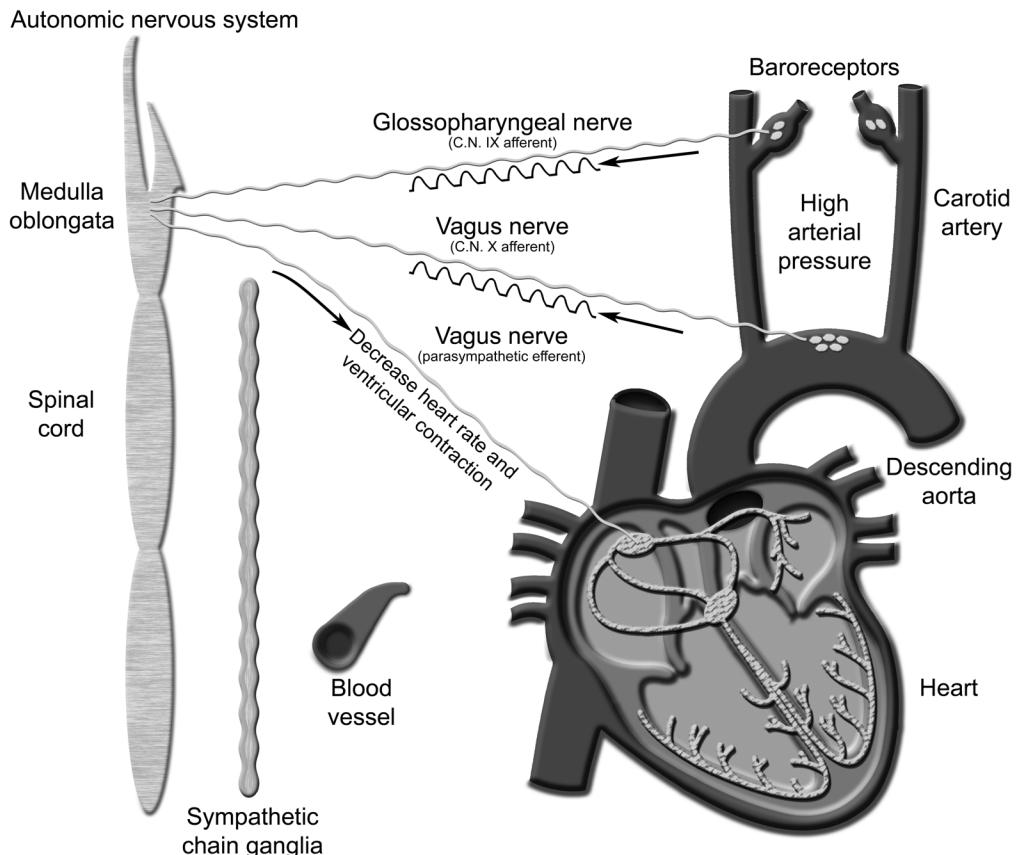


Figure 11-7 Parasympathetic nervous system effect when the blood pressure increases

11-3 RENIN-ANGIOTENSIN-ALDOSTERONE SYSTEM (RAAS)

The Renin-Angiotensin-Aldosterone System (RAAS) is a regulatory system that controls the levels of blood pressure and the levels of fluids. The system gets activated when a decrease in blood volume is detected in the kidneys by a special type of cells known as *juxtaglomerular cells*. The juxtaglomerular cells work in a similar way to the baroreceptors discussed in the last section. If the blood pressure decreases, there is a decrease in the stretch of the juxtaglomerular cells that is interpreted as a decrease in the volume of blood. The juxtaglomerular cells in the kidneys respond directly by releasing an enzyme known as *renin* into the bloodstream as shown in Figure 11-8.

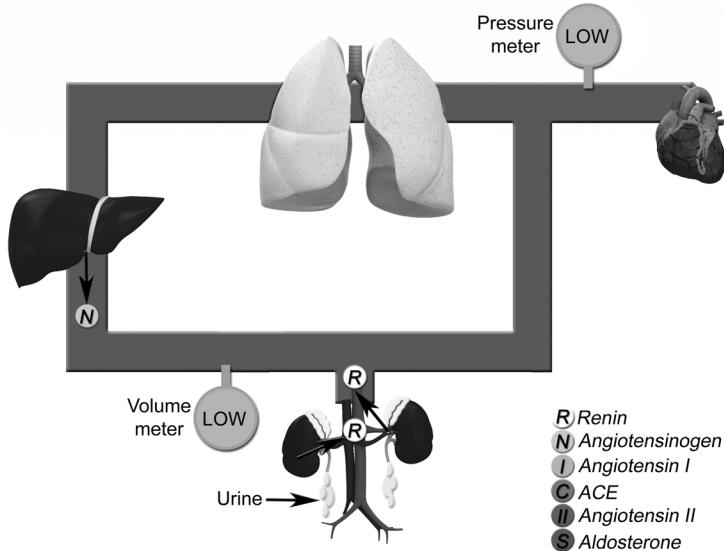


Figure 11-8 Response of the kidneys to the detection of low blood volume

Once the renin is flowing in the bloodstream, it encounters a chemical compound known as *angiotensinogen* which is produced and released mainly by the liver. Renin splits the *angiotensinogen* to form another chemical known as *angiotensin I* as shown in Figure 11-9.

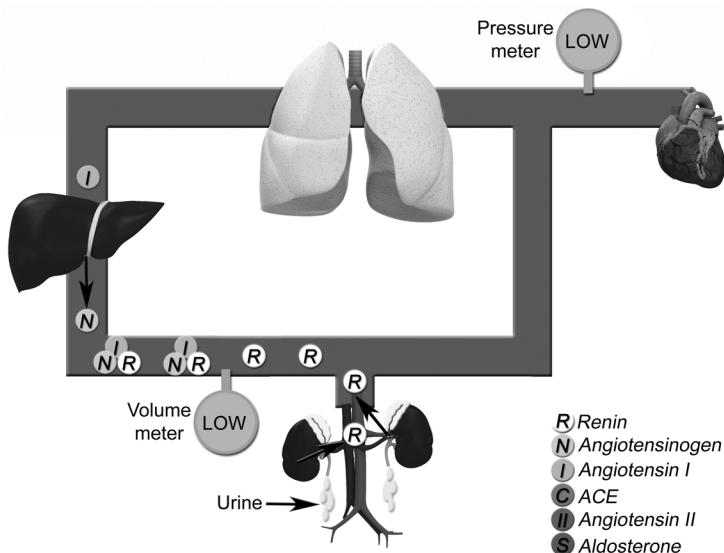


Figure 11-9 Creation of Angiotensin I

Angiotensin I has no biological effect other than reacting with another enzyme known as *Angiotensin-Converting Enzyme (ACE)* which is mainly secreted by a special type of cells in the lungs. When angiotensin I makes its way to the lungs, it encounters the Angiotensin-Converting Enzyme which catalyzes the conversion of Angiotensin I to *angiotensin II* as illustrated in Figure 11-10.

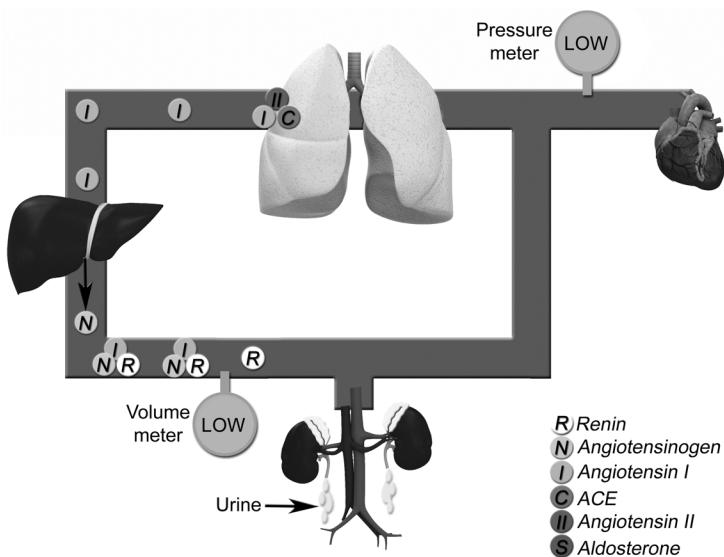


Figure 11-10 Conversion of Angiotensin I to Angiotensin II

Angiotensin II has three effects in response to the initial detection of low blood pressure. One of them is to reduce the size of blood vessels, therefore increasing the blood pressure as shown in Figure 11-11.

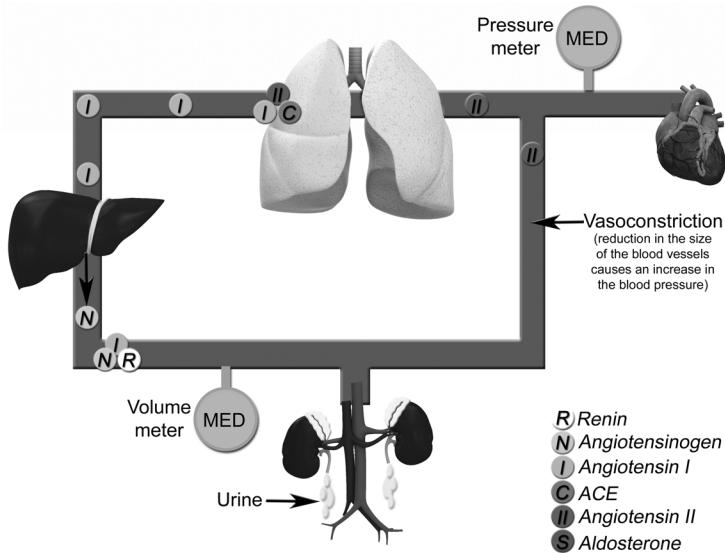


Figure 11-11 Vasoconstriction caused by Angiotensin II in an effort to increase blood pressure

The second effect of the Angiotensin II in response to the initial detection of low blood volume is to increase the sensation of thirst through the brain in an effort to increase the blood volume and therefore the blood pressure following the ingestion of liquids.

The third effect of the Angiotensin II in response to the initial detection of low blood pressure is to stimulate the release of a hormone known as *aldosterone* as explained in the next paragraph.

Once the Angiotensin II has been created in the lungs, it advances down to the *adrenal glands* which sit on top of the kidneys and stimulate the release of a hormone known as *aldosterone*. Aldosterone is a hormone that affects kidney function by increasing the reabsorption of sodium ions and water from the urine commonly known as fluid retention, and by releasing potassium and hydrogen ions. This increases blood volume, which in turn leads to an increase in blood pressure. Figure 11-12 illustrates the effect of Angiotensin II on the adrenal glands.

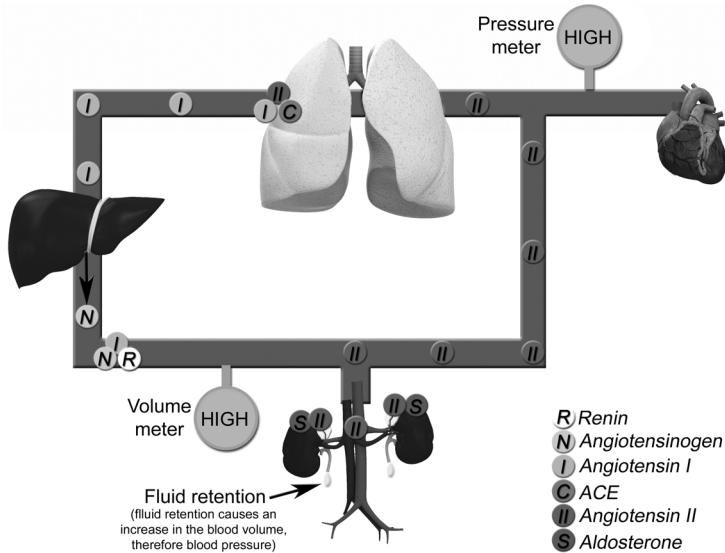


Figure 11-12 Fluid retention by the kidneys activated by the Aldosterone hormone

11-4 HYPERTENSION

High blood pressure or *hypertension* is a very common condition that may lead to health problems, like heart attack and stroke if uncontrolled.

Most individuals with hypertension have no signs or symptoms. A few may experience headaches, dizziness or nosebleeds but it is only when pressure has reached very dangerous levels and it is almost too late to start a treatment to control it.

Table 11-1 shows the classification of hypertension as defined by the American Heart Association for adults 18 years of age and older.

Classification	Systolic Pressure (mmHg)	Diastolic Pressure (mmHg)
Normal	90-119	60-79
Prehypertension	120-139	80-89
Stage I	140-159	90-99
Stage II	≥ 160	≥ 100
Isolated systolic hypertension	≥ 140	<90

Table 11-1 Hypertension classification

There are two types of hypertension depending on the root cause. *Primary hypertension* also known as essential hypertension accounts for 90% of all cases and it is diagnosed when there is no direct cause for the condition other than natural aging. The second type of hypertension is known as *secondary hypertension* and it is diagnosed among those patients whose high blood pressure is caused as a side effect of a different health condition like:

- Kidney disease.
- Adrenal glands tumors.
- Defects in blood vessels dating from birth.
- Certain medications such as birth control pills, cold medicines, antidepressants, prescription drugs and recreational drugs.

Major risk factors to develop hypertension according to the Mayo Clinic include:

- Chronic diseases: diabetes, high cholesterol and kidney disease among others.
- Age: the cardiovascular and renal system function declines as we age.
- Gender: more common among men, but incidence increases among women after menopause.
- Race: hypertension is common among blacks.
- Family history of hypertension.
- Overweight or obesity: larger size demands more oxygen for the tissues therefore more blood. An increase in the volume of blood causes an increase in the blood pressure as described in the last section.
- Diet: too much salt can cause fluid retention, therefore an increase in blood pressure also described in the last section.
- Lack of physical activity: a weak heart needs to work harder by increasing the heart rate, therefore the blood pressure.

-
- Alcohol.
 - Tobacco.
 - Stress.

According to the World Health Organization (WHO), hypertension is a disease with the largest incidence rates. Hypertension is mostly a result of the increased life expectancy and modern treatments for chronic degenerative diseases.

Not only people suffering from hypertension need to control their blood pressure levels, but also people suffering from other cardiovascular diseases like: aneurism, coronary artery disease, kidney disease, stroke and diabetes.

Furthermore, those people undergoing surgical procedures need to have a strict control of blood pressure in order to avoid *hypoxia* (lack of oxygen in the tissues), heart arrest or heart stroke due to the anesthesia, loss of blood and the mechanical manipulation of blood vessels during surgery.

11-5 INDIRECT MEASUREMENT OF ARTERIAL BLOOD PRESSURE

A number of different kinds of techniques are used by doctors to measure arterial blood pressure. Some of them are invasive and are designed to measure the arterial blood pressure by directly placing the sensor element in touch with the vascular system through the insertion of a catheter. Others try to make an attempt to measure arterial blood pressure noninvasively and are the type of technique described in this section.

The traditional manual technique used by doctors employs a *sphygmomanometer* and a *stethoscope*. The sphygmomanometer consists of an inflatable cuff for occlusion of the blood vessel, a rubber bulb for inflation of the cuff, a mechanical valve for deflation of the cuff and a mercury or mechanical *manometer* for the detection of pressure as shown in Figure 11-13.

Blood pressure is measured by using the stethoscope for the auditory detection of the systole and diastole phases in a technique known as the *oscillometric method* that will be described in the next paragraphs.

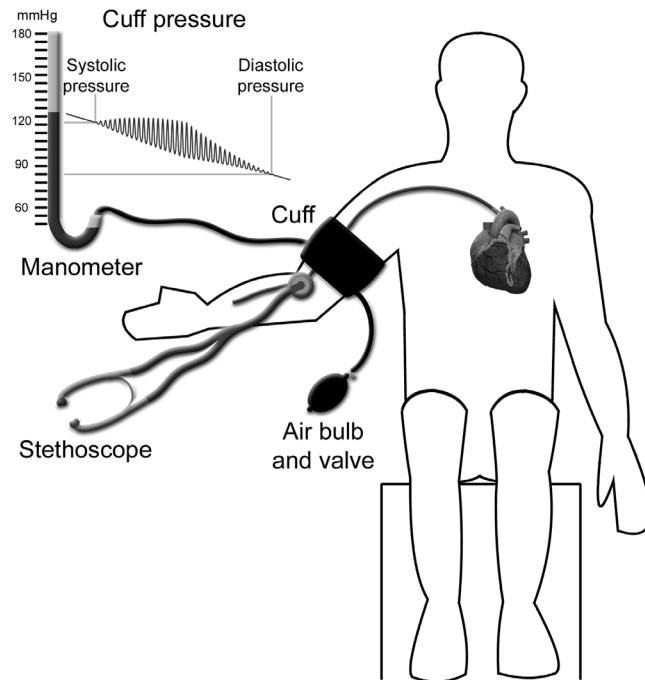


Figure 11-13 Manual sphygmomanometer

The oscillometric method is illustrated in Figure 11-13 and it is based on the assumption that the pressure applied by the blood to the artery wall is the same pressure inside the cuff. The cuff is placed at heart level wrapped around the upper arm and over the brachial artery while the stethoscope is placed in the joint of the arm where the brachial artery is also quite accessible and superficial.

The cuff is inflated until the pressure is above the expected systolic pressure (Table 11-1) and then the valve is opened in order to deflate the cuff at a slow regular rate of 2 mmHg/sec. As long as the systolic blood pressure is below the cuff pressure only silence is detected with the stethoscope because the cuff is preventing the passage of blood through the artery. As the cuff keeps deflating, the systolic blood pressure will eventually be higher than the cuff pressure. It is at that point when the blood is able to spurt under the cuff and cause an audible sound that can be detected through a stethoscope. The manometer reading at this point indicates the systolic pressure and as the pressure in the cuff keeps decreasing the audible sounds will keep pounding at the same rate of the heart. The audible sounds will be detected until the pressure cuff falls below the diastolic pressure and of course at that time the reading from the manometer indicates the diastolic pressure.

The blood pressure monitor presented in the next section is based on the same method except that the manometer is replaced with a solid state pressure sensor and a microcontroller not only to determine the systolic and diastolic pressures but also to control an air pump and solenoid valve to inflate and deflate the cuff automatically.

11-6 DESIGN OF A BLOOD PRESSURE MONITOR

Figure 11-14 shows the block diagram of the blood pressure monitor presented in this chapter. The design includes two boards from Freescale and a series of third party parts that connect through hoses as explained in the next pages..

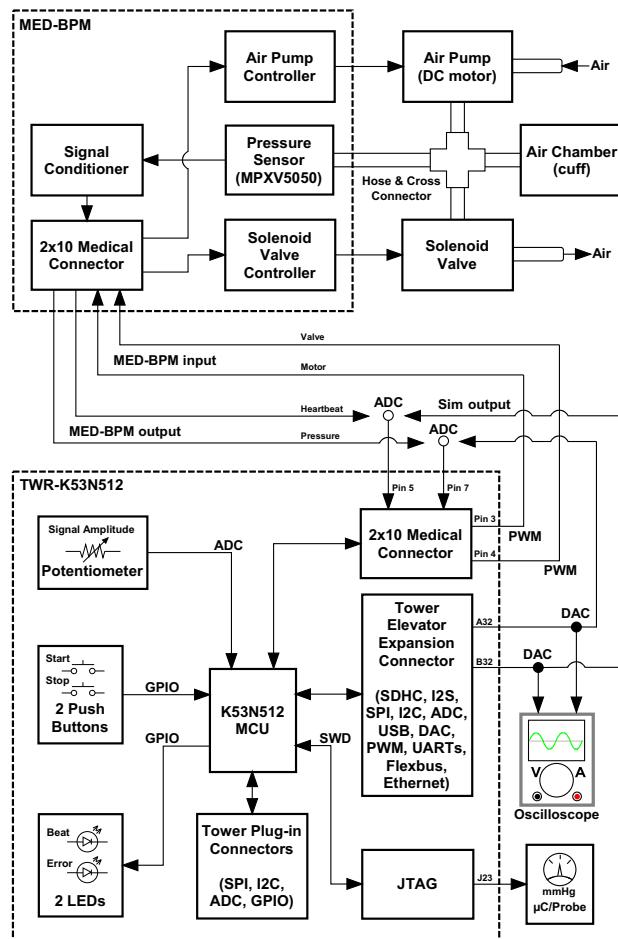


Figure 11-14 Block diagram of the blood pressure monitor

One of the key components is the pressure sensor manufactured by Freescale. The MPXx5050 series of piezoresistive transducers provide an accurate, high level analog output signal that is proportional to the applied pressure. The sensor is offered in different packages and the part used in this design is shown in Figure 11-15.

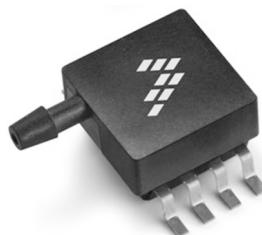


Figure 11-15 Pressure sensor from Freescale: Part # MPXV5050GP

The cuff can be purchased from any pharmacy or medical equipment store that carries replacement parts for commercial blood pressure monitors. The cuff used in this design was purchased from Walgreens pharmacy and it is shown in Figure 11-16.



Figure 11-16 Blood pressure cuff

The air pump is distributed by Cole-Parmer (www.coleparmer.com). The part used in this design provides 0.5 liter-per-minute, maximum pressure of 6.8 psi, it runs at 3 VDC and has a 3 mm barbed port. Figure 11-17 shows part # EW-79600-08:



Figure 11-17 Miniature OEM pressure pump by Cole-Parmer: Part # EW-79600-08

The solenoid valve is manufactured by Gems Sensors & Controls (www.gemssensors.com). The part used in this design runs at 3 VDC, has 3 mm barbed ports and it is part # MB202-VB30-L200 shown in Figure 11-18.



Figure 11-18 Solenoid valve by Gems Sensors & Controls: Part # MB202-VB30-L200

A flexible air hose or vacuum hose with an inside diameter of 3 mm and a 3 mm barbed cross connector from Cole-Parmer allows you to connect the cuff, pressure sensor, air pump and solenoid valve. Figure 11-19 shows part # WU-30705-06.



Figure 11-19 3 mm barbed cross connector from Cole-Parmer

Depending on the inside diameter of the hose or the port outside diameter of the parts you were able to find you may need a barbed reducing connector like the one shown in Figure 11-20, also from Cole-Parmer that connects a 3 mm hose to a $\frac{1}{4}$ " hose.



Figure 11-20 $\frac{1}{4}$ " to 3 mm barbed reducing connector from Cole-Parmer

The analog front end that contains the pressure sensor along with the circuits to filter and amplify the signal is offered by Freescale. The board also includes the circuits necessary to drive the air pump and solenoid valve. The part number is MED-BPM and it is shown in Figure 11-21 connected to the TWR-K53N512 controller through the 2x10 medical connector.

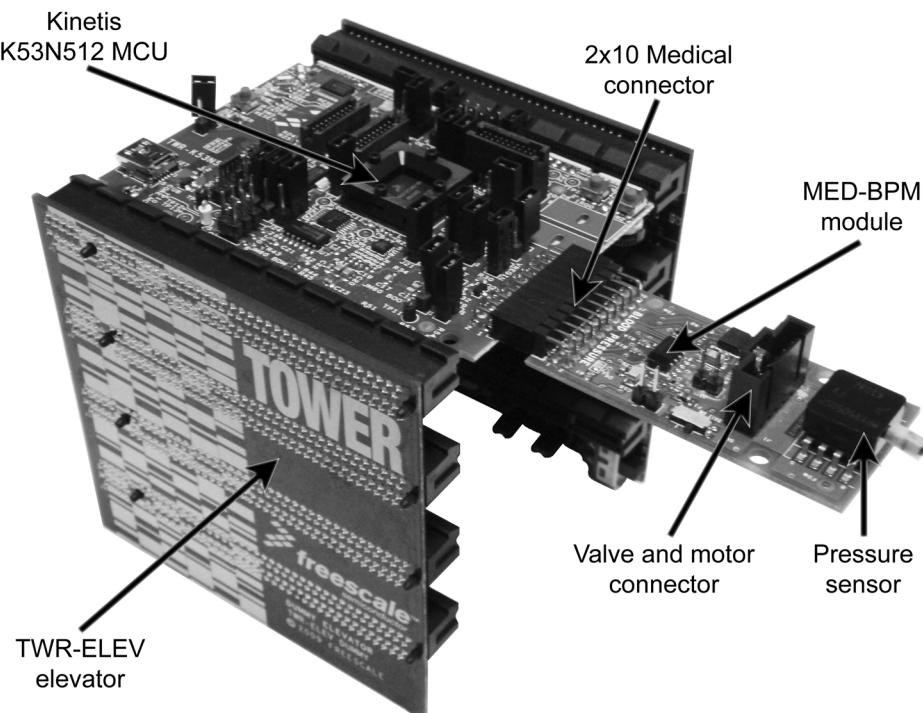


Figure 11-21 TWR-K53N512 and MED-BPM

The circuits to drive the air pump and solenoid valve in the MED-BPM module are identical and are shown in Figure 11-22 and Figure 11-23. They are typical MOSFET based inductive load drivers. The circuits in this design feature an optical isolator in order to allow the use of two different power sources that can be configured via the jumpers. The preferred power source used in this design is external power provided by a 3VDC 700mA AC-to-DC wall power adapter.

The PUMP_CTRL and VALVE_CTRL signals are outputs of a PWM module and the rate of inflation and deflation is controlled by adjusting the duty cycle of these two signals as we will describe in the last section of this chapter where we explain how the code works.

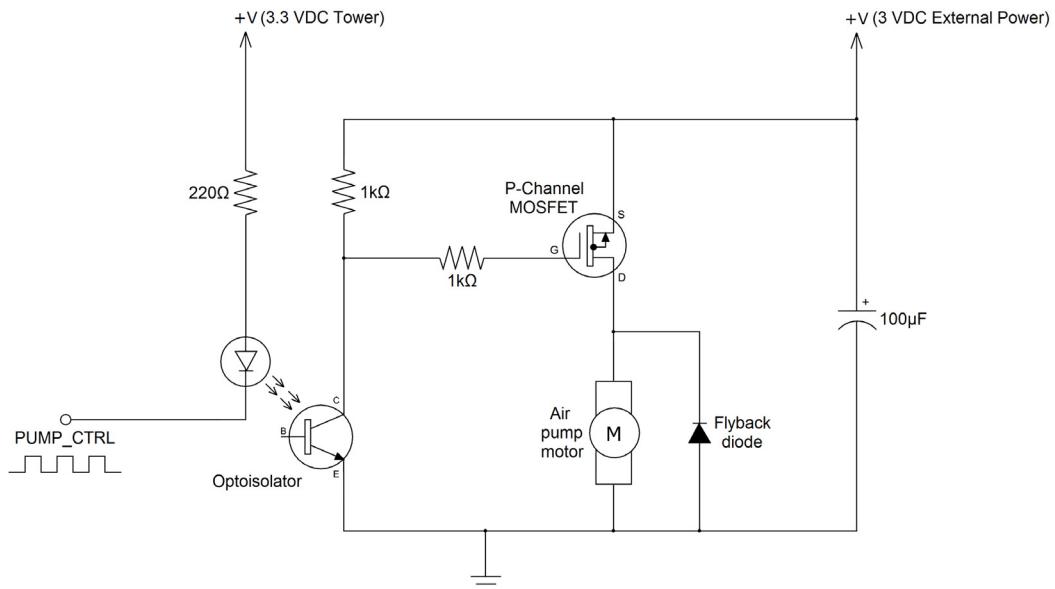


Figure 11-22 Air pump controller

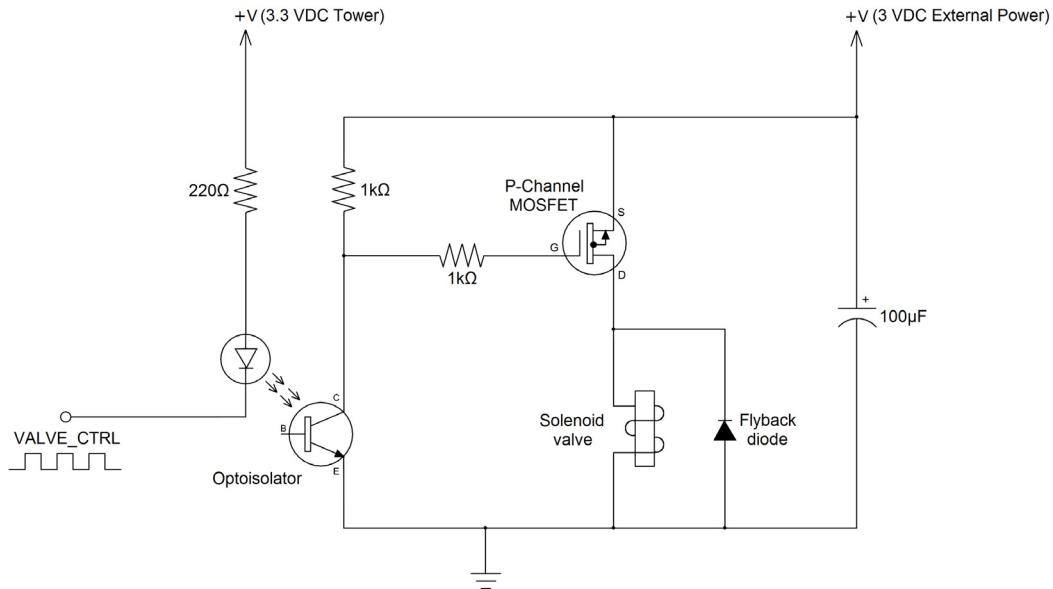


Figure 11-23 Solenoid valve controller

The signal conditioning circuit for the pressure sensor in the MED-BPM module is shown in Figure 11-24. It includes a series of passive low and high pass filters along with operational amplifiers to not only amplify the signals but also to avoid any loading effects common to passive filters.

The circuit has two output signals; the CUFF_PRESSURE and the HEARTBEAT signal. The HEARTBEAT signal is used to determine the time of the systolic and diastolic phases and the CUFF_PRESSURE signal indicates the actual arterial blood pressure at such phases. In other words, the HEARTBEAT signal replaces the stethoscope used in the oscillometric method discussed in section 11-5 “Indirect Measurement of Arterial Blood Pressure” on page 224.

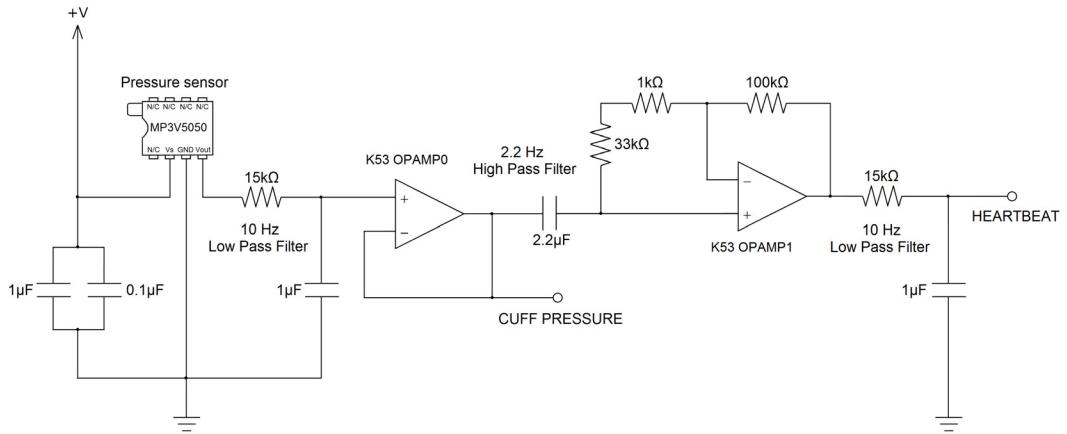
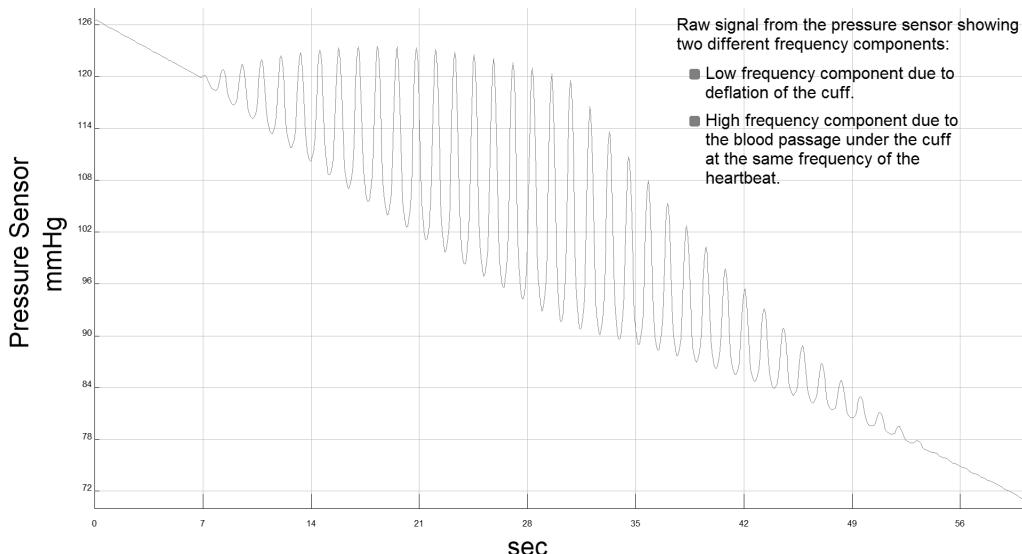


Figure 11-24 Pressure sensor signal conditioning

The following Figure 11-25 is provided as a supplement to the discussion of the oscillometric method in section 11-5 “Indirect Measurement of Arterial Blood Pressure” on page 224 and also to illustrate the purpose of all the different filters in the signal conditioning circuit.

The figure illustrates the raw signal at the pressure sensor output, the CUFF_PRESSURE signal at the output of the first low pass filter and the HEARTBEAT signal at the output of the last band pass filter.



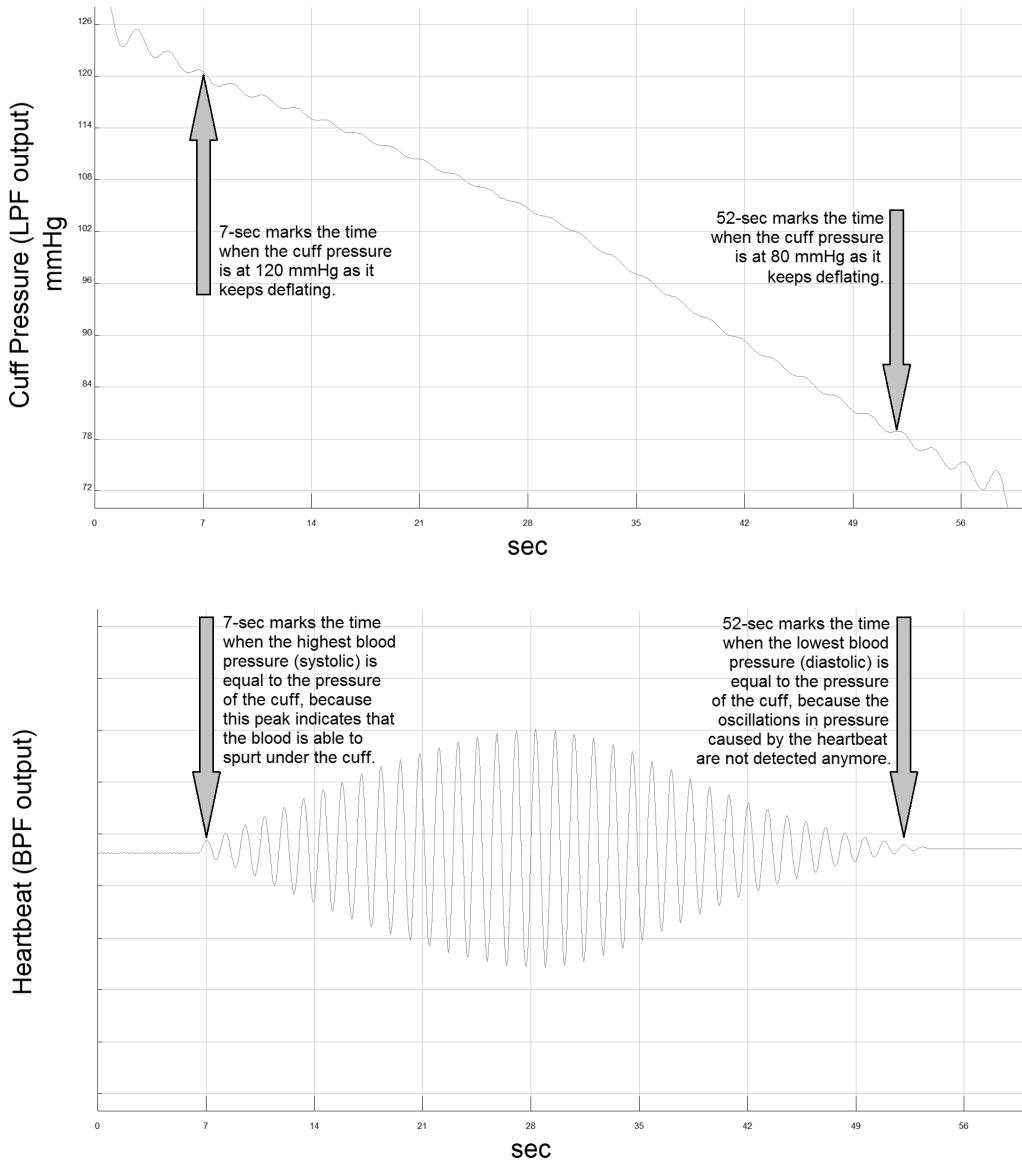


Figure 11-25 Pressure sensor signal conditioning signals at different stages

The 2x10 medical connector of the MED-BPM not only provides connectivity with the TWR-K53N512 board to make the heartbeat and cuff pressure signals available to the ADC modules but also allows the use of the on-chip op-amps and transimpedance amplifiers of the Kinetis 32-bit microcontroller to implement the required signal conditioning. The rest of

the signals have been omitted from the block diagram for the sake of simplicity but Table 11-2 shows the signal present in each pin of the medical connector.

MED-BPM Signal	Pin		MED-BPM Signal
VDD	1	2	GND
PUMP_CTRL	3	4	VALVE_CTRL
HEARTBEAT	5	6	Not connected
CUFF_PRESSURE	7	8	Not connected
OPAMP0 VOUT0	9	10	OPAMP1 VOUT1
OPAMP0 INN0-	11	12	OPAMP1 INN1-
OPAMP0 INP0+	13	14	OPAMP1 INP1+
Not connected	15	16	Not connected
Not connected	17	18	Not connected
Not connected	19	20	Not connected

Table 11-2 **Medical Connector 2x10 Pin Header Connections**

Notice from the block diagram in Figure 11-14 that in the absence of the MED-BPM module you can still run the application by using the simulated heartbeat and pressure signals coming out of the DAC modules. You just need to connect a jumper wire between pin 5 of the medical connector and pin B32 of the elevator module in the tower system for the heartbeat signal and a jumper wire between pin 7 of the medical connector and pin A32 of the elevator module in the tower system for the cuff pressure signal. The potentiometer is used in such situation to increase or decrease the simulated blood pressure.

The communication with μC/Probe which will be used to display the results of the blood pressure monitor is via the J-Link debug probe from Segger through the JTAG connector labeled as J23.

All the rest of third party parts including the air pump motor, solenoid valve and hoses need to be put together as shown in the block diagram in Figure 11-14. We put the parts together using a perfboard and a transparent lid case as shown in Figure 11-26:

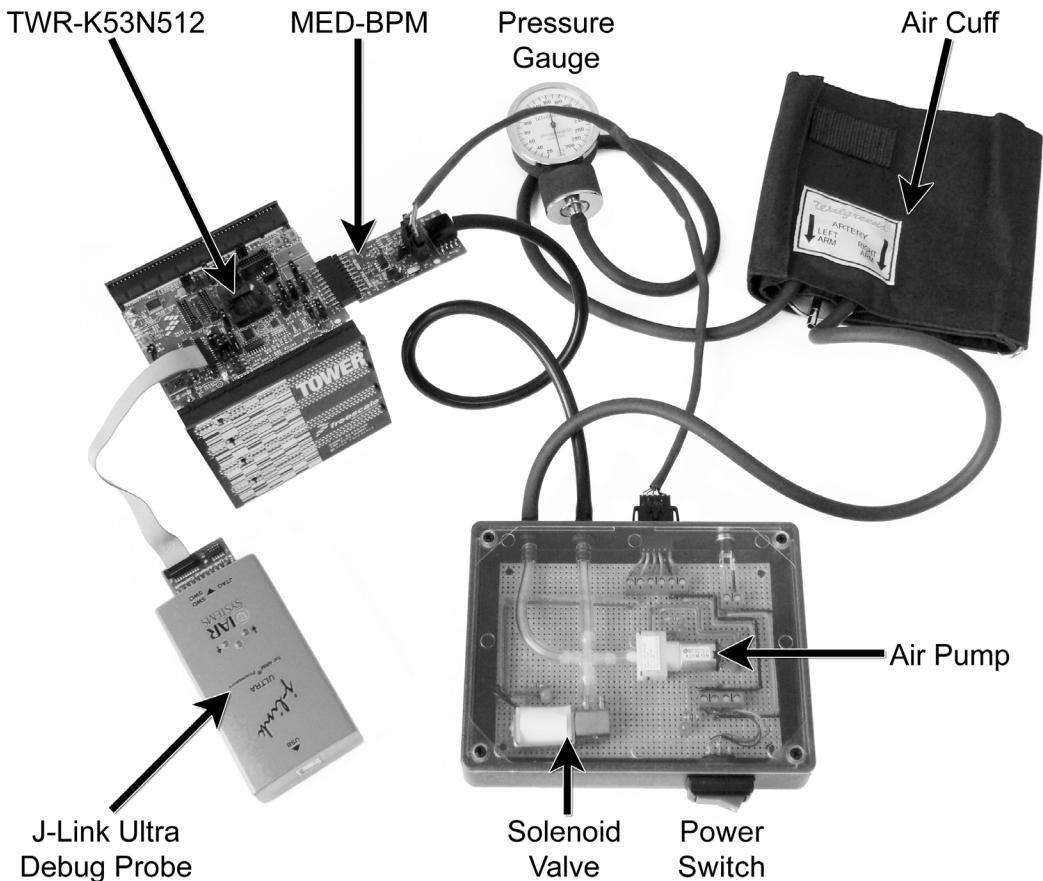


Figure 11-26 Blood pressure monitor setup

Connecting the solenoid valve and air pump is pretty straight forward; all you really need to pay special attention is to the 1x6 connector's pin-out that connects the MED-BPM in J1 header and your own setup (3VDC power adapter, air pump motor and solenoid valve).

The MED-BPM comes with a male header in J1 that mates with a female socket from Molex (part number 50-57-9406) shown in Figure 11-27. You need to purchase this connector separately along with the contact crimps (Molex part number 16-02-1124) also shown in Figure 11-27.

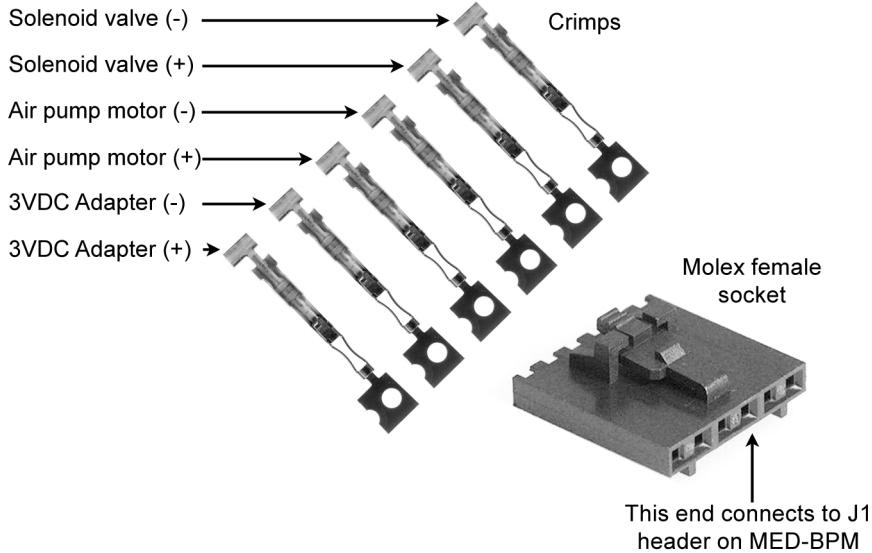


Figure 11-27 Female socket from Molex part # 50-57-9406

Table 11-3 shows the pinout of the J1 header on the MED-BPM:

MED-BPM J1 Header Pin	MED-BPM Signal
1	3VDC (+)
2	3VDC (-)
3	Air pump motor (+)
4	Air pump motor (-)
5	Solenoid valve (+)
6	Solenoid valve (-)

Table 11-3 Pin-out of the 1x6 MED-BPM connector in J1

Notice from Figure 11-26 that we included a mechanical pressure gauge in the system which we found very practical in order to validate the results you get on the µC/Probe screen.

11-7 RUNNING THE EXAMPLE PROJECT

This section describes the steps you need to follow to run the blood pressure monitor example. Start by connecting the MED-BPM module to the TWR-K53N512 controller through the 2x10 medical connector as shown in Figure 11-21.

Connect the air pump, solenoid valve, cuff and pressure sensor as shown in Figure 11-26.

Assuming you have followed all the setup steps described in Chapter 7, “Setup” on page 101, proceed to connect one end of the Segger's J-Link for ARM debug probe to the JTAG connector on J23 of the TWR-K53N512 and the other end of the probe to any USB port available in your PC.

Run IAR's Embedded Workbench for ARM (EWARM) and open the workspace at:

```
$\Micrium\Examples\Freescale\TWR-K53N512\OS2-TCPIP-BPM\IAR\  
OS2-TCPIP-BPM.eww
```

The workspace window shows all of the files in the project which are sorted into groups represented by folder icons. Figure 11-28 shows the project files for OS2-TCPIP-BPM in the workspace explorer window.

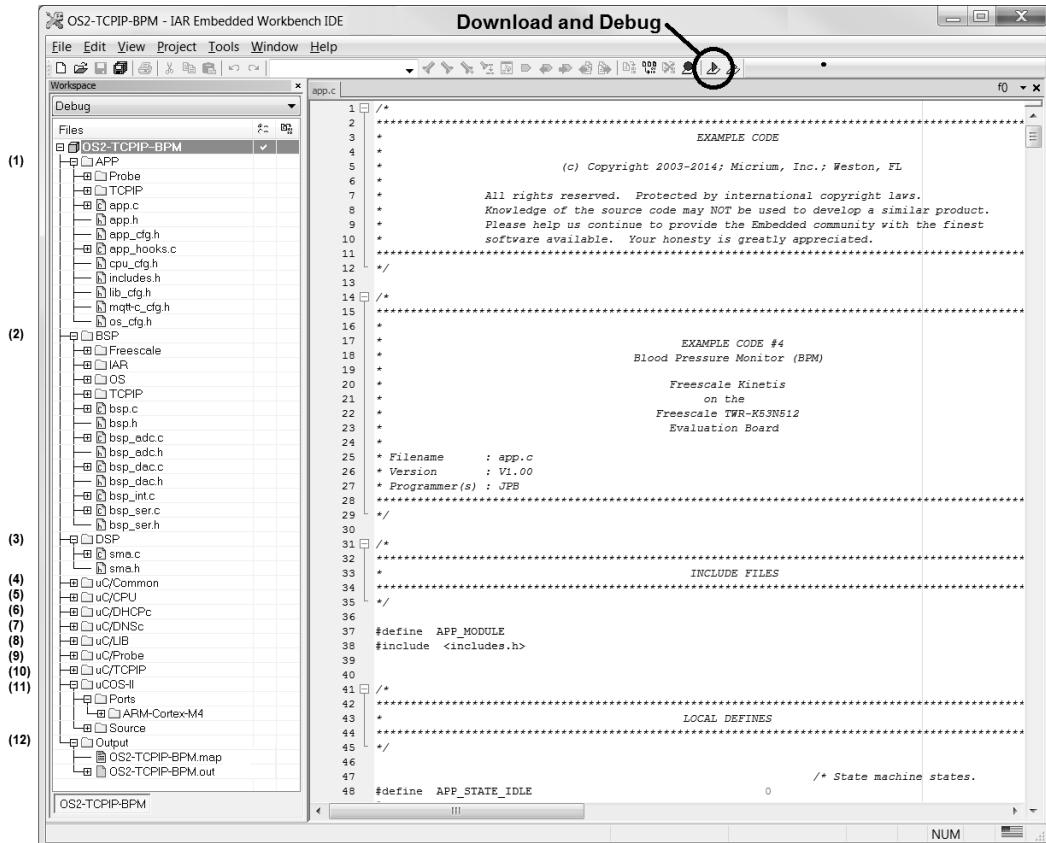


Figure 11-28 Running the project with EWARM

- F11-28(1) The **APP** group includes all of the application files for this example, including the header files used to configure the application.
- F11-28(2) The **BSP** group contains the files that comprise the board support package. The board support package includes the code used to control the peripherals on the board. For this example, the software to control the LEDs, pushbuttons, ADCs, DACs, operational amplifiers and transimpedance amplifiers will be used. The subgroup **Freescale** includes the header files for the Kinetis K50 family of microcontrollers.
- F11-28(3) The **DSP** group contains the files that define some of the Digital Signal Processing functions called by the application.

- F11-28(4) The **uC/Common** group contains the kernel abstraction layer.
- F11-28(5) The **uC/CPU** group contains the uC/CPU source code files.
- F11-28(6) The **uC/DHCPC** group contains the header files for the DHCP client that allows you to negotiate an IP address with your network's router.
- F11-28(7) The **uC/DNSc** group contains the header files for the DNS client that allows you to translate domain names into IP addresses.
- F11-28(8) The **uC/LIB** group contains the uC/LIB source code files.
- F11-28(9) The **uC/Probe** group contains the source code to enable communication with uC/Probe via TCP/IP.
- F11-28(10) The **uC/TCP-IP** group contains the basic header files for the TCP/IP stack.
- F11-28(11) The **uC/OS-II** group contains the uCOS-II source code.
- F11-28(12) The **Output** group contains the files generated by the compiler/linker.

Start the debugger by clicking the “Download and Debug” icon as shown in Figure 11-28. The application is programmed to the internal Flash of the K53N512 microcontroller and the debugger starts. The code automatically starts executing and stops at the `main()` function in the `app.c` file.

Click the “Go” icon in IAR Embedded Workbench’s debugging tools to run the application as shown in Figure 11-29.

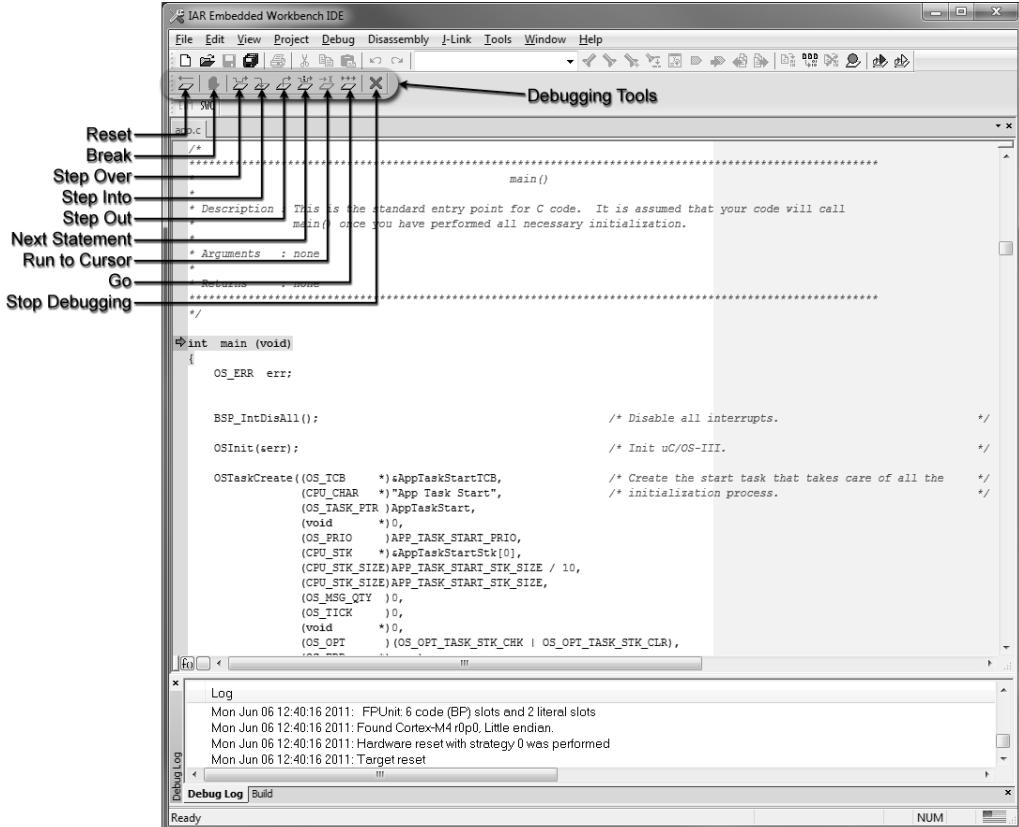


Figure 11-29 Downloading the Application and Starting the Debugger

You can test the application by starting μC/Probe and open the workspace for the Blood Pressure Monitor at:

`$\Micrium\Examples\Freescale\TWR-K53N512\OS2-TCPIP-BPM\IAR\OS2-TCPIP-BPM.wspx`

After μC/Probe starts, click the run button (green triangle). Once μC/Probe is running, you will see a screen similar to the one shown in Figure 11-30

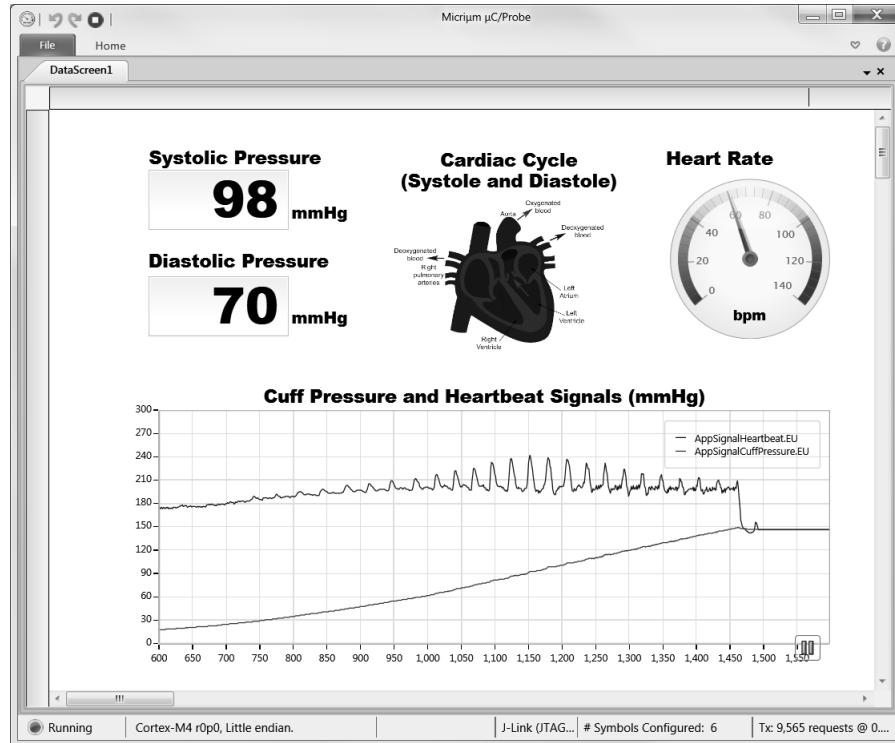


Figure 11-30 µC/Probe Blood Pressure Monitor Dashboard

It is important for the pressure sensor to be at the same height of your heart, take a seat, wrap your left arm around the air cuff and press the pushbutton on the TWR-K53N512 board labeled as SW1 to start the application.

The application will turn on the air pump motor, close the solenoid valve and the air cuff will start to inflate. You can press the pushbutton on the TWR-K53N512 board labeled as SW2 at anytime to stop the application (stopping the application will stop inflating and will open the solenoid valve in case of panic).

The application will inflate the cuff slowly up to a default setting of 150 mmHg and you can actually watch the pressure and heartbeat signals in real time on the dashboard. The test takes a total of 30-seconds during which you should keep your arm steady to avoid any noise in the signals. At the end of the test the solenoid valve is opened to allow deflation and your diastolic, systolic and mean arterial pressures will be displayed in the three vertical meters on the dashboard. Your heart rate will also be displayed in the gauge.

11-8 HOW THE CODE WORKS

The blood pressure monitor application consists of three different tasks:

- User IF task: The user interface task monitors the state of the two pushbuttons onboard the TWR-K53N512. The pushbuttons are used to either start or stop the application. The task is defined by `AppTaskUserIF()` in `app.c`.
- Sim task: The simulator task monitors the state of the potentiometer and updates the amplitude of a cuff pressure simulated signal generated by the DAC according to the position of the potentiometer. The task is defined by `AppTaskSim()` in `app.c`.
- DAQ task: The data acquisition task is the main task in the application and implements the state machine that processes the analog input samples to calculate the blood pressure and the heart rate. The task is defined by `AppTaskDAQ()` in `app.c`.

The interaction among the different tasks is facilitated by the use of µC/OS-II's semaphores and message queues and it is illustrated in Figure 11-31.

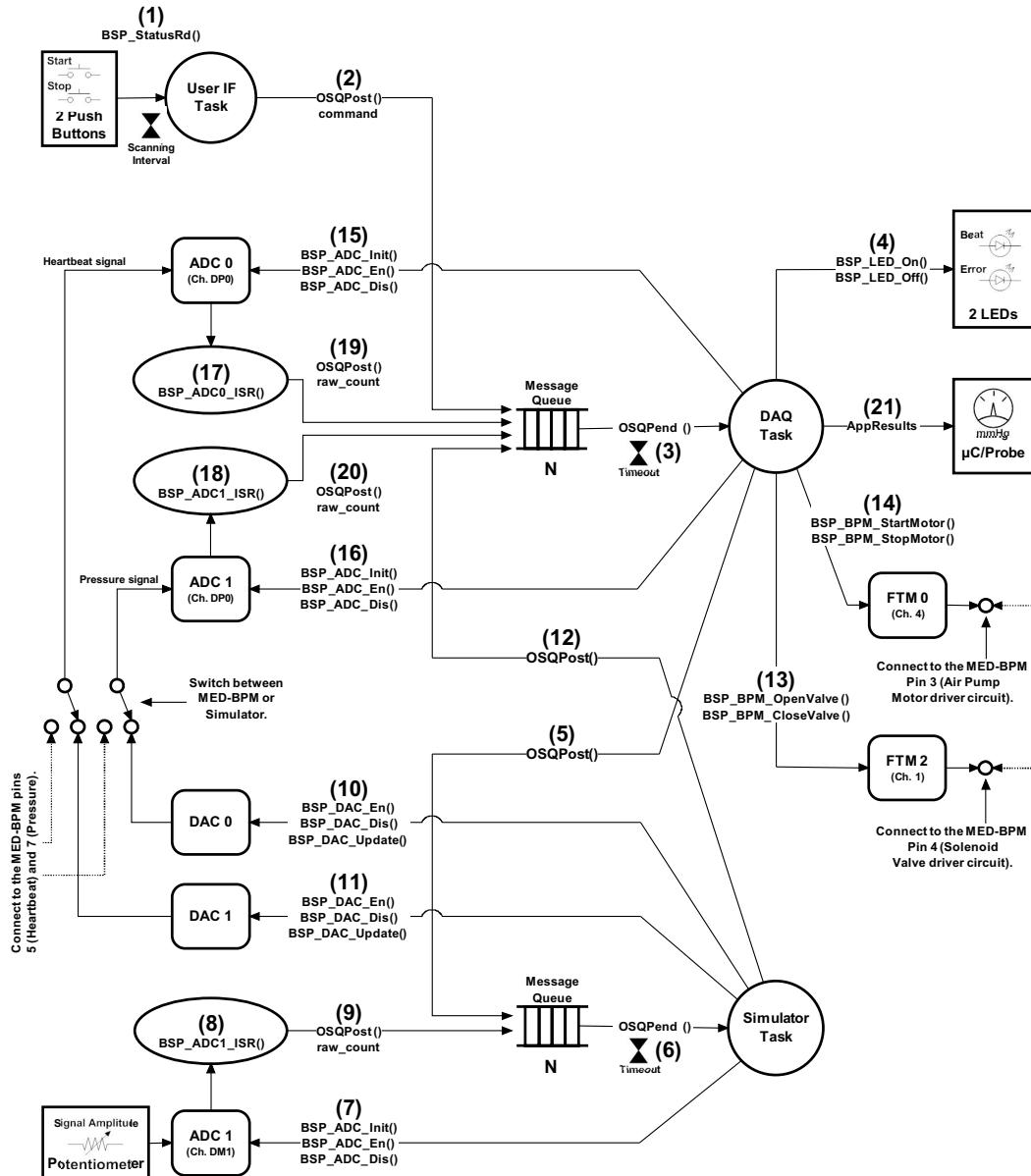


Figure 11-31 Interaction of Application Tasks

F11-31(1) The state of the push buttons is monitored by the user IF task at a scanning interval defined by `BSP_STATUS_CHECK_INTERVAL`. See `AppTaskUserIF()` in `app.c` and `BSP_StatusRd()` in `bsp.c`.

-
- F11-31(2) Depending on the pushbutton pressed by the user a message from the user IF task to either start or stop the application is posted to the DAQ task's built-in message queue by calling `OSTaskQPost()`. See `AppTaskUserIF()` in `app.c`.
 - F11-31(3) The DAQ task can receive data messages directly from the ADCs ISRs or command messages from the other two tasks. When `OSTaskQPend()` is called, the message is retrieved and processed in two different ways depending on the sender and the type of message. If the message comes from one of the ADC's ISR then the message is processed by calling the function `AppTaskDAQ_ProcessData()` and if the message comes from any other task then the message is processed by calling the function `AppTaskDAQ_ProcessCmd()` in `app.c`.
 - F11-31(4) If the message retrieved by the DAQ task is a command to start the blood pressure measurement then the DAQ task turns off the LEDs by calling the function `BSP_LED_Off()` in `bsp.c`.
 - F11-31(5) If the message retrieved by the DAQ task is a command to start the blood pressure measurement in simulation mode then the DAQ task posts a message to the sim task's built-in message queue to start the simulator. See `AppTaskDAQ()` in `app.c` and `AppTaskDAQ_ProcessCmd()` in `app.c`.
 - F11-31(6) In similar fashion, the sim task can receive messages directly from the ADC1's ISR or from the DAQ task. When `OSTaskQPend()` is called, the message is retrieved and processed in different ways depending on the sender. The sim task retrieves the received message from its message queue and proceeds according to the command issued by the DAQ task as follows:
 - F11-31(7) If the message contains a start command then the sim task starts ADC1 channel DM1 in order to read the potentiometer. See `AppTaskSim()` in `app.c`.
 - F11-31(8) The CPU is interrupted by the completion of a conversion of ADC1 Channel DM1 and the interrupt is handled by `BSP_ADC1_ISR()` in `bsp_adc.c`.
 - F11-31(9) The ADC1 ISR posts the result of the conversion in the form of a 16-bit raw count to the sim task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC1_ISR()` in `bsp_adc.c`.

-
- F11-31(10) The sim task retrieves the message from the queue and configures the DAC0 to output a simulated cuff pressure waveform at an amplitude proportional to the raw count of the ADC1 channel DM1. See `AppTaskSim()` in `app.c`.
 - F11-31(11) The sim task retrieves the message from the queue and configures the DAC1 to output a simulated heartbeat waveform. See `AppTaskSim()` in `app.c`.
 - F11-31(12) The sim task posts a message to the DAQ task's built-in message queue notifying the DAQ task that the simulator is running. See `AppTaskSim()` in `app.c`.
 - F11-31(13) The DAQ task retrieves the message from the queue and initializes FlexTimer FTM2 to output a PWM signal to drive the solenoid valve. See `AppTaskDAQ()` and `AppTaskDAQ_ProcessCmd()` in `app.c`.
 - F11-31(14) The DAQ task retrieves the message from the queue and initializes FlexTimer FTM0 to output a PWM signal to drive the air pump motor. See `AppTaskDAQ()` and `AppTaskDAQ_ProcessCmd()` in `app.c`.
 - F11-31(15) The DAQ task retrieves the message from the queue and enables ADC0 channel DP0 to convert the heartbeat signal. See `AppTaskDAQ()`, `AppTaskDAQ_ProcessCmd()` and `AppTaskDAQ_ExecCmd()` in `app.c`.
 - F11-31(16) The DAQ task retrieves the message from the queue and enables ADC1 channel DP0 to convert the cuff pressure signal. See `AppTaskDAQ()`, `AppTaskDAQ_ProcessCmd()` and `AppTaskDAQ_ExecCmd()` in `app.c`.
 - F11-31(17) The CPU is interrupted by the completion of a conversion of ADC0 channel DP0 and the interrupt is handled by `BSP_ADC0_ISR()` in `bsp_adc.c`.
 - F11-31(18) About the same time, the CPU is interrupted by the completion of a conversion of ADC1 channel DP0 and the interrupt is handled by `BSP_ADC1_ISR` in `bsp_adc.c`.
 - F11-31(19) The ADC0 ISR posts the result of the conversion in the form of a 16-bit raw count to the DAQ task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC0_ISR()` in `bsp_adc.c`.

- F11-31(20) The ADC1 ISR posts the result of the conversion in the form of a 16-bit raw count to the DAQ task's built-in message queue by calling the function `OSTaskQPost()`. See `BSP_ADC1_ISR()` in `bsp_adc.c`.
- F11-31(21) The DAQ task retrieves the message from the queue and uses the raw counts from ADC0 and ADC1 to calculate the heart rate, the blood pressure and display the values in μ C/Probe by calling the function `AppTaskDAQ_ProcessData()` and `AppCalcResults()` in `app.c`.

11-8-1 BIOMEDICAL SIGNAL ANALYSIS

The algorithm to calculate the arterial blood pressure is based on the oscillometric method described in Figure 11-13 and Figure 11-25 with just one modification: it doesn't look at the ramp-down but at the ramp-up. The reason why is because as opposed to the method that uses a manual air bulb, this digital blood pressure monitor uses an air pump motor, which means that the inflation rate can be controlled in a smooth manner. Given this advantage over the manual air bulb where the inflation rate is controlled by manually squeezing the bulb, we can spare the patient the slight pain caused by the cuff pressure during an unnecessary extended test that includes a manual inflation and a slow deflation. A digital blood pressure monitor that uses an air pump motor can instead inflate slowly while running the calculations in parallel and then deflate as fast as possible. The ramp-up algorithm is illustrated in Figure 11-32 and similar to the pulse detection algorithm for the pulse oximeter described in Chapter 10, "Pulse Oximeter" on page 179 it keeps two sliding windows of five points. One for the summit and one for the valley. A pulse is detected when a consecutive summit and valley are detected within the thresholds defined in volts by:

```
#define APP_MIN_HEARTBEAT_PK2PK_AMP_VOLTS    0.150
#define APP_MAX_HEARTBEAT_PK2PK_AMP_VOLTS    0.750
```

Listing 11-1 Heartbeat's Peak-to-Peak Amplitude Detection Thresholds

The thresholds are compared against the value of the peak to peak amplitude of each potential pulse and if it meets the requirements then the time-domain features of the pulse are stored into an array of pulses.

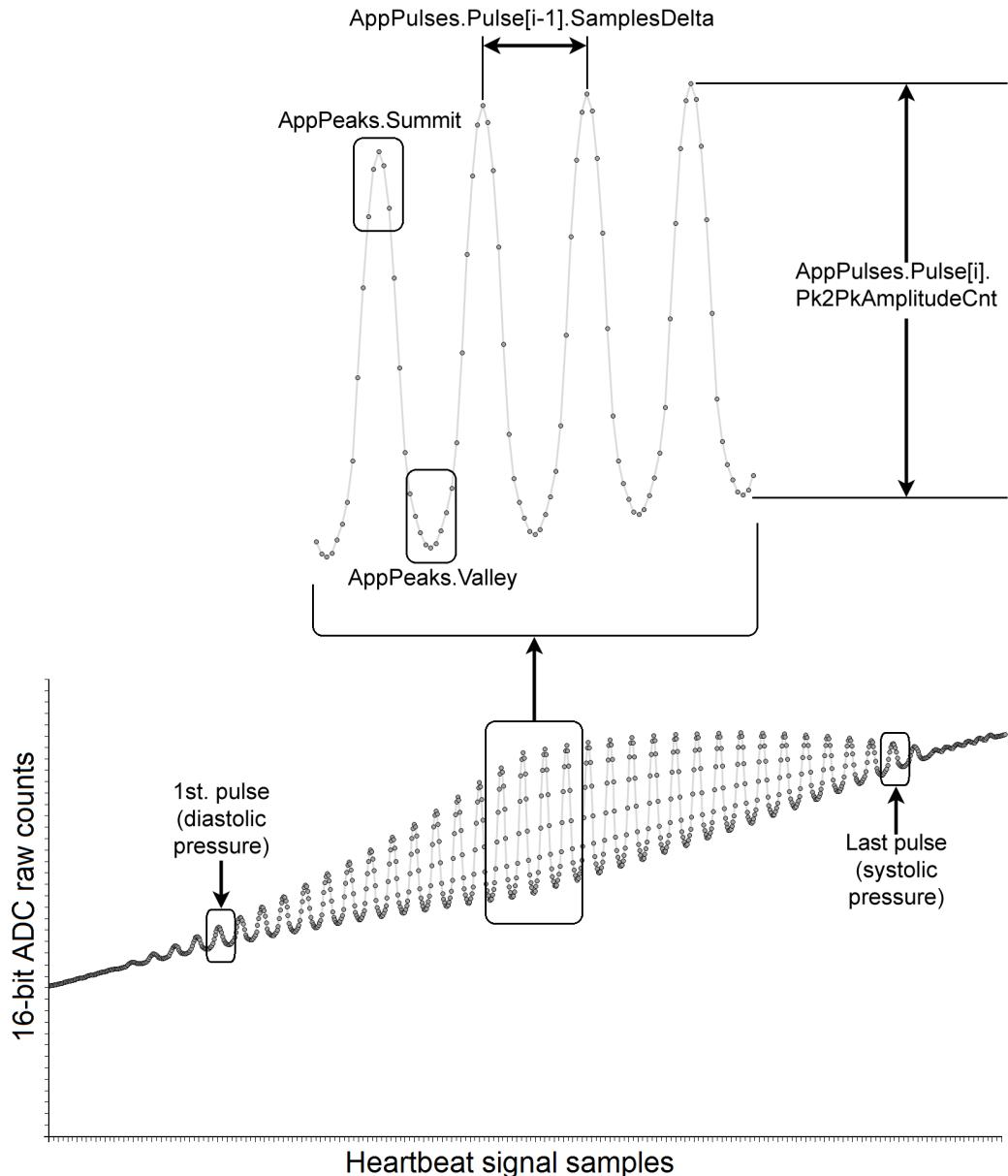


Figure 11-32 Blood Pressures Detection Algorithm

Similar thresholds are used to detect the diastolic and systolic pressures respectively:

```
#define APP_MIN_HEARTBEAT_DIA_PK2PK_AMP_VOLTS 0.125
#define APP_MAX_HEARTBEAT_DIA_PK2PK_AMP_VOLTS 0.750
#define APP_MIN_HEARTBEAT_SYS_PK2PK_AMP_VOLTS 0.175
#define APP_MAX_HEARTBEAT_SYS_PK2PK_AMP_VOLTS 0.750
```

Listing 11-2 Heartbeat's Peak-to-Peak Amplitude Diastolic and Systolic Detection Thresholds

The diastolic pressure is the one when the first pulse is detected during the ramp-up and the systolic pressure is the one when the last pulse is detected.

The mean arterial pressure is the one when the pulse reaches the maximum peak to peak amplitude.

The transfer function for the pressure sensor is linear at a temperature range between 0° F and 185° F and assuming a voltage reference of 3.3V the pressure in mmHg in function of the output voltage of the pressure sensor is given by $P = (V / 0.0079193268) - 16.66808$ and is shown in Figure 11-33:

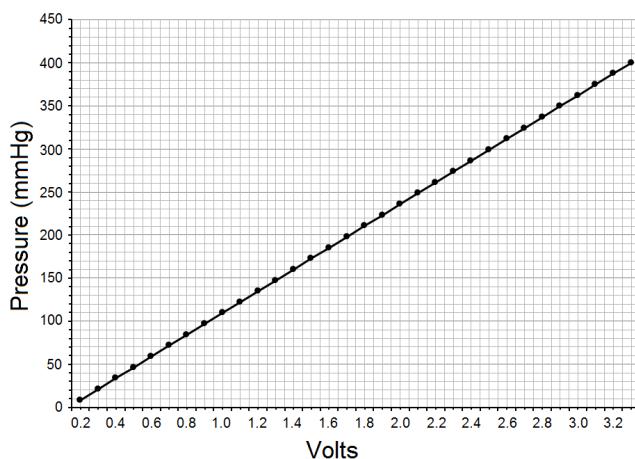


Figure 11-33 Pressure Sensor Transfer Function

In the next section you will learn that the algorithm has been fully implemented to support both ramp-up and ramp-down methods. Enabling both methods will slightly increase the accuracy of the results at the expense of the patient's comfort.

11-8-2 DATA PROCESSING STATE MACHINE

The state machine that processes every sample in `AppTaskDAQ_ProcessData()` is illustrated in Figure 11-34:

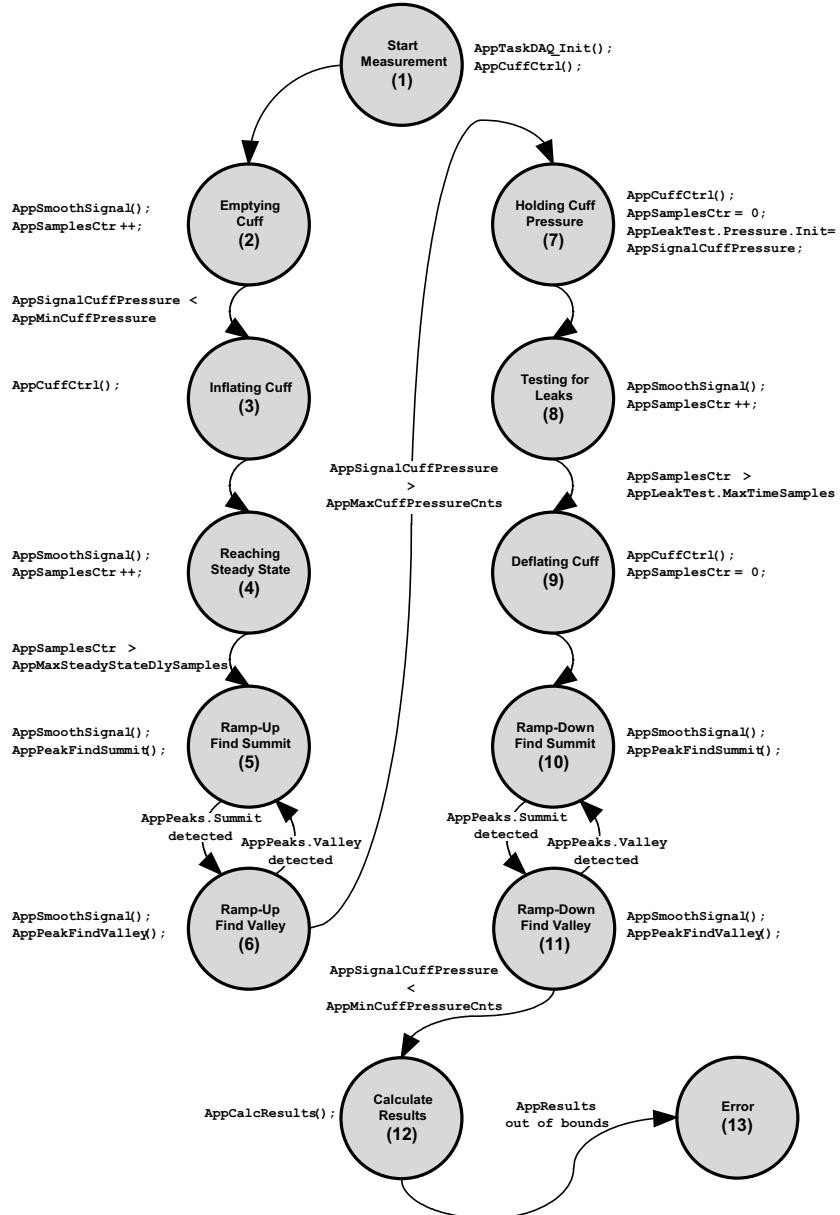


Figure 11-34 Data Processing State Machine

- F11-34(1) Once the start button is pressed, the application starts the blood pressure measurement by calling `AppTaskDAQ_Init()` to initialize some global variables and by calling `AppCuffCtrl()` to stop the air pump motor and open the solenoid valve which in turn will empty the cuff as fast as possible.
- F11-34(2) In this state, the cuff pressure signal is smoothed by applying a simple moving average filter and the result is stored in `AppSignalCuffPressure.Result`. This result is monitored until the cuff pressure is below a minimum threshold defined in mmHg by `APP_MIN_CUFF_PRESSURE_MMHG`.
- F11-34(3) Once the cuff is empty, it can be wrapped around the patient's arm and in this state, the cuff starts to be slowly inflated by calling `AppCuffCtrl()` with the correct arguments to start the air pump motor and close the solenoid valve.
- F11-34(4) It is necessary to wait a few seconds for the cuff and pressure sensor to stabilize and in this state, both the cuff pressure and heartbeat signals are smoothed by applying a simple moving average filter. The number of samples processed is kept in `AppSamplesCtr` in order to count the number of samples until it reaches the steady state threshold time.
- F11-34(5) Because a pulse is defined as a consecutive summit and valley as illustrated in Figure 11-32, in this state, every sample is inserted into the 5-sample sliding window in order to detect a summit.
- F11-34(6) In this state, every sample is inserted into the 5-sample sliding window in order to detect a valley. If the consecutive summit and valley form a valid pulse as illustrated in Figure 11-32, then the time-domain features of the pulse are inserted into the array of pulses `AppPulsesAtRampUp`. The process goes back and forth between finding summits and valleys until the cuff pressure is above the systolic pressure at a maximum threshold defined in mmHg by `APP_MAX_CUFF_PRESSURE_MMHG`.
- F11-34(7) Once the systolic pressure is reached, the pressure in the cuff is held for a few seconds in order to test for any leaks. The function `AppCuffCtrl()` is called with the appropriate arguments to stop the air pump motor and close the solenoid valve, which in turn will hold the cuff pressure. The initial pressure is stored in `AppLeakTest.Pressure.Init`.

-
- F11-34(8) In this state, the cuff pressure signal is smoothed by applying a simple moving average filter and the result is monitored for any leaks. The number of samples is counted in order to determine the amount of time to test for leaks. Such time is defined in seconds by `APP_MAX_CUFF_HOLDING_TIME`.
 - F11-34(9) In this state, the cuff is deflated by calling `AppCuffCtrl()` with the right arguments to stop the air pump motor and open the solenoid valve.
 - F11-34(10) In a similar fashion to the ramp-up, in this state, every sample is inserted into the 5-sample sliding window in order to detect a summit.
 - F11-34(11) In this state, every sample is inserted into the 5-sample sliding window in order to detect a valley. A consecutive summit and valley may be a potential pulse. If the consecutive summit and valley form a valid pulse as illustrated in Figure 11-32, then the time-domain features of the pulse are inserted into the array of pulses `AppPulsesAtRampDown`. The process goes back and forth between finding summits and valleys until the cuff pressure is below the diastolic pressure at a minimum threshold defined in mmHg by `APP_MIN_CUFF_PRESSURE_MMHG`.
 - F11-34(12) The results are calculated from the two sets of pulses, `AppPulsesAtRampUp` and `AppPulsesAtRampDown` with the algorithm described in section 11-8-1 “Biomedical Signal Analysis” on page 246. The function `AppCalcResults()` iterates through the two arrays of pulses and the results are stored in `AppResults`.
 - F11-34(13) An error occurs in case any of the results are out of the ranges defined by `APP_MIN_BLOOD_PRESSURE_DIA_MMHG`, `APP_MAX_BLOOD_PRESSURE_DIA_MMHG` for the diastolic pressure result, the range defined in mmHg by `APP_MIN_BLOOD_PRESSURE_SYS_MMHG`, `APP_MAX_BLOOD_PRESSURE_SYS_MMHG` for the systolic pressure result and the range defined in beats-per-minute by `APP_MIN_HEARTRATE_BPM` and `APP_MAX_HEARTRATE_BPM` for the heart rate result.

11-9 SUMMARY

This chapter explained the anatomy and physiology of the organs involved in the regulation of the arterial blood pressure and the theory of operation of a blood pressure monitor including the hardware and software.

This example application demonstrated how simple it is to implement a blood pressure monitor using a combination of Micrium's µC/OS-II and Freescale hardware. It also demonstrated one of the most important features of µC/OS-II: *Message Queues*.

Message queues are built into each task and the user can send messages directly to a task from another task or an ISR. Similar to the rest of medical applications featured in this book, task message queues are used in this example because they allow the developer to encapsulate each task's functionality into a clean and simple message-based API. For example, we used the DAQ task's built-in message queue as a means to receive different types of commands from other tasks to perform an action like start or stop the simulator, start or stop the data acquisition and process samples from the ADC's ISRs.

This example provides a solid platform to create a commercial medical product based on a task-oriented approach. Most commercial blood pressure monitors come with an LCD display and this type of display control is a great candidate to encapsulate into a task and use the task message queue to receive commands to update the display.

Other blood pressure monitors like the ones used at an Intensive Care Unit (ICU) inflate and deflate the cuff every two minutes and require some type of connectivity to report the blood pressure readings to a central station where healthcare personnel can monitor the patients. Regardless of the communication protocol, chances are that you may need at least two more tasks to handle the reception and transmission of packets and one more task to handle all time-outs related to the communication protocol.

Freescale and Micrium support the addition of other functionality to this blood pressure monitor by offering more tower system compatible peripherals and software stacks. Visit the Freescale and Micrium websites to learn more about other products that can help you take your design to the next level.

Appendix

A

μ C/OS-II Port for the Cortex-M4

This appendix describes the adaptation of μ C/OS-II to the Cortex-M4 which is called a *Port*.

The port files are found in the following directory:

\Micrium\Software\uCOS-II\Ports\ARM-Cortex-M4\Generic\IAR

The port consists of three files:

`os_cpu.h`
`os_cpu_c.c`
`os_cpu_a.asm`

They are described in the following sections.

Appendix A

A-1 OS_CPU.H

`os_cpu.h` contains processor- and implementation-specific `#defines` constants, macros, and typedefs. `os_cpu.h` is shown in Listing A-1.

```

#ifndef OS_CPU_H                                (1)
#define OS_CPU_H
#ifndef OS_CPU_GLOBALS                         (2)
#define OS_CPU_EXT
#else
#define OS_CPU_EXT extern
#endif

#ifndef OS_CPU_EXCEPT_STK_SIZE                 (3)
#define OS_CPU_EXCEPT_STK_SIZE      128u
#endif

/*
***** DEFINES *****
*/
#ifndef __ARMVFP__
#define OS_CPU_ARM_FP_EN       1u          (4)
#else
#define OS_CPU_ARM_FP_EN       0u
#endif

/*
***** OS TICK INTERRUPT PRIORITY CONFIGURATION *****
*/
#define OS_CPU_CFG_SYSTICK_PRIO    0u          (5)

```

Listing A-1 `os_cpu.h` (first part)

- LA-1(1) Typical multiple header file inclusion protection.
- LA-1(2) `OS_CPU_GLOBALS` and `OS_CPU_EXT` allow us to declare global variables that are specific to this port. However, the port doesn't contain any global variables and thus these statements are just included for completeness and consistency.
- LA-1(3) The default exception stack size is 128 `OS_STK` entries.

-
- LA-1(4) The IAR pragma `__ARMVFP__` allows you to enable and disable floating point support from the IDE's project settings.
 - LA-1(5) For systems that don't need any high, real-time priority interrupts; the tick interrupt should be configured as the highest priority interrupt but won't adversely affect system operations.

For systems that need one or more high, real-time interrupts; these should be configured higher than the tick interrupt which MAY delay execution of the tick interrupt.

The header file `os_cpu.h` also declares the compiler-specific data types and function prototypes as shown in Listing A-2:

```
/*
***** DATA TYPES *****
*/
typedef unsigned char  BOOLEAN;                                (1)
typedef unsigned char  INT8U;
typedef signed   char  INT8S;
typedef unsigned short INT16U;
typedef signed   short INT16S;
typedef unsigned int   INT32U;
typedef signed   int   INT32S;
typedef float        FP32;
typedef double       FP64;
typedef unsigned int  OS_STK;
typedef unsigned int  OS_CPU_SR;

/*
***** CRITICAL SECTION MANAGEMENT *****
*/
#define  OS_CRITICAL_METHOD      3u                                (2)

#if OS_CRITICAL_METHOD == 3u
#define  OS_ENTER_CRITICAL()    {cpu_sr = OS_CPU_SR_Save(); }
#define  OS_EXIT_CRITICAL()     {OS_CPU_SR_Restore(cpu_sr); }
#endif
```

Listing A-2 `os_cpu.h` (second part)

Appendix A

- LA-2(1) The data types are specific to IAR's compiler.
- LA-2(2) The µC/OS-II port for the ARM Cortex-M4 disables and enables interrupts by preserving the state of interrupts. Generally speaking you would store the state of the interrupt disable flag in the local variable `cpu_sr` and then disable interrupts. `cpu_sr` is allocated in all of uC/OS-II's functions that need to disable interrupts. You would restore the interrupt disable state by copying back `cpu_sr` into the CPU's status register.

Continuing with the same header file `os_cpu.h` in Listing A-3:

```

/*
*****Cortex-M4 Miscellaneous*****
*/
#define OS_STK_GROWTH      1u          (1)
#define OS_TASK_SW()        OSCtxSw()

/*
*****GLOBAL VARIABLES*****
*/
OS_CPU_EXT OS_STK  OS_CPU_ExceptStk[OS_CPU_EXCEPT_STK_SIZE];           (2)
OS_CPU_EXT OS_STK *OS_CPU_ExceptStkBase;

/*
*****FUNCTION PROTOTYPES*****
*/
#if OS_CRITICAL_METHOD == 3u          (3)
OS_CPU_SR  OS_CPU_SR_Save    (void);
void       OS_CPU_SR_Restore (OS_CPU_SR cpu_sr);
#endif

void   OSCtxSw            (void);
void   OSIntCtxSw         (void);
void   OSStartHighRdy     (void);

void   OS_CPU_PendSVHandler (void);                                         (4)

void   OS_CPU_SysTickHandler (void);                                         (5)
void   OS_CPU_SysTickInit   (INT32U     cnts);

#if (OS_CPU_ARM_FP_EN > 0u)
void   OS_CPU_FP_Reg_Push  (OS_STK    *stkPtr);
void   OS_CPU_FP_Reg_Pop   (OS_STK    *stkPtr);
#endif

```

Listing A-3 `os_cpu.h` (third part)

LA-3(1) In the ARM architecture the stack grows from high to low memory.

Appendix A

- LA-3(2) The interrupt service routine (ISR) or trap handler (also called the exception handler) is declared as a global variable. This exception handler must vector to the assembly language function OSCTxSw().
- LA-3(3) The prototypes of mandatory µC/OS-II functions.
- LA-3(4) The Cortex-M4 processor provides a special interrupt handler specifically designed for use by context switch code. This is called the PendSV Handler and is implemented by OS_CPU_PendSVHandler(). The function is found in os_cpu_a.asm.
- LA-3(5) The Cortex-M4 has a timer dedicated for RTOS use called the SysTick. The code to initialize and handle the SysTick interrupt is found in os_cpu_c.c. Note that this code is part of the µC/OS-II port file and not the Board Support Package (BSP), because the SysTick is available to all Cortex-M4 implementations and is always handled the same by µC/OS-II.

A-2 OS_CPU_C.C

A µC/OS-II port requires that the following functions be declared:

```
OSInitHookBegin()  
OSInitHookEnd()  
OSTaskCreateHook()  
OSTaskDelHook()  
OSTaskIdleHook()  
OSTaskReturnHook()  
OSTaskStatHook()  
OSTaskStkInit()  
OSTaskSwHook()  
OSTCBInitHook()  
OSTimeTickHook()
```

The Cortex-M4 port implements two additional functions as described in the previous sections:

```
OS_CPU_SysTickHandler()  
OS_CPU_SysTickInit()
```

A-2-1 os_cpu_c.c – OSTaskIdleHook()

The idle task hook allows the port developer to extend the functionality of the idle task. For example, you can place the processor in low power mode when no other higher-priority tasks are running. This is especially useful in battery-powered applications. Listing A-4 shows the typical code for OSTaskIdleHook().

```
void OSTaskIdleHook (void)
{
    #if OS_APP_HOOKS_EN > 0u
        App_TaskIdleHook();                                     (1)
    #endif
}
```

Listing A-4 os_cpu_c.c – OSIdleTaskHook()

- LA-4(1) Application level hook functions are enabled by OS_APP_HOOKS_EN.
- LA-4(2) The application level idle task hook is called without any argument. And it *must not* make any blocking calls because the idle task must never block. In other words, it cannot call OSTimeDly(), OSTimeDlyHMSM(), or OSTaskSuspend() (to suspend ‘self’), and any of the OS???Pend() functions.

A-2-2 os_cpu_c.c – OSInitHookBegin() and OSInitHookEnd()

OSInitHookBegin() is called by OSInit() at the beginning of OSInit() and OSInitHookEnd() is called by OSInit() at the end of OSInit().

The OSInitHookBegin() function is where the exception stack gets cleared and initialized.

A-2-3 os_cpu_c.c – OSStatTaskHook()

OSTaskStatHook() allows the port developer to extend the functionality of the statistics task by allowing him/her to add additional statistics. OSStatTaskHook() is called every second by µC/OS-II's statistics task

A-2-4 os_cpu_c.c – OSTaskCreateHook()

OSTaskCreateHook() is called by OS_TCBInit() whenever a task is created. This function allows you or the user of your port to extend the functionality of µC/OS-II. OSTaskCreateHook() is called when µC/OS-II is done setting up most of the OS_TCB but before the OS_TCB is linked to the active task chain and before the task is made ready to run.

A-2-5 os_cpu_c.c – OSTaskDelHook()

OSTaskDelHook() gives the port developer the opportunity to add code specific to the port when a task is deleted. OSTaskDelHook() is called once the task has been removed from either the read list or the wait list. It is called before unlinking the task from µC/OS-II's internal linked list of active tasks.

A-2-6 os_cpu_c.c – OSTaskStkInit()

This function is called by either OSTaskCreate() or OSTaskCreateExt() to initialize the stack frame of the task being created. Generally speaking, the stack frame is made to look as if an interrupt has just occurred and all the CPU registers have been saved onto it.

Listing A-5 shows part of the Cortex-M4 code for OSTaskStkInit().

```

OS_STK *OSTaskStkInit (void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT16U opt)      (1)
{
    OS_STK *p_stk;

    p_stk      = ptos + 1u;                                (2)
    p_stk      = (OS_STK *) ((OS_STK) (p_stk) & 0xFFFFFFFF8u);   (3)

    *(--p_stk) = (OS_STK) 0x01000000L;                      (4)
    *(--p_stk) = (OS_STK) task;                            (5)
    *(--p_stk) = (OS_STK) OS_TaskReturn;                  (6)
    *(--p_stk) = (OS_STK) 0x12121212L;                   (7)
    *(--p_stk) = (OS_STK) 0x03030303L;
    *(--p_stk) = (OS_STK) 0x02020202L;
    *(--p_stk) = (OS_STK) 0x01010101L;
    *(--p_stk) = (OS_STK) p_arg;                          (8)
    *(--p_stk) = (OS_STK) 0x11111111L;                  (9)
    *(--p_stk) = (OS_STK) 0x10101010L;
    *(--p_stk) = (OS_STK) 0x09090909L;
    *(--p_stk) = (OS_STK) 0x08080808L;
    *(--p_stk) = (OS_STK) 0x07070707L;
    *(--p_stk) = (OS_STK) 0x06060606L;
    *(--p_stk) = (OS_STK) 0x05050505L;
    *(--p_stk) = (OS_STK) 0x04040404L;

```

Listing A-5 **os_cpu_c.c – OSTaskStkInit()**

LA-5(1) OSTaskStkInit() is called by OSTaskCreate() and is passed four arguments:

- 1 The task's entry point (i.e., the address of the task).
- 2 A pointer to a user-supplied data area that is be passed to the task when the task first executes.

- 3 A pointer to the top of the stack (`ptos`). It is assumed that `ptos` points to a free entry on the task stack. If `OS_STK_GROWTH` is set to 1, then `ptos` contains the highest valid address of the stack. Similarly, if `OS_STK_GROWTH` is set to 0, `ptos` contains the lowest valid address of the stack.
- 4 Finally, the fourth argument specifies options that can be used to alter the behavior of `OSTaskStkInit()`.

- LA-5(2) Load the stack pointer.
- LA-5(3) Align the stack to 8-bytes.
- LA-5(4) The Cortex-M4's PSR register is initialized. The initial value sets the 'T' bit in the PSR, which causes the Cortex-M4 to use Thumb instructions (this should always be the case).
- LA-5(5) This register corresponds to R15 which is the program counter. We initialize this register to point to the task entry point.
- LA-5(6) This register corresponds to R14 (the link register), which contains the return address of the task.
- LA-5(7) Registers R12, R3, R2 and R1 are initialized to a value that makes it easy for them to be identified when a debugger performs a memory dump.
- LA-5(8) R0 is the register used by the C compiler to pass the first argument to a function. Recall that the prototype for a task looks as shown below.

```
void MyTask (void *p_arg);
```

In this case, '`p_arg`' is simply passed in R0 so that when the task starts, it will think it was called as with any other function.

- LA-5(9) Registers R11, R10, R9 and R8, R7, R6, R5 and R4 are initialized to a value that makes it easy for them to be identified when a debugger performs a memory dump.

The stack frame of the task being created is shown in Figure A-1.

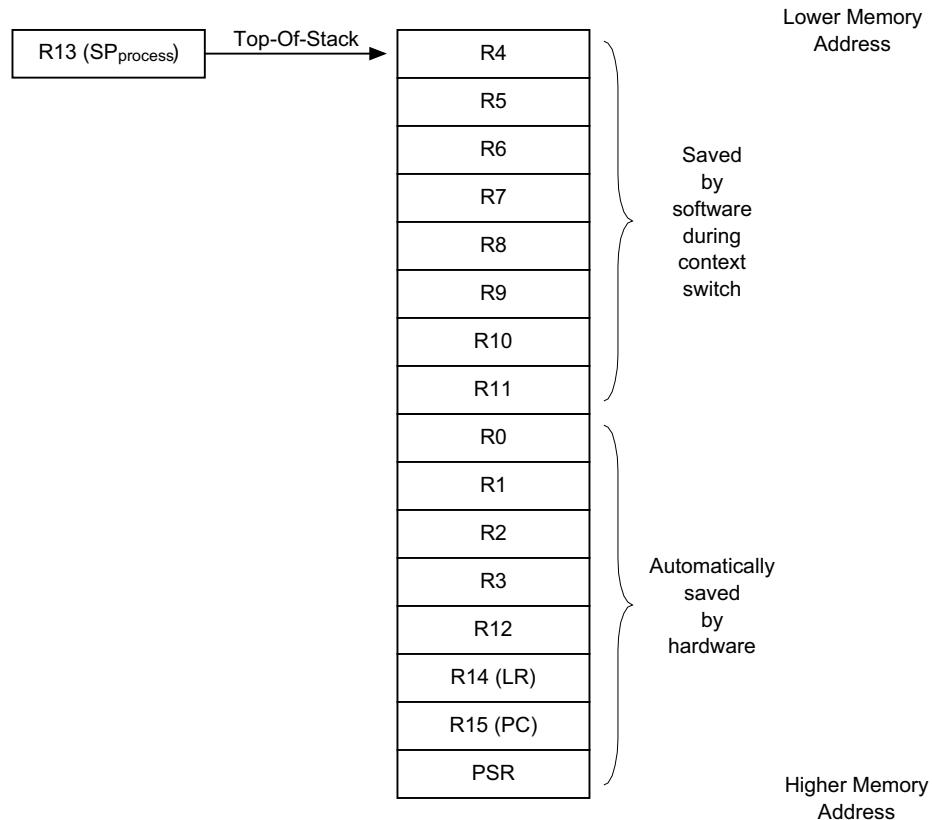


Figure A-1 Stack frame of task being created

Appendix A

If floating-point support is enabled then additional registers need to be taken into account. Code shows the additional registers that need to be initialized:

```
#if (OS_CPU_ARM_FP_EN > 0u)
    if ((opt & OS_TASK_OPT_SAVE_FP) != (INT16U)0) {
        *--p_stk = (OS_STK)0x02000000u;
        *--p_stk = (OS_STK)0x41F80000u;
        *--p_stk = (OS_STK)0x41F00000u;
        *--p_stk = (OS_STK)0x41E80000u;
        *--p_stk = (OS_STK)0x41E00000u;
        *--p_stk = (OS_STK)0x41D80000u;
        *--p_stk = (OS_STK)0x41D00000u;
        *--p_stk = (OS_STK)0x41C80000u;
        *--p_stk = (OS_STK)0x41C00000u;
        *--p_stk = (OS_STK)0x41B80000u;
        *--p_stk = (OS_STK)0x41B00000u;
        *--p_stk = (OS_STK)0x41A80000u;
        *--p_stk = (OS_STK)0x41A00000u;
        *--p_stk = (OS_STK)0x41980000u;
        *--p_stk = (OS_STK)0x41900000u;
        *--p_stk = (OS_STK)0x41880000u;
        *--p_stk = (OS_STK)0x41800000u;
        *--p_stk = (OS_STK)0x41700000u;
        *--p_stk = (OS_STK)0x41600000u;
        *--p_stk = (OS_STK)0x41500000u;
        *--p_stk = (OS_STK)0x41400000u;
        *--p_stk = (OS_STK)0x41300000u;
        *--p_stk = (OS_STK)0x41200000u;
        *--p_stk = (OS_STK)0x41100000u;
        *--p_stk = (OS_STK)0x41000000u;
        *--p_stk = (OS_STK)0x40E00000u;
        *--p_stk = (OS_STK)0x40C00000u;
        *--p_stk = (OS_STK)0x40A00000u;
        *--p_stk = (OS_STK)0x40800000u;
        *--p_stk = (OS_STK)0x40400000u;
        *--p_stk = (OS_STK)0x40000000u;
        *--p_stk = (OS_STK)0x3F800000u;
        *--p_stk = (OS_STK)0x00000000u;
    }
#endif
```

Listing A-6 **os_cpu_c.c – OSTaskStkInit()**

A-2-7 os_cpu_c.c – OSTaskSwHook()

OSTaskSwHook() is called when μC/OS-II performs a context switch. In fact, OSTaskSwHook() is called after saving the context of the task being suspended. Also, OSTaskSwHook() is called with interrupts disabled.

You can use this function to save/restore the contents of floating-point registers or MMU registers, to keep track of task-execution time and of how many times the task has been switched in, and more.

Listing A-7 shows the code for OSTaskSwHook().

```
void OSTaskSwHook (void)
{
    #if (OS_CPU_ARM_FP_EN > 0u)
        if ((OSTCBCur->OSTCBOpt & OS_TASK_OPT_SAVE_FP) != (INT16U)0) {
            OS_CPU_FP_Reg_Push(OSTCBCur->OSTCBStkPtr);
        }

        if ((OSTCBHighRdy->OSTCBOpt & OS_TASK_OPT_SAVE_FP) != (INT16U)0) {
            OS_CPU_FP_Reg_Pop(OSTCBHighRdy->OSTCBStkPtr);
        }
    #endif

    #if OS_APP_HOOKS_EN > 0u
        App_TaskSwHook();
    #endif
}
```

Listing A-7 **os_cpu_c.c – OSTaskSwHook()**

- LA-7(1) If floating-point support is enabled, the FPU registers are pushed and popped.
- LA-7(2) The application hook function *must not* make any blocking calls and should perform its function as quickly as possible. The application level task switch hook is not passed any arguments.

A-2-8 os_cpu_c.c – OSTimeTickHook()

`OSTimeTickHook()` gives the port developer the opportunity to add code that will be called by `OSTimeTick()`. `OSTimeTickHook()` is called from the tick ISR and must not make any blocking calls (it would be allowed to anyway) and must execute as quickly as possible.

Listing A-8 shows the typical code for `OSTimeTickHook()`.

```
void OSTimeTickHook (void)
{
    #if OS_APP_HOOKS_EN > 0u
        App_TimeTickHook();                                         (1)
    #endif

    #if OS_TMR_EN > 0u
        OSTmrCtr++;
        if (OSTmrCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC)) {
            OSTmrCtr = 0;
            OSTmrSignal();
        }
    #endif
}
```

Listing A-8 **os_cpu_c.c – OSTimeTickHook()**

- LA-8(1) If the application developer wants his/her own function to be called when a tick interrupt occurs, the developer needs to declare the application level function `App_TimeTickHook()`.
- LA-8(2) μC/OS-II timers system is driven by this hook.

A-2-9 os_cpu_c.c – OS_CPU_SysTickHandler()

`OS_CPU_SysTickHandler()` is automatically invoked by the Cortex-M4 when a SysTick interrupt occurs and interrupts are enabled. For this to happen, however, the address of `OS_CPU_SysTickHandler()` must be placed in the interrupt vector table at the SysTick entry (the 15th entry in the vector table of the Cortex-M4).

Listing A-9 shows the Cortex-M4 code for `OS_CPU_SysTickHandler()`.

```
void  OS_CPU_SysTickHandler (void)          (1)
{
    OS_CPU_SR  cpu_sr;

    CPU_CRITICAL_ENTER();                   (2)
    OSIntNestingCtr++;
    CPU_CRITICAL_EXIT();                  (3)
    OSTimeTick();                         (4)
    OSIntExit();
}
```

Listing A-9 os_cpu_c.c – OS_CPU_SysTickHandler()

- LA-9(1) When the Cortex-M4 enters an interrupt, the CPU automatically saves critical registers (R0, R1, R2, R3, R12, PC, LR and XPSR) onto the current task's stack and switches to the Main Stack (MSP) to handle the interrupt.

This means that R4 through R11 are not saved when the interrupt starts and the ARM Architecture Procedure Call Standard (AAPCS) requires that all interrupt handlers preserve the values of the other registers, if they are required during the ISR.

- LA-9(2) The interrupt nesting counter is incremented in a critical section because the SysTick interrupt handler could be interrupted by a higher priority interrupt.
- LA-9(3) The μC/OS-II tick interrupt needs to call `OSTimeTick()`.
- LA-9(4) Every interrupt handler must call `OSIntExit()` at the end of the handler.

A-2-10 os_cpu_c.c – OS_CPU_SysTickInit()

`OS_CPU_SysTickInit()` is called by your application code to initialize the SysTick interrupt.

Listing A-10 shows the Cortex-M4 code for `OS_CPU_SysTickInit()`.

```

void  OS_CPU_SysTickInit (INT32U  cnts)                                (1)
{
    INT32U    prio;

    OS_CPU_CM4_NVIC_ST_RELOAD = cnts - 1u;

    prio  = OS_CPU_CM4_NVIC_SHPRI3;                                         (2)
    prio &= DEF_BIT_FIELD(24, 0);
    prio |= DEF_BIT_MASK(OS_CPU_CFG_SYSTICK_PRIO, 24);

    OS_CPU_CM4_NVIC_SHPRI3 = prio;

    OS_CPU_CM4_NVIC_ST_CTRL |= OS_CPU_CM4_NVIC_ST_CTRL_CLK_SRC |
                           OS_CPU_CM4_NVIC_ST_CTRL_ENABLE;

    OS_CPU_CM4_NVIC_ST_CTRL |= OS_CPU_CM4_NVIC_ST_CTRL_INTEN;
}

```

Listing A-10 **os_cpu_c.c – OS_CPU_SysTickInit()**

- LA-10(1) `OS_CPU_SysTickInit()` must be informed about the counts to reload into the SysTick timer. The counts to reload depend on the CPU clock frequency and the configured tick rate.

The reload value is typically computed by the first application task to run as follows:

```

cpu_clk_freq = BSP_CPU_ClkFreq();
cnts          = cpu_clk_freq / (INT32U)OS_CFG_TICK_RATE_HZ;

```

`BSP_CPU_ClkFreq()` is a BSP function that returns the CPU clock frequency. We then compute reload counts from the tick rate.

-
- LA-10(2) The SysTick interrupt is set to the lowest priority because ticks are mostly used for coarse time delays and timeouts, and we want application interrupts to be handled first.

A-3 OS_CPU_A.ASM

`os_cpu_a.asm` contains processor-specific code for three functions that must be written in assembly language:

```
OSStartHighRdy()  
OSCtxSw()  
OSIntCtxSw()
```

In addition, the Cortex-M4 requires the definition of a function to handle the PendSV exception.

```
OS_CPU_PendSVHandler()
```

And if floating-point support is enabled, then a couple of functions need to be declared to save and restore the FPU registers:

```
OS_CPU_FP_Reg_Push()  
OS_CPU_FP_Reg_Pop()
```

A-3-1 os_cpu_a.asm – OSStartHighRdy()

`OSStartHighRdy()` is called by `osstart()` to start the process of multitasking. μC/OS-II will switch to the highest priority task that is ready to run.

Listing A-11 shows the Cortex-M4 code for `OSStartHighRdy()`.

Appendix A

```

OSStartHighRdy
    LDR    R0, =NVIC_SYSPRI14          (1)
    LDR    R1, =NVIC_PENDSV_PRI
    STRB   R1, [R0]
    MOVS   R0, #0
    MSR    PSP, R0
    LDR    R0, =NVIC_INT_CTRL          (2)
    LDR    R1, =NVIC_PENDSVSET
    STR    R1, [R0]
    CPSIE  I                          (3)
OSStartHang
    B      OSStartHang                (4)

```

Listing A-11 os_cpu_a.asm – OSStartHighRdy()

- LA-11(1) OSStartHighRdy() starts by setting the priority level of the PendSV handler. The PendSV handler is used to perform all context switches and is always set at the lowest priority so that it executes after the last nested ISR.
- LA-11(2) The PendSV handler is invoked by ‘triggering’ it. However, the PendSV will not execute immediately because it is assumed that interrupts are disabled.
- LA-11(3) Interrupts are enabled and this should cause the Cortex-M4 processor to vector to the PendSV handler (described later).
- LA-11(4) The PendSV handler should pass control to the highest-priority task that was created and the code should never come back to OSStartHighRdy().

A-3-2 os_cpu_a.asm – OSCtxSw() and OSIntCtxSw()

OSCtxSw() is called by OSSched() and OS_Sched0() to perform a context switch from a task.

OSIntCtxSw() is called by OSIntExit() to perform a context switch after an ISR has completed.

Both of these functions simply ‘trigger’ the PendSV exception handler, which does the actual context switching.

Listing A-12 shows the Cortex-M4 code for OSCtxSw() and OSIntCtxSw().

```
OSCtxSw
    LDR    R0, =NVIC_INT_CTRL
    LDR    R1, =NVIC_PENDSVSET
    STR    R1, [R0]
    BX    LR
OSIntCtxSw
    LDR    R0, =NVIC_INT_CTRL
    LDR    R1, =NVIC_PENDSVSET
    STR    R1, [R0]
    BX    LR
```

Listing A-12 **os_cpu_a.asm – OSCtxSw() and OSIntCtxSw()**

A-3-3 os_cpu_a.asm – OS_CPU_PendSVHandler()

`OS_CPU_PendSVHandler()` is the code that performs a context switch initiated by a task, or at the completion of an ISR. `OS_CPU_PendSVHandler()` is invoked by `osStartHighRdy()`, `OSCtxSw()` and `OSIntCtxSw()`.

Listing A-13 shows the Cortex-M4 code for `OS_CPU_PendSVHandler()`.

Appendix A

```

OS_CPU_PendSVHandler
    CPSID    I                                (1)
    MRS     R0, PSP                            (2)
    CBZ     R0, OS_CPU_PendSVHandler_nosave
    SUBS   R0, R0, #0x20                      (3)
    STM    R0, {R4-R11}
    LDR    R1, =OSTCBCurPtr                  (4)
    LDR    R1, [R1]
    STR    R0, [R1]

OS_CPU_PendSVHandler_nosave
    PUSH   {R14}                            (5)
    LDR    R0, =OSTaskSwHook
    BLX    R0
    POP    {R14}
    LDR    R0, =OSPrioCur                   (6)
    LDR    R1, =OSPrioHighRdy
    LDRB   R2, [R1]
    STRB   R2, [R0]
    LDR    R0, =OSTCBCurPtr
    LDR    R1, =OSTCBHighRdyPtr            (7)
    LDR    R2, [R1]
    STR    R2, [R0]
    LDR    R0, [R2]
    LDM    R0, {R4-R11}                     (8)
    ADDS   R0, R0, #0x20
    MSR    PSP, R0                          (9)
    ORR    LR, LR, #0x04
    CPSIE  I                                (10)
    BX    LR                               (11)

```

Listing A-13 os_cpu_a.asm – OS_CPU_PendSVHandler()

- LA-13(1) OS_CPU_PendSVHandler() starts by disabling all interrupts because interrupt should not occur during a context switch.
- LA-13(2) This code skips saving the remaining eight registers if this is the first time the PendSV is called. In other words, when OSStartHighRdy() triggers the PendSV handler, there is nothing to save from the ‘previous task’ as there is no previous task.
- LA-13(3) If OS_CPU_PendSVHandler() is invoked from either OSCtxSw() or OSIntCtxSw(), the PendSV handler saves the remaining eight CPU registers (R4 through R11) onto the stack of the task switched out.

-
- LA-13(4) OS_CPU_PendSVHandler() saves the stack pointer of the task switched out into that task's OS_TCB. Note that the first field of an OS_TCB is .StkPtr (the task's stack pointer), which makes it convenient for assembly language code since there are no offsets to determine.
 - LA-13(5) The task switch hook (OSTaskSwHook()) is then called.
 - LA-13(6) OS_CPU_PendSVHandler() copies the priority of the new task into the priority of the current task, i.e.:

```
OSPrioCur = OSPrioHighRdy;
```
 - LA-13(7) OS_CPU_PendSVHandler() copies the pointer to the new task's OS_TCB into the pointer to the current task's OS_TCB, i.e.,:

```
OSTCBCurPtr = OSTCBHighRdyPtr;
```
 - LA-13(8) OS_CPU_PendSVHandler() retrieves the stack pointer from the new task's OS_TCB.
 - LA-13(9) CPU registers R4 through R11 from the new task are loaded into the CPU.
 - LA-13(10) The task stack pointer is updated with the new top-of-stack pointer.
 - LA-13(11) Interrupts are re-enabled since we are finished performing the critical portion of the context switch. If another interrupt occurs before we return from the PendSV handler, the Cortex-M4 knows that there are eight registers still saved on the stack, and there would be no need for it to save them. This is called Tail Chaining and it makes servicing back-to-back interrupts quite efficient on the Cortex-M4.
 - LA-13(12) By performing a return from the PendSV handler, the Cortex-M4 processor knows that it is returning from interrupt and will thus restore the remaining registers.

Appendix A

Appendix

B

Micrium's µC/Probe

µC/Probe is an award-winning Microsoft Windows™-based application that allows a user to display or change the value (at run time) of virtually any variable or memory location on a connected embedded target. The user simply populates µC/Probe's graphical environment with gauges, numeric indicators, tables, graphs, virtual LEDs, bar graphs, sliders, switches, push buttons, and other components, and associates each of these to a variable or memory location.

With µC/Probe, it is not necessary to instrument the target code in order to display or change variables at run time. In fact, there is no need to add `printf()` statements, hardware such as Light Emitting Diodes (LEDs), Liquid Crystal Displays (LCDs), or use any other means to get visibility inside an embedded target at run time.

µC/Probe is available in different editions from Micrium (See section B-3 “Downloading µC/Probe” on page 279).

The examples provided with this book assume that you have downloaded and installed one of these editions of µC/Probe.

This appendix provides a brief introduction to µC/Probe.

B-1 SYSTEM OVERVIEW

µC/Probe is a Windows application designed to read and write the memory of any embedded target processor during run-time. Memory locations are mapped to a set of virtual controls and indicators placed on a dashboard. Figure B-1 shows an overview of the system and data flow.

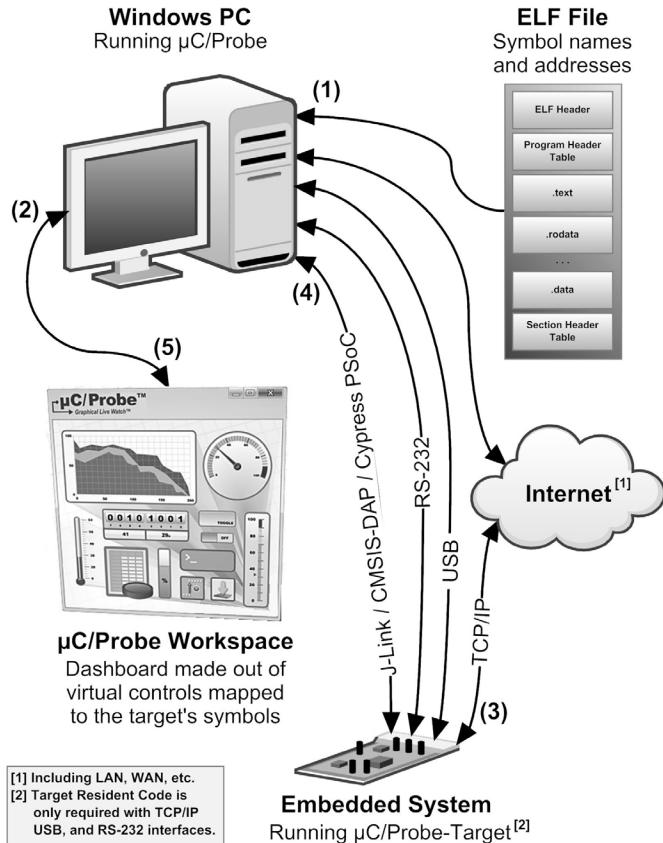


Figure B-1 μC/Probe Data Flow Diagram

- FB-1(1) You have to provide μC/Probe with an ELF file with DWARF-2, -3 or -4 debugging information. The ELF file is generated by your toolchain's linker. μC/Probe parses the ELF file and reads the addresses of each of the embedded target's symbols (i.e. global variables) and creates a catalog known as *symbol browser*, which will be used by you during design-time to select the symbols you want to display on your dashboard. Refer to the document μC/Probe Target Manual for more information on installing the μC/Probe Target C files and building the ELF file.

Alternatively, you can also provide a chip definition file that contains the chip's peripheral register addresses or provide your own custom XML based symbol file for those cases when your toolchain cannot generate one of the supported ELF formats.

- FB-1(2) During design-time, you create a µC/Probe workspace using a Windows PC and µC/Probe. You design your own dashboard by dragging and dropping virtual controls and indicators onto a *data screen*. Each virtual control and indicator needs to be mapped to an embedded target's symbol by selecting it from the symbol browser. See the µC/Probe User's Manual for more information on creating your own dashboard with µC/Probe.
- FB-1(3) Before proceeding to the run-time stage, µC/Probe needs to be configured to use one of the following communication interfaces: J-Link, CMSIS-DAP, Cypress PSoC Prog, USB, RS-232 or TCP/IP. In order to start the run-time stage, you click the *Run* button and µC/Probe starts making requests to read the value of all the memory locations associated with each virtual control and indicator (i.e. buttons and gauges respectively). At the same time, µC/Probe sends commands to write the memory locations associated with each virtual control (i.e. buttons on a click event).
- FB-1(4) In the case of a reading request, the embedded target responds with the latest value. In the case of a write command, the embedded target responds with an acknowledgement. In case the communication of your choice is USB, RS-232 or TCP/IP, refer to the document µC/Probe Target Manual for more information on all you need in regards to the firmware that implements the communication interface that runs on the embedded target.
- FB-1(5) µC/Probe parses the responses from the embedded target and updates the virtual controls and indicators.

B-2 µC/PROBE ON THE KINETIS

Figure B-2 shows a block diagram of a typical development environment with the addition of µC/Probe as used with the TWR-K53N512, available from Freescale.

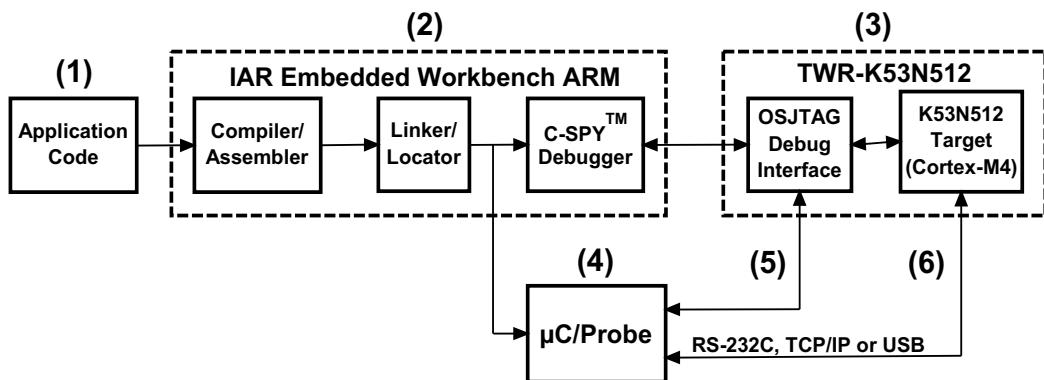


Figure B-2 Development environment using the TWR-K53N512

- FB-2(1) This is the application code you are developing. It is assumed that you are using µC/OS-II provided with this book. However, µC/Probe does not require an RTOS, and can work with or without an RTOS.
- FB-2(2) The examples provided with this book assume the IAR Embedded Workbench for ARM, but µC/Probe works with any toolchain as long as the linker/locator is able to produce a .ELF or .IEEE695 output file.
- FB-2(3) The TWR-K53N512 evaluation board available from Freescale is able to interface to a J-Link provided by Segger. The J-Link allows the C-SPY™ debugger to download Flash code onto the on-board Cortex-M4-based Micro Controller Unit (MCU). C-SPY also allows you to debug application code.
- FB-2(4) µC/Probe reads the exact same .ELF or .IEEE695 output file produced by the linker/locator. From this file, µC/Probe is able to extract names, data types and addresses of all the global variables of the application code. This information allows µC/Probe to display any of the values of the variables using the display objects available in µC/Probe (gauges, meters, virtual LEDs, bar graphs, numeric indicators, graphs, and more).

-
- FB-2(5) µC/Probe is able to interface to the Cortex-M4 processor via the SWD interface of the J-Link. In fact, both the C-SPY debugger and µC/Probe can access the target through the J-Link at the same time. This allows µC/Probe to monitor or change any target variable while you are stepping through the code using the C-SPY debugger. Interfacing through the J-Link also has the advantage of not requiring any target resident code to interface to µC/Probe.
- FB-2(6) µC/Probe can also interface to the TWR-K53N512 evaluation board using RS-232C, Ethernet (using TCP/IP) or USB.

Target resident code must be added when using RS-232C. This code is however provided by Micrium, and the user needs only to add it to the application code as part of the build. Also, unlike when using the J-Link, target data can only be displayed or changed by µC/Probe when the target is running. However, the RS-232C interface allows data to be collected faster than through the onboard J-Link.

Target resident code is also required if using the Ethernet port on the TWR-K53N512 evaluation board. In fact, you'll need a full TCP/IP stack such as Micrium's µC/TCP-IP. Again, data can only be displayed and changed when the target is running. However, the Ethernet interface provides the best throughput and data update rates for µC/Probe.

Finally, µC/Probe also works over the onboard USB-Device connector and requires a USB-Device (Vendor class) stack such as Micrium's µC/USB-Device with the Vendor class option. As with the RS-232C and TCP/IP, data can only be displayed or changed when the target is running when using this interface.

B-3 DOWNLOADING µC/PROBE

µC/Probe is an application that allows users to display or change the value (at run time) of virtually any variable or memory location on a connected embedded target.

µC/Probe is used in all of the examples described in Chapter 4, "Introduction to Medical Applications" on page 49 to gain run-time visibility.

The Educational Edition of µC/Probe is available for your evaluation free of charge. It may be used without a license as long the software is used for educational purposes:

<http://micrium.com/tools/ucprobe/software-and-docs/>

When you are satisfied that Micrium's µC/Probe is the right tool for you, you can purchase µC/Probe online and receive product updates and technical support.

µC/Probe is available for purchase in two editions, Basic and Professional.

µC/Probe Edition	Description
Basic Edition	All features of the free Educational Edition, but with no time limit, and the ability to import/export data screens.
Professional Edition	All features of the Basic Edition, plus: <ul style="list-style-type: none">•Scatter X-Y Chart•Terminal Window Control•Scripting Control•Microsoft® Excel® Bridge Control•µC/Trace Trigger Control

Both the Basic and Professional editions of µC/Probe can be purchased with a renewable or permanent license. All license types are locked to a specific computer.

To purchase µC/Probe go to: <http://micrium.com/tools/ucprobe/buy/>

Appendix

C

IAR Systems IAR Embedded Workbench for ARM

IAR Embedded Workbench is a set of highly sophisticated and easy-to-use development tools for embedded applications. It integrates the IAR C/C++ Compiler™, assembler, linker, librarian, text editor, project manager and C-SPY® Debugger in an integrated development environment (IDE).

With its built-in chip-specific code optimizer, IAR Embedded Workbench generates very efficient and reliable FLASH/ROMable code for ARM devices. In addition to this solid technology, the IAR Systems also provides professional world-wide technical support.

The KickStart™ edition of IAR Embedded Workbench is free of charge and you may use it for as long as you want. KickStart tools are ideal for creating small applications, or for getting started fast on a new project. The only requirement is that you register to obtain a license key.

The KickStart edition is code-size limited, but a fully functional integrated development environment that includes a project manager, editor, compiler, assembler, linker, librarian, and debugger tools. A complete set of user guides is included in PDF format.

The KickStart edition corresponds to the latest release of the full edition of IAR Embedded Workbench, with the following exceptions:

- It has a code size limitation (32 Kbytes).
- It does not include source code for runtime libraries.
- It does not include support for MISRA C.
- There is limited technical support.

The KickStart edition of IAR Embedded Workbench allows you to run all of the examples provided in this book.

C-1 IAR EMBEDDED WORKBENCH FOR ARM – HIGHLIGHTS

The full version of the IAR Embedded Workbench for ARM offers the following features.

- Support for:
 - ARM7™ (ARM7TDMI, ARM7TDMI-S and ARM720T)
 - ARM7E™ (ARM7EJ-S)
 - ARM9™ (ARM9TDMI, ARM920T, ARM922T and ARM940T)
 - ARM9E™ (ARM926EJ-S, ARM946E-S and ARM966E-S, ARM968E-S)
 - ARM10ETM (ARM1020E and ARM1022E)
 - ARM11™
 - SecurCore™ (SC000, SC100, SC110, SC200, SC210, SC300)
 - Cortex-A5™
 - Cortex-A8™
 - Cortex-R4(F)
 - Cortex-M0™
 - Cortex-M1™
 - Cortex-M3™
 - Cortex-M4™
 - XScale™
- Most compact and efficient code
- ARM Embedded Application Binary Interface (EABI)
- Extensive support for hardware and RTOS-aware debugging
- Total solutions for ARM
- New Cortex-M4 debug features

- Function profiler
- Interrupt graph window
- Data log window
- MISRA C:2004 support
- Extensive device support
- Over 1400 example projects
- µC/OS-II Kernel Awareness built-into the C-Spy debugger

Figure C-1 shows a block diagram of the major EWARM components.

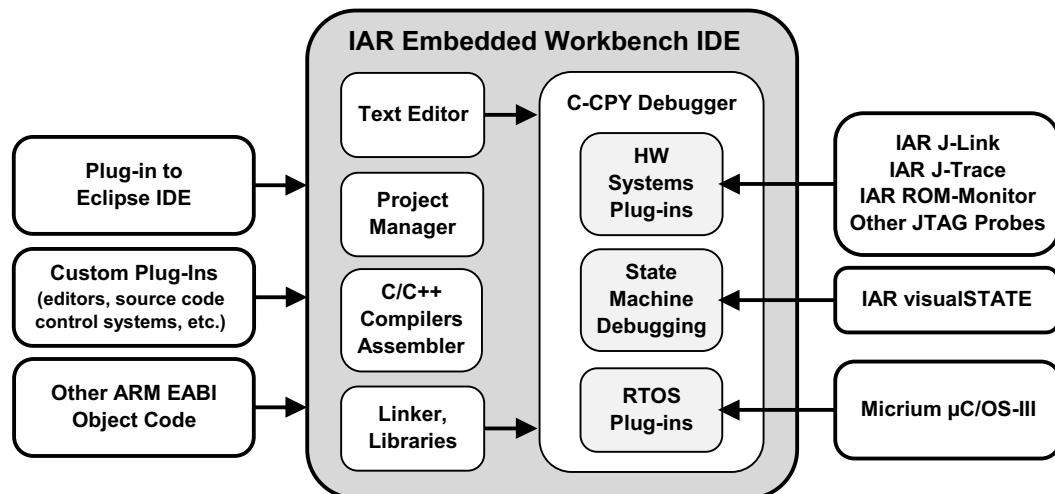


Figure C-1 IAR Embedded Workbench

C-2 MODULAR AND EXTENSIBLE IDE

- A seamlessly Integrated Development Environment (IDE) for building and debugging embedded applications
- Powerful project management allowing multiple projects in one workspace
- Build integration with IAR visualSTATE
- Hierarchical project representation
- Dockable and floating windows management
- Smart source browser
- Tool options configurable on global, group of source files, or individual source files level
- Multi-file compilation support for even better code optimization
- Flexible project building via batch build, pre/post-build or custom build with access to external tools in the build process.
- Integration with source code control systems

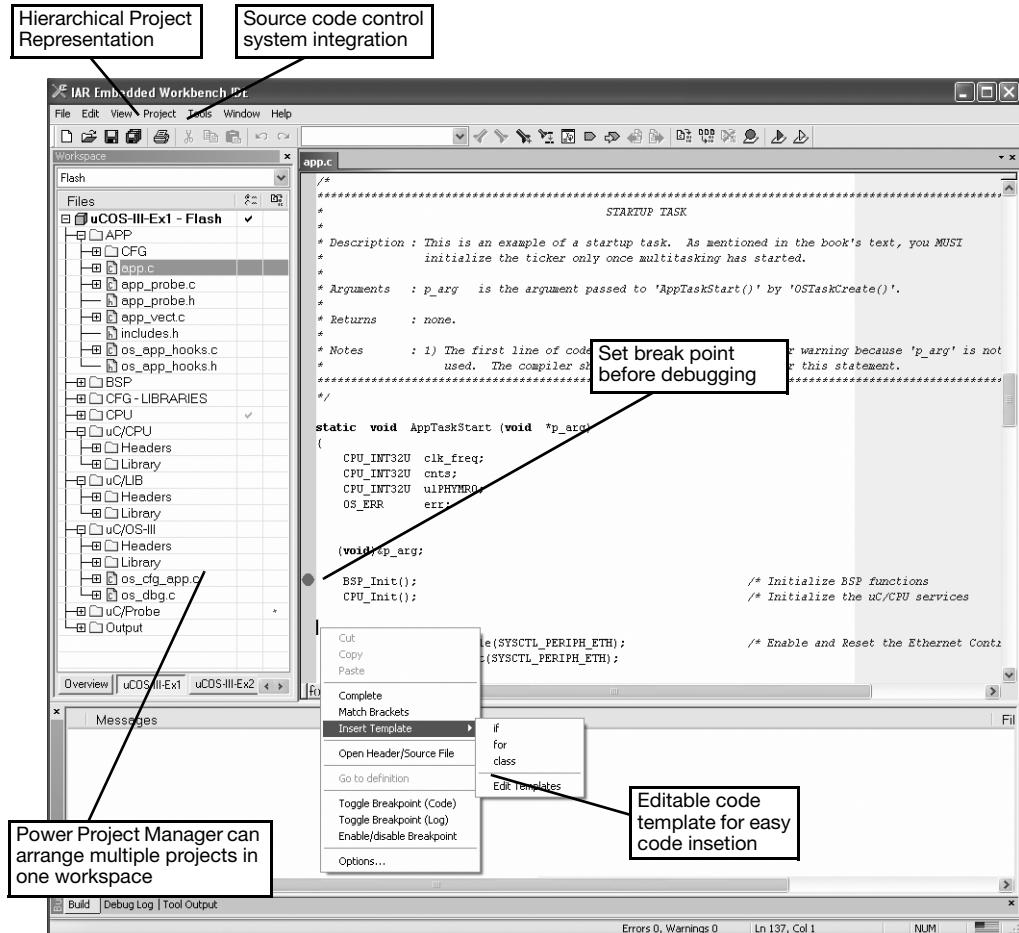


Figure C-2 IAR Embedded Workbench IDE

C-3 HIGHLY OPTIMIZING C/C++ COMPILER

- Support for C, EC++ and extended EC++ including templates, namespace, standard template library (STL) etc.
- ARM Embedded Application Binary Interface (EABI) and ARM Cortex Microcontroller Software Interface Standard (CMSIS) compliant
- Interoperability and binary compatibility with other EABI compliant tools
- Automatic checking of MISRA C rules
- Support for ARM, Thumb1 and Thumb2 processor modes
- Support for 4 Gbyte applications in all processor modes
- Support for 64-bit long
- Reentrant code
- 32- and 64-bit floating-point types in standard IEEE format
- Multiple levels of optimizations on code size and execution speed allowing different transformations enabled, such as function inlining, loop unrolling etc.
- Advanced global and target-specific optimizer generating the most compact and stable code
- Compressed initializers
- Support for ARM7, ARM7E, ARM9, ARM9E, ARM10E, ARM11, Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4, Cortex-R4 and Intel XScale
- Support for ARM, Thumb1 and Thumb2 processor modes
- Generates code for ARM VFP series of floating-point coprocessors
- Little/big endian mode

C-4 DEVICE SUPPORT

Device support on five levels:

- Core support - instruction set, debugger interface (for all supported devices)
- Header/DDF files - peripheral register names in C/asm source and debugger (for all supported devices)
- Flash loader for on-chip flash or off-chip EVB flash (for most of our supported devices)
- Project examples - varies from simple to fairly complex applications (for most of our supported devices)
- Detailed device support list at www.iar.com/ewarm

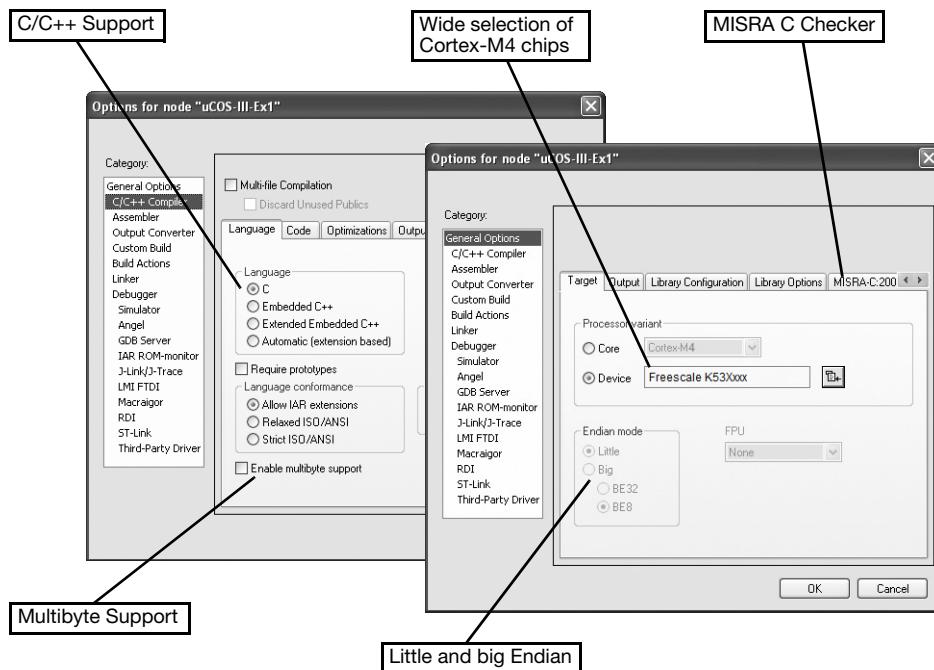


Figure C-3 Device Support

C-5 STATE-OF-THE-ART C-SPY® DEBUGGER

- Cortex-M4 SWV/SWO debugger support
- Complex code and data breakpoints
- User selectable breakpoint types (hardware/software)
- Unlimited number of breakpoints in flash via optional license for J-Link
- Runtime stack analysis - stack window to monitor the memory consumption and integrity of the stack
- Complete support for stack unwinding even at high optimization levels
- Profiling and code coverage performance analysis tools
- Trace utility with expressions, such as variables and register values, to examine execution history
- Versatile monitoring of registers, structures, call chain, locals, global variables and peripheral registers
- Smart STL container display in Watch window
- Symbolic memory window and static watch window
- I/O and interrupt simulation
- True editing-while-debugging
- Drag and drop model
- Target access to host file system via file I/O
- Built-in µC/OS-II Kernel Awareness

C-6 C-SPY DEBUGGER AND TARGET SYSTEM SUPPORT

The C-SPY Debugger for the ARM core is available with drivers for the following target systems:

- Simulator
- Emulator (JTAG/SWD)
 - J-Link probe, JTAG and SWD support, connection via USB or TCP/IP server
 - RDI (Remote Debug Interface), such as Abatron BDI1000 & BDI2000, EPI Majic, Ashling Opella, Aiji OpenICE, Signum JTAGjet, ARM Multi-ICE
 - Macraigor JTAG interfaces: Macraigor Raven, Wiggler, mpDemon, usbDemon, usb2Demon and usb2Sprite
 - ST ST-LINK JTAG debug probe

C-7 IAR ASSEMBLER

- A powerful relocating macro assembler with a versatile set of directives and operators
- Built-in C language preprocessor, accepting all C macro definitions

C-8 IAR J-LINK LINKER

- Complete linking, relocation and format generation to produce FLASH/PROMable code
- Flexible commands allowing detailed control of code and data placement
- Optimized linking removing unused code and data
- Direct linking of raw binary images, for instance multimedia files
- Comprehensive cross-reference and dependency memory maps
- Link compatibility with object files and libraries generated by other EABI compliant tools

C-9 IAR LIBRARY AND LIBRARY TOOLS

- All required ISO/ANSI C and C++ libraries and source included
- All low-level routines such as `writechar()` and `readchar()` provided in full source code
- Lightweight runtime library, user-configurable to match the needs of the application; full source included
- Library tools for creating and maintaining library projects, libraries and library modules
- Listings of entry points and symbolic information

C-10 COMPREHENSIVE DOCUMENTATION

- Efficient coding hints for embedded application
- Extensive step-by-step tutorials
- Context sensitive help and hypertext versions of the user documentation available online

C-11 FIRST CLASS TECHNICAL SUPPORT

IAR Systems has a global organization with local presence through branch offices and a worldwide distributor network. Extended, customized technical services are available.

Appendix

D

Bibliography

- Webster John G. *Medical Instrumentation: Application and Design*. Wiley, 4th. edition, 2009.
- Rangayyan Rangaraj M. *Biomedical Signal Analysis*. Wiley-IEEE Press, 1st. edition, 2001.
- Freescale Semiconductor. *K53 Sub-Family Reference Manual: K53P144M100SF2RM*. Rev. 4, 2011.

Appendix



Licensing µC/OS-II

This book contains µC/OS-II in source form for free short-term evaluation, for educational use or for peaceful research. If you plan or intend to use µC/OS-II in a commercial application/product then, you need to contact Micrium to properly license µC/OS-II for its use in your application/product.

We provide all the source code for your convenience and to help you experience µC/OS-II. The fact that the source is provided does not mean that you can use it commercially without paying a licensing fee. Knowledge of the source code may not be used to develop a similar product.

The reader can purchase the TWR-K53N512 controller along with the AFE medical modules (MED-EKG, MED-GLU, MED-SPO2 and MED-BPM) separately from Freescale. The user may use µC/OS-II with the TWR-K53N512 controller and it is not necessary to purchase anything else as long as the initial purchase is used for short-term evaluation or educational purposes. It is necessary to purchase the license when the decision to use µC/OS-II in a design is made, not when the design is ready to go to production.

If you are unsure about whether you need to obtain a license for your application, please contact Micrium and discuss the intended use with a sales representative.

Micrium
1290 Weston Road, Suite 306
Weston, FL 33326

+1 954 217 2036
+1 954 217 2037 (FAX)

E-Mail : sales@micrium.com
Website : www.micrium.com

Appendix E

E

Index

A

adipose 150
adrenal glands 221
air pump controller
 MED-BPM 230
aldosterone 221
alpha cells 149
alveoli 181
angiotensin I 219
angiotensin II 220
angiotensin-converting enzyme 220
angiotensinogen 219
ANSI C 15, 25
aorta 112
aortic arch 216
API 15
APP.C 138, 141, 143–145, 168, 171, 173–175, 197, 200, 202–204, 239, 242–245
APP_BASELINE_UPPER_LIMIT_MAX_VOLTS 209–210
APP_BASELINE_UPPER_LIMIT_MIN_VOLTS 209–210
APP_BLOOD_DETECT_THR_MVOLTS 176
AppCalcResults() 145, 175, 204, 211, 246, 251
AppCalibrateLED_IR() 209
AppCalibrateLED_Red() 209
AppCardiacCycleState 141, 145, 200, 204
app_cfg.h 34
AppCuffCtrl() 250–251
AppLeakTest.Pressure.Init 250
application code 25
application programming interface 15
APP_MAX_BLOOD_PRESSURE_DIA_MMHG 251
APP_MAX_BLOOD_PRESSURE_SYS_MMHG 251
APP_MAX_CUFF_HOLDING_TIME 251
APP_MAX_CUFF_PRESSURE_MMHG 250
APP_MAX_GLUCOSE_LEVEL_MGDL 176
APP_MAX_HEARTRATE_BPM 211, 251
APP_MAX_OXYGEN_SATURATION 211
APP_MIN_BLOOD_PRESSURE_DIA_MMHG 251
APP_MIN_BLOOD_PRESSURE_SYS_MMHG 251
APP_MIN_CUFF_PRESSURE_MMHG 250–251
APP_MIN_GLUCOSE_LEVEL_MGDL 176
APP_MIN_HEARTBEATS_PER_CALC 206

APP_MIN_HEARTRATE_BPM 211, 251
APP_MIN_OXYGEN_SATURATION 211
APP_MIN_PULSE_SLOPE 146
AppPeakRingBufferClr() 209
AppPulsesAtRampDown 251
AppPulsesAtRampUp 250–251
AppPulsesIR 210–211
AppPulsesRed 211
AppResults 211, 251
AppSamplesCtr 209–211, 250
AppSignal1stSpl 145–146
AppSignal2ndSpl 145–146
AppSignal3rdSpl 145–146
AppSignalCuffPressure.Result 250
AppSignalWorking.Result 176
APP_SMA_WIN_SIZE 176
AppSpIsBtwPulsesCtr 145–146
AppTaskDAQ() . 141, 143–144, 171, 173–174, 200, 202–203, 242, 244–245
AppTaskDAQ_DetectBlood() 176
AppTaskDAQ_DetectQRS() 145
AppTaskDAQ_ExecCmd() 144, 174, 203, 245
AppTaskDAQ_Init() 176, 250
AppTaskDAQ_ProcessCmd() . 143–144, 173–174, 202–203, 244–245
AppTaskDAQ_ProcessData() . 143, 145, 173, 175, 202, 204, 207, 244, 246, 249
AppTaskHeartbeat() 141, 145, 200, 204
AppTaskSim() .. 141, 143–144, 171, 173–174, 200, 202–203, 242, 244–245
AppTaskStart() 37, 39, 46
AppTaskUserIF() 141, 143, 171, 173, 200, 202, 242–244
atrioventricular (AV) node 117
atriums 111
autonomic nervous system 216
aVF 123
aVL 123
aVR 122

B

background 12, 15
baroreceptor 216
 reflex 216

Beer-Lambert law 187
beta cells 149
biological electrical potentials 118
biomedical signal analysis 145, 246
blood glucose meter .81, 108, 149, 156–157, 162, 164–166, 169–171, 177
 block diagram 164
 communication via JTAG 165
 dashboard 170
 design 162
blood glucose sensor 157
blood pressure 214–215
 arterial 215
 arterial, measurement 224
blood pressure monitor 81, 108, 213, 226–227, 234–235, 237, 241–242, 246, 252
 block diagram 226
 dashboard 241
 design 226
 setup 235
board support package 25–26, 34, 103, 258
bronchi 181
BSP 103
BSP.C 143, 145, 173, 202, 204, 243–244
bsp.h 34
BSP_ADC.C 143–145, 173–175, 202–204, 244–246
BSP_ADC0_ISR 144, 174, 203, 245
BSP_ADC0_ISR() 144, 174, 204, 245
BSP_ADC1_ISR 143–144, 173–174, 202–203, 244–245
BSP_ADC1_ISR() 144–145, 174–175, 203–204, 244, 246
BSP_Init() 34, 39–40, 45
BSP_LED_Off() 143, 145, 173, 202, 204, 244
BSP_LED_On() 34, 40, 145, 204
BSP_MUX_Select() 209–210
BSP_POX_LED_On() 209–210
BSP_STATUS_CHECK_INTERVAL 143, 173, 202, 243
BSP_StatusRd() 143, 173, 202, 243
bundle of His 117

C

capillaries 181
carbaminohemoglobin 183
cardiovascular diseases 126
carotid sinuses 216
certification 50, 55, 57, 59, 65, 73
Clark-type electrode 157
context switch 17
conventions 17
Cortex-M4 104, 258, 261–262, 267–271, 273, 278–279
 μC/OS-III port 253
countdown 16
counter electrode 161, 174
CPU.H 254–255, 257
cpu.h 35
CPU_A.ASM 258, 269–272
CPU_Init() 39–40, 45

CPU_STK 35
C-SPY Debugger 289
CVD 126

D

DAQ task 141, 143–145, 147, 171, 173–175, 177, 200, 202–206, 212, 242, 244–246, 252
data flow 276
data processing state machine 207, 249
deadlock 17
 prevention 17
deoxyhemoglobin 183
device class 66, 69
device support 287
dextrose 149
diabetes 155–156
diabetes mellitus 155
diaphragm 180
diastole 214
diastolic pressure 213
DO-178B 73
DO-254 74
DO-278B 73
downloading
 IAR Embedded Workbench for ARM 107
 μC/OS-III projects 101
 μC/Probe 106

E

ECG design 127
ECG leads 121
ED-12B 74
Einthoven's law 122
Einthoven's triangle 121
electrical potentials 118
electrocardiogram 111
electrocardiograph 81
electrode 119–121, 123–124, 127–128, 134–136, 161
 anode 159–160
 biopotential 118–119, 124
 cathode 159
 Clark-type 157
 counter 161, 174
 ECG 121
 illustration 136
 reference 159, 161–162
 signals 165, 171
 skin 127
 test strip 161–162
 test strip signal conditioning schematics 163
 working 159, 161, 174, 176
electroenzymatic 157
electrolyte 157
embedded systems 11, 16
EN 50128 74
EN 50129 74

endocrine system	149
error checking	16
Ethernet	14, 23, 27
EUROCAE	74
EvalBoards	103
EWARM	50, 137, 167, 195–196, 237–238
block diagram	283
extinction coefficient	186
F	
FDA 510(k)	65
FDA development guidance	65
FDA guidance	76
FDA premarket approval	65
foreground	12, 15
foreground/background systems	12
Freescale	90, 103
G	
gauges	275, 278
ghrelin	155
glucagon	149
glucono delta-lactone	160
glucose 81, 108, 149–162, 164–166, 169–171, 173, 175–177	
measuring concentration	159
oxidase enzyme	157
oxidation	157
regulation	150, 153
stock solution	161
glycogen	154
graphs	275, 278
GUI	15
H	
heart	111
heart rate monitor ..	107, 111, 127, 129–130, 132, 134, 139–141, 143, 147–148
block diagram	129–130
dashboard	140
heartbeat	118
heartbeat task	141, 145, 147, 200, 204, 212
hemoglobin	181
hormones	149
hydroxides	160
hyperglycaemia	155
hypertension	222
hypoxia	224
I	
IAR assembler	289
IAR Embedded Workbench for ARM	282
downloading	107
IAR J-LINK Linker	289
IAR Library	290
IEC 61508	73
IEC 61511	74
IEC 61513	74
IEC 62061	74
IEC 62304	70, 74
industry and standards bodies	63
infinite loop	37, 40
insulin	149, 153
interrupt	40
Introduction	49
intuitive	15, 18
ISO 13485	72
ISO 14971	72
J	
J-Link	278–279, 289
J-Link Ultra	108, 132, 137, 165–166, 193, 195, 234, 237
J-Link-Lite for ARM Processors	51
JTAG	289
jumper settings	
MED-EKG	135
TWR-K53N512	87–89, 96–97
juxtaglomerular cells	218
K	
kernel awareness debuggers	17
L	
lancing device	156, 160
larynx	181
lead	120
LED	40
LED driver	
MED-PULSE	191
level of concern	67
LIB_DEF.H	104
licensing	293
lipids	149
liver	150
lungs	181
M	
main()	138, 168, 197, 239
malloc()	35
manometer	224
MCU	25–27
mean arterial pressure	215
MED-BGM	50, 81, 108, 163, 165–166
block diagram	164
signal conditioning circuit	163
with TWR-K53N512	166

MED-BPM	50, 81, 108, 229–231, 233–235, 237	
air pump controller	230	
pin-out	236	
sensor signal conditioning	232	
solenoid valve controller	231	
with TWR-K53N512	229	
MED-EKG	50, 81, 107, 129, 131–134	
jumper settings	135	
signal conditioning circuit	132	
with TWR-K53N512	133	
MEDICAL CONNECTOR	87	
Medical Connector	86	
medical connector	51, 131, 134, 163, 165–166, 191, 193, 195, 229, 233–234, 237	
medical device software		
recalls	57	
medical standards	70	
MED-PULSE	50, 81, 108, 191, 193–195, 203	
LED driver	191	
signal conditioning	190	
with TWR-K53N512	194	
meters	278	
mitral	117	
mode	206	
modular and extensible IDE	284	
monosaccharides	153	
multiple tasks		
application with kernel objects	41	
multitasking	13, 15	
N		
nasal passage	180	
numeric indicators	275, 278	
O		
object names	17	
oral cavity	180	
os.h	29, 35, 37	
os_cfg.h	28, 35	
OS_CFG_APP_HOOKS_EN	259	
oscillometric method	224	
os_core.c	28	
OS_CPU.H	253–255, 257	
os_cpu.h	30	
OS_CPU_A.ASM	253, 258, 269–272	
OS_CPU_PendSVHandler()	271	
OSCtxSw()	270	
OSStartHighRdy()	269	
os_cpu_a.asm	30	
OS_CPU_C.C	253, 258–261, 264–268	
OS_CPU_SysTickHandler()	267	
OS_CPU_SysTickInit()	268	
OSIdleTaskHook()	259	
OSInitHook()	259	
OSStatTaskHook()	260	
OSTaskCreateHook()	260	
OSTaskDelHook()	260	
OSTaskStkInit()	261	
OSTaskSwHook()	265	
OSTimeTickHook()	266	
os_cpu_c.c	30	
OS_CPU_PendSVHandler()	271	
OS_CPU_SysTickHandler()	258, 267	
OS_CPU_SysTickInit()	258, 268	
OSCtxSw()	269–272	
os_dbg.c	29	
os_flag.c	29	
OSIdleTaskHook()	258–259	
OSInitHook()	259	
OSIntCtxSw()	269–272	
os_mem.c	29	
os_mutex.c	29	
os_q.c	29	
os_sem.c	29	
OSStartHighRdy()	269–272	
OSStatTaskHook()	260	
os_task.c	29	
OSTaskCreateHook()	258, 260	
OSTaskQPend()	143, 173, 202, 244	
OSTaskQPost()	143–145, 173–175, 202–204, 244–246	
OSTaskReturnHook()	258	
OSTaskSemPost()	145, 204	
OSTaskStkInit()	258, 261, 264	
OSTaskSwHook()	258, 265, 273	
OSTaskDelHook()	258	
os_time.c	29	
OSTimeTickHook()	258, 266	
os_tmr.c	29	
OS_TS_GET()	34	
oxygen saturation	179	
oxyhemoglobin	181	
P		
pancreas	149–150	
parasympathetic	216	
peripherals	11, 25	
pharynx	181	
pin usage		
TWR-K53N512	84–86	
potassium chloride	159	
precordial leads	124	
Primary hypertension	223	
priority		
inheritance	46	
pulse oximeter	81, 179, 185, 187, 190, 192–193, 195, 199, 211	
block diagram	192	
dashboard	199	
design	190	
signal conditioning	190	
pulse oximetry	185	
Purkinje fibers	117	

R

RAAS	218
real-time kernels	13
real-time operating system	15
red blood cells	181
reference electrode	159, 161–162
regulators	63
regulatory environment	63–64, 69
renin	218
renin-angiotensin-aldosterone system	216, 218
respiration	180
response time	12
ROMable	15–16
RTCA	73

S

safety critical	58
safety critical software	
cost	61
safety-critical development standards	73
scalable	15–16
secondary hypertension	223
semaphore	41
sensor signal conditioning	
MED-BPM	232
setting up	107
sim task ...	141, 143–144, 171, 173–174, 200, 202–203, 242, 244
single task application	34
sinoatrial (SA)	117
sliders	275
small intestine	150
software timers	16, 29
solenoid valve controller	
MED-BPM	231
sphygmomanometer	224
stethoscope	224
stock glucose solutions	161
superloops	12
SWD	279, 289
sympathetic	216
SysTick	258, 267–269
systole	214
systolic pressure	213

T

tail chaining	273
task	
signals	18
task message queues	147, 177, 212, 252
task semaphores	147, 212
test strip	156, 161–163, 166, 170, 174, 176
thread	13

time stamps	40
-------------------	----

timeouts	17
tower system ...	9, 50–52, 81, 90–91, 98–99, 107–108, 131–133, 148, 165, 177, 193–194, 212, 234, 252
traceability	60
trachea	181
transimpedance amplifier	162
tricuspid	117
TWR-ELEV	90
TWR-K53N512 ...	50–52, 81, 83, 90, 98, 103, 108, 129, 131–134, 137, 140–141, 163, 165–166, 170–171, 191, 193–195, 199–200, 229, 233, 237, 241
block diagram	82
connecting	109
documentation	110
jumper settings	87–89, 96–97
pin usage	84–86
pinout	93
with MED-BGM	166
with MED-BPM	229
with MED-EKG	133
with MED-PULSE	194
TWR-K53N512-KIT	90, 107
TWR-LCD	99
TWR-PROTO	98
TWR-SER	90

U

UARTs	23
uC-CPU	104
uC-LIB	104
uCOS-III	105
uCOS-III-Ex1-HRM	104
uCOS-III-Ex3-POX	104
uCOS-III-Ex4-BPM	104
UK Defense Standard 00-56 Issue 2	73
unbounded priority inversion	16
universal asynchronous receiver/transmitters	23
user definable hooks	17
user IF task	141, 143, 171, 173, 200, 202, 242–243

V

vasoconstriction	217
venae cavae	112
ventricles	111

W

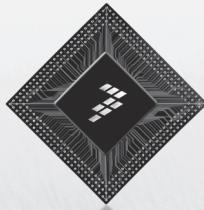
Wilson's central terminal	123
working electrode	159, 161, 174, 176

Z

μC/CPU	104
μC/FS	15
μC/GUI	15

Index

µC/LIB	104
µC/OS-III	15
µC/Probe	17, 29, 275, 278
downloading	106
µC/Probe	
downloading	279
µC/TCP-IP	15, 279
µC/USB	15, 279

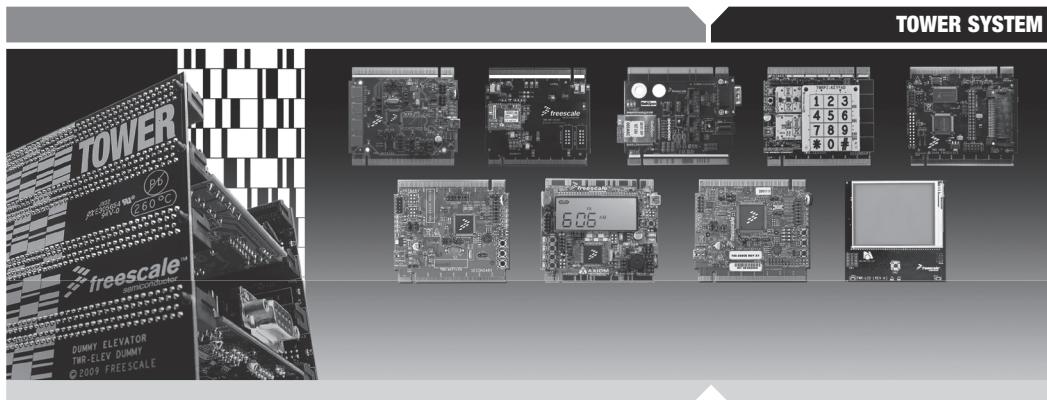


It's embedded design made easy.

At Freescale, we work hard to make the entire design process easier. That's why our MCU solutions offer you one consistent development environment across S08, ColdFire and ARM® architectures. It's why our comprehensive smart mobile device solutions include i.MX processors, sensors and power management components. And it's why our networking multicore designs come to you as production-ready solutions complete with an array of enablement tools and reference designs for quick development. From silicon to software to third-party support, it's easier to design with Freescale. Learn more at freescale.com/discover



Freescale, the Freescale logo and ColdFire are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. ARM is a registered trademark of ARM Limited. All other product or service names are the property of their respective owners. © 2004, 2011 Freescale Semiconductor, Inc.



8-, 16- and 32-bit Microcontrollers/Microprocessors

Freescale Tower System

Modular development platform

Overview

The Freescale Tower System is a modular development platform for 8-, 16- and 32-bit microcontrollers and microprocessors that enables advanced development through rapid prototyping. Featuring multiple development boards or modules, the Tower System provides designers with building blocks for entry-level to advanced microcontroller development.

Modular and Expandable

- Controller modules provide easy-to-use, reconfigurable hardware
- Interchangeable peripheral modules (including serial, memory and graphical LCD) make customization easy
- Open-source hardware and standardized specifications promote the development of additional modules for added functionality and customization

Speeds Development Time

- Open source hardware and software allows quick development with proven designs
- Integrated debugging interface allows for easy programming and run-control via standard USB cable

Cost Effective

- Peripheral modules can be re-used with all Tower System controller modules, eliminating the need to purchase redundant hardware for future designs
- Enabling technologies like LCD, serial and memory interfacing are offered off-the-shelf at a low cost to provide a customized enablement solution

Software Enablement and Support

The increasing complexity of industrial applications and expanding functionality of semiconductors are driving embedded developers toward solutions that require the integration of proven hardware and software platforms. Freescale, along with a strong alliance network, offers comprehensive solutions, including development tools, debuggers, programmers and software.

Complimentary Software and Tools

- Ethernet, FileSystem, USB stacks and more*
- CodeWarrior Development Studio
- Processor Expert software: A rapid application development tool in the CodeWarrior tool suite
- Digital signal processing library: Provides algorithms optimized for the ColdFire architecture

* Visit freescale.com/software for a list of supported devices

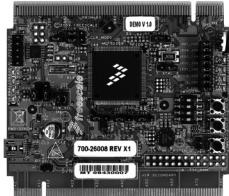
Take Your Design to the Next Level

For a complete list of development kits and modules offered as part of the Freescale Tower System, please visit freescale.com/Tower.

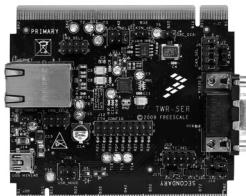
TOWER SYSTEM

Tower System Modules	
Features	Benefits
Controller Modules (8-, 16-, 32-bit)	
• Works stand-alone or as part of Tower System	• Allows rapid prototyping
• Features open source debugging interface	• Provides easy programming and run-control via standard USB cable
Peripheral Modules	
• Can be re-used with all Tower System controller modules	• Eliminates the need to buy/develop redundant hardware
• Interchangeable peripheral modules—serial, memory, graphical LCD, prototyping, sensor	• Enables advanced development and broad functionality
Elevator Boards	
• Two 2x80 connectors	• Provides easy signal access and side-mounting board (i.e. LCD module)
• Power regulation circuitry	• Provides power to all boards
• Standardized signal assignments	• Allows for customized peripheral module development
• Four card-edge connectors available	• Allows easy expansion using PCI Express connectors (x16, 90 mm/3.5" long, 164 pins)

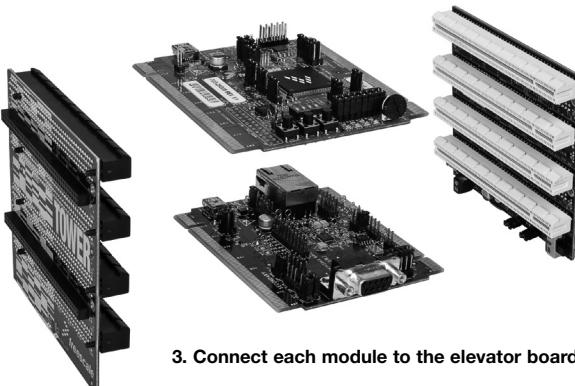
Build Your System in Three Steps or Less



1. Choose a controller module



2. Choose peripheral modules
(up to three standard modules plus side-mounting module(s))



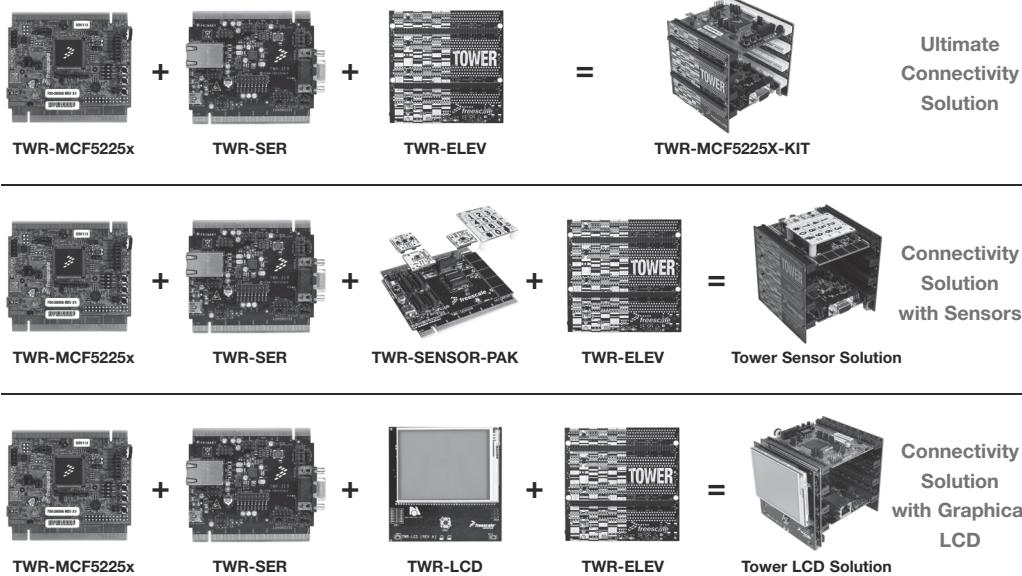
3. Connect each module to the elevator boards

 **freescale**TM
semiconductor

TOWER SYSTEM

Available Tower System Modules	
Controller Modules	Features
• TWR-MCF51CN	MCF51CN ColdFire V1 Ethernet module
• TWR-MCF5225X	MCF5225X ColdFire V2 connectivity module
• TWR-S08LL64	MC9S08LL64 8-bit segment LCD module
• TWR-S08LH64	MC9S08LH64 8-bit segment LCD module with integrated 16-bit ADC
• TWR-MPC5125	MPC5125 e300c4 module built on Power Architecture® technology
• TWR-MCF51MM	MCF51MM ColdFire V1 microcontroller module designed for medical applications
• TWR-S08MM128	MC9S08MM128 8-bit microcontroller module designed for medical applications
• TWR-MCF51JE	MCF51JE ColdFire V1 USB microcontroller module
• TWR-MCF5441X	MCF5441X ColdFire V4 connectivity module with dual Ethernet
• TWR-56F8257	MC56F8257 DSC module
• TWR-K40X256	K40X256 Kinetis module (based on ARM® Cortex™-M4 core) with full-speed USB 2.0 On-The-Go and segment LCD controller
• TWR-K60N512	K60N512 Kinetis module (based on ARM Cortex-M4 core) with IEEE® 1588 Ethernet, full- and High-Speed USB 2.0 On-the-Go, hardware encryption and tamper detection
Peripheral Modules	Features
• TWR-ELEV	Elevator modules: Primary and secondary
• TWR-SER	Serial module with RS232/RS485, Ethernet, CAN, USB
• TWR-SER2	Enhanced serial module featuring dual Ethernet and High-Speed USB
• TWR-PROTO	Prototyping module
• TWR-LCD	Graphical LCD module with 3.2" QVGA display
• TWR-MEM	Memory module with serial flash, MRAM, SD card and compact flash interfaces
• TWR-SENSOR-PAK	Swappable sensor module with accelerometer, barometer and touch-sensing controller
• TWR-WIFI-RS2101	802.11n Wi-Fi® board featuring Redpine Signals' RS9110-N-11-21 Connect-io-n™ Wi-Fi module on board
• TWR-WIFI-G1011MI	802.11b Wi-Fi board featuring GainSpan's GS1011MIP Wi-Fi module on board
• TWR-WIFI-AR4100	802.11n Wi-Fi module featuring the Atheros AR4100 ultra-low power Wi-Fi solution
• TWR-RF-SNAP	Wireless Mesh Networking module featuring SNAP Technology from Synapse Wireless (based on the Freescale MC13224 802.15.4 platform)
• MED-EKG	Sold as part of a complete kit, electrocardiograph sensor for medical applications
• TWR-ADCDAC-LTC	Linear Technology analog module featuring high-precision ADCs and DACs
Complete Kits	Includes
• TWR-MCF51CN-KIT	TWR-MCF51CN, TWR-SER and TWR-ELEV modules
• TWR-MCF5225X-KIT	TWR-MCF5225X, TWR-SER and TWR-ELEV modules
• TWR-S08LL64-KIT	TWR-S08LL64, TWR-PROTO and TWR-ELEV modules
• TWR-S08LH64-KIT	TWR-S08LH64, TWR-PROTO and TWR-ELEV modules
• TWR-MPC5125-KIT	TWR-MPC5125, TWR-SER and TWR-ELEV modules
• TWR-MCF51MM-KIT	TWR-MCF51MM, TWR-SER, TWR-ELEV and MED-EKG modules
• TWR-S08MM128-KIT	TWR-S08MM128, TWR-SER, TWR-ELEV and MED-EKG modules
• TWR-MCF51JE-KIT	TWR-MCF51JE-KIT, TWR-SER and TWR-ELEV modules
• TWR-MCF5441X-KIT	TWR-MCF5441X, TWR-SER2 and TWR-ELEV modules
• TWR-K40X256-KIT	TWR-K40X256, TWR-SER and TWR-ELEV modules
• TWR-K60N512-KIT	TWR-K60N512, TWR-SER and TWR-ELEV modules
• TWR-K60N512-IAR	TWR-K60N512-KIT, TWR-PROTO, IAR J-Link (Lite) Debug Probe

TOWER SYSTEM



Example Configurations Partner Modules

Tap into a powerful ecosystem of Freescale technology alliances for building smarter, better connected solutions. Designed to help you shorten your design cycle and get your products to market faster, these technology alliances provide you with access to rich design tools, peripherals and world-class support and training.

A number of partners have developed modules for the Tower System. Some examples include the i.MX515 ARM® Cortex™-A8 Tower Computer Module and StackableUSB™ I/O Device

Carrier module from Micro/sys, as well as the Rapid Prototyping System (RPS) AM1 and FM1 modules from iMN MicroControl.

A complete list of partner-developed modules is available at freescale.com/Tower.

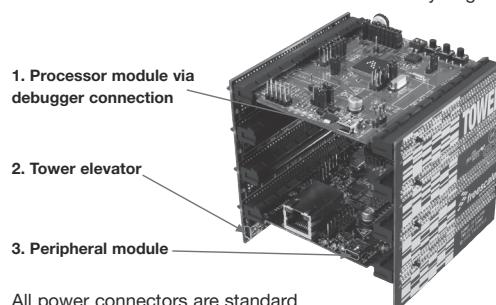
Multiple Power Options

The Freescale Tower System can be powered entirely over a USB cable via a host PC or USB wall power adaptor. Alternatively, power can be supplied to the Tower via a screw terminal on the Primary Elevator.

Protection circuitry is built into all Tower System modules to avoid contention on the power rails. Although power can be supplied through any module, power supplied through the elevator modules takes precedence.

Tower Geeks Online Community

TowerGeeks.org is an online design engineer community that allows members to interact, develop designs and share ideas. Offering a direct path to explore and interact with other engineers designing with the Tower System, TowerGeeks.org is a great way to discuss your projects, post videos of your progress, ask questions through the forum and upload software. With updates through Twitter and Facebook, it's easy to get involved.



All power connectors are standard USB connectors that can be powered by a USB host/hub or an AC-to-DC adapter with a USB cable.

 **freescale**™
semiconductor

TOWER SYSTEM

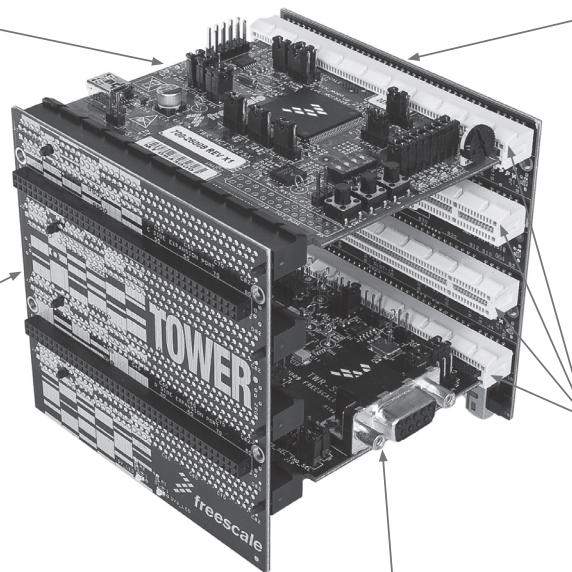
The Freescale Tower System

Controller Module

- Tower MCU/MPU board
- Works stand-alone or in Tower System
- Features integrated debugging interface for easy programming and run control via standard USB cable

Secondary Elevator

- Additional and secondary serial and expansion bus signals
- Standardized signal assignments
- Mounting holes and expansion connectors for side-mounting peripheral boards



Size

- Tower is approx. 3.5" H x 3.5" W x 3.5" D when fully assembled

Primary Elevator

- Common serial and expansion bus signals
- Two 2x80 connectors on backside for easy signal access and side-mounting board (LCD module)
- Power regulation circuitry
- Standardized signal assignments
- Mounting holes

Board Connectors

- Four card-edge connectors
- Uses PCI Express® connectors (x16, 90 mm/3.5" long, 164 pins)

Peripheral Module

- Examples include serial interface module, memory expansion module and Wi-Fi®

Tower Geeks Online Community

TowerGeeks.org is an online design engineer community that allows members to interact, develop designs and share ideas. Offering a direct path to explore and interact with other engineers designing with the Tower System, **TowerGeeks.org** is a great way to discuss your projects, post videos of your progress, ask questions through the forum and upload software. With updates through **Twitter** and **Facebook**, it's easy to get involved.



Follow Tower Geeks on Twitter
twitter.com/towergeeks



Visit Freescale on Facebook
facebook.com/freescale

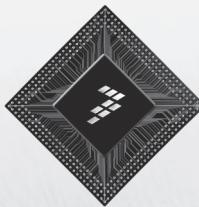
Learn More:

For more information about the Freescale Tower System, please visit freescale.com/Tower.

Freescale, the Freescale logo, CodeWarrior, ColdFire and Processor Expert are trademarks or registered trademarks of Freescale Semiconductor, Inc. Reg. U.S. Pat. & Tm. Off. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. ARM is a registered trademark of ARM Limited. Cortex-A8 and Cortex M-4 are trademarks of ARM Limited. All other product or service names are the property of their respective owners.
© 2010, 2011 Freescale Semiconductor, Inc.

Document Number: TWRFS / REV 7

 **freescale™**
semiconductor



It's design potential realized.

Freescale introduces the industry's most scalable 32-bit MCU portfolio—Kinetis MCUs based on ARM® Cortex™-M4 technology. This portfolio expands your 32-bit choices with over 200 pin- and software-compatible MCUs. Kinetis MCUs are also supported by a market-leading enablement bundle, including the CodeWarrior IDE, Tower development systems and a huge community of ARM third-party ecosystem partners. Get started today with a complete package that is key to your design success and a faster time to market. Learn more at freescale.com/realize



Freescale, the Freescale logo and CodeWarrior are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Kinetis is a trademark of Freescale Semiconductor, Inc. ARM is a registered trademark of ARM Limited. Cortex-M4 is a trademark of ARM Limited. All other product or service names are the property of their respective owners. © 2004, 2011 Freescale Semiconductor, Inc.

Your project deserves world-class embedded development tools!

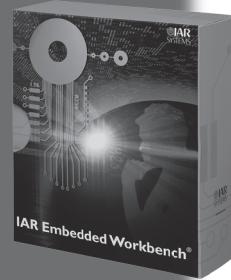
We are the world's leading supplier of software tools for embedded systems that enable companies to develop premium products based on 8-, 16-, and 32-bit microcontrollers. You find our customers mainly in the areas of industrial automation, medical devices, consumer electronics and automotive products. We have an extensive network of partners and cooperate with the world's leading semiconductor vendors.

Your project deserves our tools.

Read more at www.iar.com.

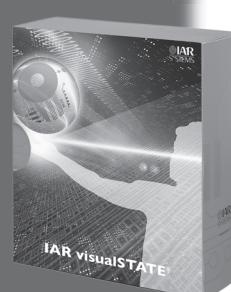
Optimize Your Application with IAR Embedded Workbench!

IAR Embedded Workbench is a set of development tools for building and debugging embedded applications using assembler, C and C++. It provides a completely integrated development environment including a project manager, editor, build tools and debugger. In a continuous workflow, you can create source files and projects, build applications and debug them in a simulator or on hardware.



Design Your Project with IAR visualSTATE!

IAR visualSTATE is a set of highly sophisticated and easy-to-use development tools for designing, testing and implementing embedded applications based on state machines. It provides advanced verification and validation utilities and generates very compact C/C++ code that is 100% consistent with your system design.



KickStart Your Application with IAR KickStart Kit!

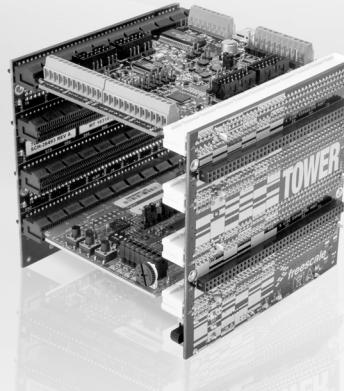
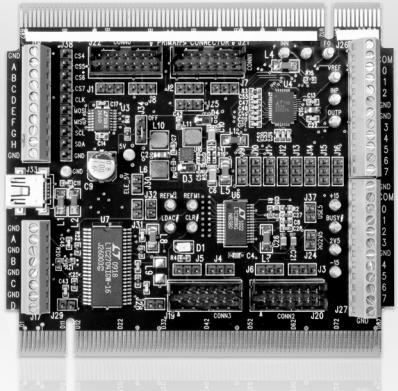
Development kits from IAR Systems provide you with all the tools you need to develop embedded applications right out of the box.

Get access to our world- class support!

We have local presence through branch offices and a worldwide distributor network. We also offer extended, customized technical services.



Linear Technology Analog Playground Module for the Freescale Tower System



Quickly evaluate Linear Technology data converters and other mixed signal solutions with the Freescale Tower System. The easy-to-use plug-in analog module (TWR-ADCDAC-LTC) expands the capabilities of the Freescale Tower System. It's a complete solution with a high precision analog peripheral module controllable by any Freescale Tower processor module with an SPI interface. The QuikEval™ interface on the analog module allows connection of more than 130 Linear Technology evaluation boards with the Freescale Tower processor for a broad range of applications.

TWR-ADCDAC-LTC Features

- Digital-to-Analog Converters (DAs)
 - LTC®2704-16: Quad 16-Bit V_{OUT} SoftSpan™ DAC with Readback
 - LTC2600: Octal 16-Bit Rail-to-Rail DACs
- Analog-to-Digital Converters (ADCs)
 - LTC1859: 8-Channel, 16-Bit, 100ksps SoftSpan ADC with Shutdown
 - LTC2498: 24-Bit 8-/16-Channel $\Delta\Sigma$ ADC with Easy Drive™ Input Current Cancellation
- Voltage Regulator
 - LTC3471: Dual 1.3A, 1.2MHz Boost/Inverter
- Voltage Reference
 - LTC6655-5: 0.25ppm Noise, Low Drift Precision Buffered 5V Reference
- Four 14-Pin Headers for Connecting to any Linear Technology QuikEval Demonstration Boards
- Demos/Applications Include:



ADC Data Logger/DAC Waveform Generator



Thermocouple Reader

Applications

- Data Acquisition
- Instrumentation
- Temperature Measurement
- Industrial Process
- Medical
- Weight Scales



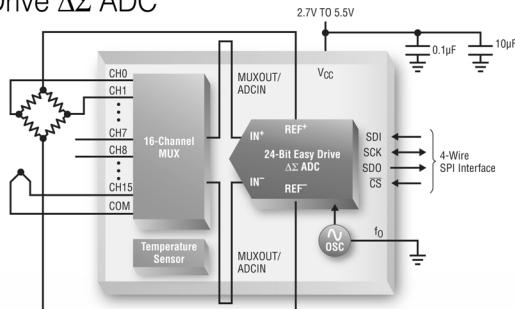
© LT, LTC, LTm, Linear Technology and the Linear logo are registered trademarks and QuikEval, SoftSpan and Easy Drive are trademarks of Linear Technology Corporation. All other trademarks are the property of their respective owners.



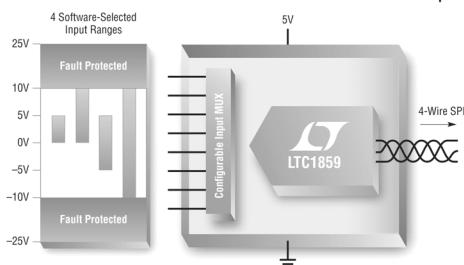
LTC2498: 24-Bit, 16-Channel Easy Drive $\Delta\Sigma$ ADC

Features

- 8 Differential/16 Single-Ended Input Channels
- Easy Drive Technology Enables Rail-to-Rail Inputs with Zero Differential Current
- Directly Digitizes High Impedance Sensors with Full Accuracy
- 600nVRMS Noise
- Internal Temperature Sensor (2°C Maximum), Internal Oscillator
- Selectable 50Hz, 60Hz Rejection, Up to 15Hz Output Rate

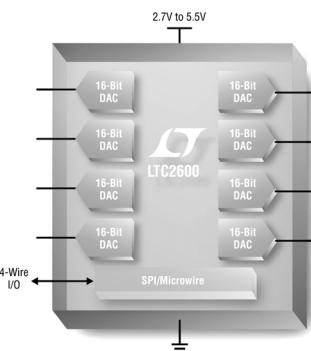


LTC1859: 8-Channel, 16-Bit, 100ksps SoftSpan A/D Converter with Shutdown



Features

- 8-Channel Multiplexer with $\pm 25V$ Protection
- Software-Programmable Input Ranges: 0V to 5V, 0V to 10V, $\pm 5V$ or $\pm 10V$, Single-Ended or Differential
- Power Dissipation: 40mW (Typ)
- SPI/MICROWIRE Compatible Serial I/O
- Signal-to-Noise Ratio: 87dB (Typ)

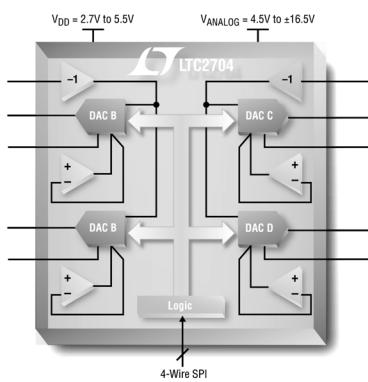


LTC2600: Octal 16-Bit Rail-to-Rail DACs

Features

- Guaranteed 16-Bit Monotonic Over Temperature
- Low Power Operation: 250μA per DAC at 3V
- Individual Channel Power-Down to 1μA, Max
- Ultralow Crosstalk Between DACs (<10μV)
- High Rail-to-Rail Output Drive ($\pm 16mA$, Min)
- Double-Buffered Digital Inputs

Quad 12-, 14- and 16-Bit Voltage Output SoftSpan DACs with Readback



Features

- Six Programmable Output Ranges
- Unipolar: 0V to 5V, 0V to 10V
- Bipolar: $\pm 5V$, $\pm 10V$, $\pm 2.5V$, $-2.5V$ to $7.5V$
- Serial Readback of All On-Chip Registers
- 1LSB INL and DNL Over the Industrial Temperature Range (LTC2704-14/LTC2704-12)
- Force/Sense Outputs Enable Remote Sensing
- Glitch Impulse: < 2nV per Second
- Outputs Drive $\pm 5mA$



Off-the-shelf certification
evidence for µC/OS on
ARM, PowerPC, and Coldfire

Got a Certification Headache?

We've got the cure ☺

Our µC/OS Validation products are ready to provide immediate relief when certification evidence is needed.

Validated Software delivers µC/OS certification evidence running and tested with your hardware and tools. No stress µC/OS certification support saves you time and money and lets you focus on the important things



All-in-one µC/OS solutions for
IEC 62304, FDA 510(k)/PMA, ISO 13485, ISO 14971

Validation Suites™ are commercial-off-the-shelf and customized to your requirements. Delivered 100% complete, submission ready and running on your target hardware. Available in weeks, not months.

Validation Kits™ are RTOS, middleware, market, and standard ready. For those who prefer to test at their own facility, Validation Kits provide everything needed to certify the RTOS or middleware components.

Validation Templates™ are for those new to the certification process looking to implement a proven, practical, and affordable approach to device certification.

Validated Software Corporation • 6848 Embarcadero Lane • Carlsbad, CA 92011
Phone: 760-230-5298 • email: sales@ValidatedSoftware.com
www.validatedsoftware.com