# Embedded SOFTWARE

## *The Works*

CD-ROM INCLUDED containing

- Source code
- PowerPoint® slides
- Spec sheets

COLIN WALLS

Newnes

# Embedded Software: The Works

# Embedded Software: The Works

*Colin Walls*

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Newnes is an imprint of Elsevier

Newnes is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
525 B Street, Suite 1900, San Diego, California 92101-4495, USA
84 Theobald's Road, London WC1X 8RR, UK


This book is printed on acid-free paper.



**Library of Congress Cataloging-in-Publication Data**

Walls, Colin.
Embedded software : the works / Colin Walls.
p. cm.
Includes bibliographical references and index.
ISBN 0-7506-7954-9 (alk. paper)
1. Embedded computer systems–Programming. I. Title.
TK7895.E42W35 2005
005.1–dc22

2005014209

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

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

Printed in the United States of America
05 06 07 08 09 9 8 7 6 5 4 3 2 1

# *Dedication*

To Blood Donors Everywhere
Your generosity saves lives every day.
Thank you.

# Contents

# Foreword

## What Do You Expect—Perfection?

*A few words from Jack Ganssle . . .*

Why are so many firmware projects so late and so bug-ridden? A lot of theories abound about software's complexity and other contributing factors, but I believe the proximate cause is that coding is not something suited to Homo Sapiens. It requires a level of accuracy that is truly super-human. And most of us are not super-human.

Cavemen did not have to get every gazelle they hunted—just enough to keep from starving. Farmers never expect the entire bag of seeds to sprout; a certain wastage is implicit and accepted. Any merchant providing a service expects to delight most, but not all, customers.

The kid who brings home straight A grades thrills his parents. Yet we get an A for being 90% correct. Perfection isn't required. Most endeavors in life succeed if we score an A, if we miss our mark by 10% or less.

Except in software. 90% correct is an utter disaster, resulting in an unusable product. 99.9% correct means we're shipping junk. One-hundred K lines of code with 99.9% accuracy suggests some 100 lurking errors. That's not good enough. Software requires near perfection, which defies the nature of intrinsically error-prone people.

Software is also highly entropic. Anyone can write a perfect 100 line-of-code system, but, as the size soars, perfection, or near perfection, requires ever-increasing investments of energy. It's as if the bits are wandering cattle trying to bust free from the corral; coding cowboys work harder and harder to avoid strays as the size of the herd grows.

So what's the solution? Is there an answer? How good does firmware have to be? How good can it be? Is our search for perfection or near-perfection an exercise in futility?

Complex systems are a new thing in this world. Many of us remember the early transistor radios which sported a half dozen active devices, max. Vacuum tube televisions, common into the 1970s, used 15 to 20 tubes, more or less equivalent to about the same number of transistors. The 1940s-era ENIAC computer required 18,000 tubes—so many that technicians wheeled shopping carts of spares through the room, constantly replacing those that burned out. Though that sounds like a lot of active

elements, even the 25-year-old Z80 chip used a quarter of that many transistors, in a die smaller than just one of the hundreds of thousands of resistors in the ENIAC.

Now the Pentium IV, merely one component of a computer, has 45 million transistors. A big memory chip might require one-third of a billion. Intel predicts that later this decade their processors will have a billion transistors. The very simplest of embedded systems, like an electronic greeting card, requires thousands of active elements.

Software has grown even faster, especially in embedded applications. In 1975, 10,000 lines of assembly code was considered huge. Given the development tools of the day—paper tape, cassettes for mass storage, and crude teletypes for consoles—working on projects of this size was very difficult. Today 10,000 lines of C, representing perhaps three to five times as much assembly, is a small program. A cell phone might contain 5 million lines of C or C++, astonishing considering the device's small form factor, miniscule power requirements, and breathtakingly short development times.

Another measure of software size is memory usage. The 256 byte [that's not a typo] EPROMs of 1975 meant even a measly 4k program used 16 devices. Clearly, even small embedded systems were quite pricey. Today? 128k of Flash is nothing, even for a tiny app. The switch from 8 to 16 bit processors, and then from 16 to 32 bitters, is driven more by addressing space requirements than raw horsepower.

In the late 1970s Seagate introduced the first small Winchester hard disk, a 5Mb 10-pound beauty that cost $1500. Five MB was more disk space than almost anyone needed. Now 20Gb fits into a shirt pocket, is almost free, and fills in the blink of an eye.

So, our systems are growing rapidly in both size and complexity. Are we smart enough to build these huge applications correctly? It's hard to make even a simple application perfect; big ones will possibly never be faultless. As the software grows it inevitably becomes more intertwined; a change in one area impacts other sections, often profoundly. Sometimes this is due to poor design; often, it's a necessary effect of system growth.

The hardware, too, is certainly a long way from perfect. Even mature processors usually come with an errata sheet, one that can rival the datasheet in size. The infamous Pentium divide bug was just one of many bugs. Even today, the Pentium 3's errata sheet [renamed "specification update"] contains 83 issues. Motorola documents nearly a hundred problems in the MPC555.

What is the current state of the reliability of embedded systems? No one knows. It's an area devoid of research. Yet a lot of raw data is available, some of which suggests we're not doing well.

The Mars Pathfinder mission succeeded beyond anyone's dreams, despite a significant error that crashed the software during the lander's descent. A priority inversion problem—noticed on Earth but attributed to a glitch and ignored—caused numerous crashes. A well-designed watchdog timer recovery strategy saved the mission. This was a

very instructive failure as it shows the importance of adding external hardware and/or software to deal with unanticipated software errors.

It's clear that the hardware and software are growing in raw size as well as complexity. Pathfinder's priority inversion problem was unheard of in the early days of microprocessors, when the applications just didn't need an RTOS. Today most embedded systems have some sort of operating system, and so are at peril for these sorts of problems.

What are we as developers to do, to cope with the explosion of complexity? Clearly the first step is a lifelong devotion to learning new things. And learning old things. And even re-learning old things we've forgotten. To that end we simply must curl up in the evenings with books such as this, this compendium of the old and the new, from the essentials of C programming to UML and more. Colin is an old hand, who here shares his experience with plenty of practical "how to do this" advice.

We do learn well from experience. But that's the most expensive way to accumulate knowledge. Much better is to take the wisdom of a virtuoso, to, at least for the time it takes to read this book, apprentice oneself to a master.

You'll come away enriched, with new insights and abilities to solve more complex problems with a wider array of tools.

Perfection may elude us, but wise developers will spend their entire careers engaged in the search.

# *Preface*

## How This Book Came About

I wonder how many people formulate a plan for their lives and then follow through with it. Some do, I'm sure, but for most of us, even if we have ambitions and ideas for the future, we are often buffeted by the random events around us. And that is how it was for me. I work for Accelerated Technology, a division of Mentor Graphics, which is dedicated to providing embedded software design and development tools and operating systems. I have done a variety of jobs within the company over the years, throughout a number of acquisitions and name changes. Of late, I have been doing outbound marketing—traveling around Europe and North America getting excited about our products and the technology behind them. I often describe my job as "professional enthusiasm." It is a great job, and I work with great people. I enjoy it. But it is also very taxing, with a lot of time spent on the road (or in the air) or just waiting for something to happen. I tend to be away from home quite a lot, but I have always accepted that this price is one I have to pay for such a rewarding job. And then it all changed. . . .

In mid-summer 2004, my wife was suddenly diagnosed with acute myeloid leukemia and immediately admitted to hospital. This meant that my working life was suddenly put on hold. I needed to be home to support her and care for our two teenage daughters. Her treatment—aggressive chemotherapy—progressed through the rest of the year. There have been many ups and downs, but the outcome, at the time of writing at least, is that she is in remission, having monthly blood checks to ensure that this progress continues.

It is under these circumstances that you find out who your friends are—an old cliché but true nevertheless. In my case, I also found out that I was working for the right company. My management and colleagues could not have been more supportive, and I am indebted to all of them. (I'll name names later.) Clearly I could not continue with my usual job, for a while at least. I was put under no pressure, but I gave careful thought as to what I might do that was useful, but would be compatible with my new lifestyle. I had been harboring the idea of writing this book for a while; I suggested the idea to my management, who readily agreed that it would be a worthwhile project.

## Where This Book Came From

It is nearly 20 years since I last wrote a book (*Programming Dedicated Microprocessors,* Macmillan Education, 1986). That book was about embedded software, even though the term was not in common use at that time. Writing that book was a realization of an ambition, and the couple of copies that I still have are among my most treasured possessions. Writing a book is a time-consuming activity. Back when I wrote my first book, my job was relatively undemanding, and we had no children. Finding the time then was not so difficult. I have long since wanted to write another book and have had various ideas for one, but spare time has been in short supply.

Nevertheless, I have still been writing over the years—lots of technical articles, conference papers, lessons for training classes, and presentations. Then I had a thought: why not gather a whole bunch of this writing together in a book? To simplify the process, I concentrated my search for material within Accelerated Technology because the company owns the copyrights. A few other pieces did eventually get donated, but that was later.

At Microtec Research, *NewBits*, a quarterly newsletter, evolved from a simple newsletter into a technical journal, and I was a regular contributor from the early 1990s. *NewBits* continued to be published after the acquisition of Microtec by Mentor Graphics, but the journal was eventually phased out. Recently, after revitalizing the embedded software team by acquiring Accelerated Technology, we decided to revive *NewBits*, and it has gone from strength to strength. Since I collected every issue of *NewBits*, I had all of the research material at my fingertips. I quickly determined that many of my articles and those by several other writers would be appropriate for inclusion in this book. Other sources of material were white papers and another Accelerated Technology newsletter called *Nucleus Reactor*.

I have endeavored to make as few alterations as possible to the articles. Some required almost no changes; others needed a little updating or the removal of product-specific information. Each article includes an introductory paragraph that describes the origins of the article and puts it into context.

## What You Will Find Here

In selecting the material for this book, my goal was to ensure that every article was relevant to current embedded software development practice and technology. Of course, many pieces have a historical perspective, but that alone was not sufficient qualification for their inclusion. Quite a few articles were rejected because they covered a technology that did not catch on or has since been superseded by something else. Maybe those topics will appear when I write *A History of Embedded Software*, which I slate for publication on the fiftieth anniversary of the start of it all (in 2020—50 years after the Intel 4004 was announced).


In this book, you will find a selection of articles covering just about every facet of embedded software: its design, development, management, debugging procedures, licensing, and reuse.

## Who This Book Is For

If you are interested in embedded software, this book includes something for you. The articles cover a wide range, and hence, it is of interest to newcomers to the field as well

as old hands. If your background is in more conventional software, some pieces will give you a perspective on the "close to the hardware" stuff. If your background is in hardware design, you may gain some understanding of "the other side."

If you teach—either in a commercial or academic context—you may find some of the articles provide useful background material for your students. Some of the CD-ROM content is designed with just you in mind. See the section "What's on the CD-ROM?" following this preface.

## How to Use This Book

There is no "right" way to use this book. I have attempted to sequence the articles so that reading the book from front to back would make sense. I have also done my best to categorize the articles across the chapters to help you find what might be of particular interest, with a few cross-references where I felt they might be helpful. I am frustrated by inadequate indexes in reference books, so I have tried to make the index in this book an effective means of finding what interests you.

## Acknowledgments

It is difficult to write this section without it sounding like an acceptance speech for an Academy Award.

Before naming individuals, I want to make a more general acknowledgment. I have spoken to many colleagues within Accelerated Technology and in the wider world of Mentor Graphics about this book project. The response has been universal encouragement and enthusiasm. This helped.

When I started working with Elsevier, my editor was Carol Lewis, who started the project. In due course, the baton was passed to Tiffany Gasbarrini, who helped me see it through to conclusion. I enjoyed working with both of them, admired their professionalism, and appreciated their guidance.

I have always enjoyed reading the work of Jack Ganssle—his books and regular columns in *Embedded Systems Programming* magazine—who manages to convey useful technical information in a thought-provoking and, more often than not, humorous way. So, when I started planning this book, I sought his advice, which he enthusiastically provided. I was also delighted when he agreed to provide the Foreword. Thanks Jack.

I have a debt of gratitude to my management and colleagues, who have stood by me and gone that extra mile at a difficult time. Apart from (perhaps) maintaining some of my sanity, they were instrumental in making this book possible. To name a few: Neil Henderson, Robert Day, Michelle Hale, Gordon Cameron, Joakim Hedenstedt. There are many others. Thanks guys.

As I have mentioned, a great deal of the material in this book had its origins in the Microtec newsletter *NewBits*. So, I am indebted to all the folks who were involved in its

publication over the years. Lucille Woo [née Ching] was the managing editor of *NewBits*, developing it from a simple newsletter into a solid technical journal, with graphics and design work by Gianfranco Paolozzi. At Microtec, various people looked after the journal over the years, including Eugene Castillo, Melanie Gill, and Rob van Blommestein. The "revival" of *NewBits* at Accelerated Technology was led by Charity Mason, who had a hard act to follow but rose to the challenge admirably.

I am allergic to one facet of business: anything with the word "legal" involved. I accept the necessity for all the procedures, but I am grateful that people like Jodi Charter in Mentor Graphics Procurement can take care of the contract processing for me.

I was pleased by the prompt and positive response I got from David Vornholt at Xilinx and Bob Garrett at Altera when I sought solid background material on FPGA processors.

Last, and by no means least, I would like to thank Archie's proud parents, Andrea and Barry Byford, for their permission to use his photographs in the Afterword.

## Contributors

I wrote about half the words in this book. The rest were contributed cooperatively or unwittingly by these individuals: Zeeshan Altaf, Fakhir Ansari, Antonio Bigazzi, Sarah Bigazzi, Paul Carrol, Lily Chang, Robert Day, Michael Eager, Michael Fay, Jack Ganssle, Bob Garrett, Kevin George, Ken Greenberg, Donald Grimes, Larry Hardin, Neil Henderson, C.C. Hung, Meador Inge, Stephen Mellor, Glen Johnson, Pravat Lall, Tammy Leino, Nick Lethaby, Steven Lewis, Alasdar Mullarney, Stephen Olsen, Doug Phillips, Uriah Pollock, James Ready, John Schneider, Robin Smith, Dan Schiro, Richard Vlamynck, David Vornholt, Fu-Hwa Wang, John Wolfe.

I would like to thank every one of them for their contributions and apologize in advance if I managed to forget attributing anyone.

It is easy to think of a book as just a bunch of words between two covers. But you will find that many articles here, as in most technical publications, include illustrations that help reinforce the message. I am happy with writing words, but I am no artist. So I am grateful for the help I received from Chase Matthews and Dan Testen, who provided the necessary artistic talent.

## A Good Cause

I was delighted when my management generously agreed that all proceeds from this book could be donated to a charitable organization of my choice. Naturally, I thought about my circumstances and decided to support a group that produced tangible results. I chose the LINC Fund (Leukaemia and Intensive Chemotherapy—www.lincfund.co.uk), which is based at the center where my wife was treated.

LINC is a charitable organization dedicated to the support and well-being of patients with leukemia and related conditions. Money collected by LINC is used to purchase equipment and aid research at Cheltenham General Hospital Oncology Unit. Since the onset of these conditions can be very sudden, with patients embarking on lengthy treatment programs with little or no notice, some people find themselves in financial difficulties. The LINC Fund also provides help to such families at a time when they need it most.

On their behalf, thank you for your contribution, which I know will be spent wisely.

## Contact Me

If you have comments or questions about this book, or indeed anything to do with embedded software, I would be very pleased to hear from you. Email is the best way to reach me:

`colin_walls@mentor.com`

If you wish to reach other contributors, please email me, and I will endeavor to put you in contact with them.

To see the latest information about this book—updates, errata, downloads—please visit www.EmbeddedSoftwareWorks.com.

Colin Walls

# What's on the CD-ROM?

*To provide as much value as possible, we decided to include a CD of supplementary material with the book. Here is a guide to the contents.*

---

## Code Fragments

Many of the articles include listings of C or C++ code. These are all provided on the CD so that you can use them without needless retyping. They have all been carefully checked, so they should build and run as expected.

The `code` folder contains subfolders for each chapter. In each of these subfolders are plain text files (with the extension `.txt`) that pertain to relevant articles. You can simply copy and paste the code as you want. There are 23 files:

**Chapter 1: Embedded Software**
Migrating Your Software to a New Processor Architecture

**Chapter 2: Design and Development**
Embedded Software and UML

**Chapter 3: Programming**
Programming for Exotic Memories
Self-Testing in Embedded Systems
A Command-Line Interpreter
Traffic Lights: An Embedded Software Application

**Chapter 4: C Language**
Interrupt Functions and ANSI Keywords
Optimization for RISC Architectures
Bit by Bit
Programming Floating-Point Applications
Looking at C—A Different Perspective
Structure Layout—Become an Expert

**Chapter 5: C++**
Why Convert from C to C++?
Clearing the Path to C++
C++ Templates—Benefits and Pitfalls
Exception Handling in C++
Looking at Code Size and Performance with C++
Write-Only Ports in C++
Using Nonvolatile RAM with C++

**Chapter 6: Real Time**
Programming for Interrupts

**Chapter 7: Real-Time Operating Systems**
A Debugging Solution for a Custom Real-Time Operating System
Introduction to RTOS Driver Development

**Chapter 8: Networking**
Who Needs a Web Server?

### *Training Materials*

The subjects of many of the articles lend themselves to training classes or seminars. Indeed, many of my articles originated as pieces for training classes or seminars. Where possible, I have created a slide set to accompany an article (but obviously I was unable to do so for every article).

The `training` folder contains subfolders for each chapter. Each of these subfolders contain files pertaining to relevant articles. These files are in pairs. One is a Microsoft PowerPoint `(.PPT)` file. To use this file effectively, you need a license for PowerPoint. You can then not only present the slides, but modify and customize them. The other file is an Adobe PDF file. To view this file, you need to obtain the free Adobe Reader software. If you simply open the PDF, it will display full screen (there's no need to press `CTRL/L`), and you can make the presentation (press `Esc` to finish). Adobe Reader is an effective presentation tool but does not facilitate making changes to the PDF files.

All the slide sets are designed to be coherent sequences, following the theme of the relevant article, but you may also find individual slides are useful for illustrating points in other contexts. There are over 400 slides, in 46 pairs of files:

**Chapter 1: Embedded Software**
What Makes an Embedded Application Tick?
Memory in Embedded Systems
Memory Architectures

How Software Influences Hardware Design
Migrating Your Software to a New Processor Architecture
Embedded Software for Transportation Applications
An Introduction to USB Software

**Chapter 2: Design and Development**
Emerging Technology for Embedded Systems Software Development
Making Development Tool Choices
Embedded Software and UML

**Chapter 3: Programming**
Programming for Exotic Memories
Self-Testing in Embedded Systems
A Command-Line Interpreter
Traffic Lights: An Embedded Software Application

**Chapter 4: C Language**
C Common
Interrupt Functions and ANSI Keywords
Bit by Bit
Looking at C—A Different Perspective
Reducing Function Call Overhead
Memory and Programming in C
Pointers and Arrays in C and C++

**Chapter 5: C++**
C++ in Embedded Systems: A Management Perspective
Why Convert from C to C++?
Clearing the Path to C++
C++ Templates – Benefits and Pitfalls
Exception Handling in C++
Looking at Code Size and Performance with C++
Write-Only Ports in C++
Using Nonvolatile RAM with C++

**Chapter 6: Real-Time**
Real-Time Systems
Visualizing Program Models of Embedded Systems
Programming for Interrupts

**Chapter 7: Real-Time Operating Systems**
Debugging Techniques with an RTOS
A Debugging Solution for a Custom Real-Time Operating System
Bring in the ProsWhen to Consider a Commercial RTOS
On The Mover
Scheduling Algorithms and Priority Inversion
OSEK—An RTOS Standard

**Chapter 8: Networking**
What's Wi-Fi?
Who Needs a Web Server?
Introduction to SNMP
IPv6—The Next Generation Internet Protocol
The Basics of DHCP
NAT Explained
PPP—Point-to-Point Protocol

**Chapter 9: Embedded Systems and Programmable Logic**
FPGAs and Processor Cores—The Future of Embedded Systems?

### *Product Information*

For the most part, I have been very careful to ensure that the material in this book is not specific to particular commercial products. However, my employers have been very generous and their "reward" is to include some promotional material on this CD. A variety of materials may be found in the `AcceleratedTechnology` folder. In here you will find a selection of data sheets as PDF files. There is also an interactive presentation; PC users can just run `AcceleratedTechnology.exe;` there is a file `AcceleratedTechnology.html` which runs the presentation within a web browser.

# 1

# *Embedded Software*

*This first collection of articles either set the scene, providing a broad view of what embedded software is all about, or address a specific area that is not really encompassed by another chapter.*

## 1.1 What Makes an Embedded Application Tick?

*This is very much a "setting the scene" article, based upon one that I wrote for* NewBits *in late 2003 and a talk that I have delivered at numerous seminars. It introduces many embedded software issues and concepts that are covered in more detail elsewhere in this book. (CW)*

Embedded systems are everywhere. You cannot get away from them. In the average American household, there are around 40 microprocessors, not counting PCs (which contribute another 5–10 each) or cars (which typically contain a few dozen). And these numbers are predicted to rise by a couple of orders of magnitude over the next decade or two. It is rather ironic that most people outside of the electronics business have no idea what "embedded" actually means.

Marketing people are fond of segmenting markets. The theory is that such segmentation analysis will yield better products by fulfilling the requirements of each segment in a specific way. For embedded, we end up with segments like telecom, mil/aero, process control, consumer and automotive. Increasingly though, devices come along that do not fit this model. For example, is a cell phone with a camera a telecom or consumer product? Who cares? An interesting area of consideration is the commonality of such applications. The major comment that we can make about them all is the amount of software in each device is growing out of all recognition. In this article, we will take a look at the inner workings of such software. The application we will use as an example is from the consumer segment—a digital camera—which is a good choice because whether or not you work on consumer devices, you will have some familiarity with their function and operation.

### Development Challenges

Consumer applications are characterized by tight time-to-market constraints and extreme cost sensitivity. This leads to some interesting challenges in software development.

#### *Multiple Processors*

Embedded system designs that include more than one processor are increasingly common. A digital camera typically has two: one deals with image processing and the other looks after the general operation of the camera. The biggest challenge with multiple processors is debugging. The code on each individual device may be debugged—the tools and techniques are well understood. The challenge arises with interactions between

the two processors. There is a clear need for debugging technology that addresses the issue of debugging the system—that is, multicore debugging.

### Limited Memory

Embedded systems almost always have limited memory. Although the amount of memory may not be small, it typically cannot be added on demand. For a consumer application, a combination of cost and power consumption considerations may result in the quantity of memory also being restricted. Traditionally, embedded software engineers have developed skills in programming in an environment with limited memory availability. Nowadays, resorting to assembly language is rarely a convenient option. A thorough understanding of the efficient use of C and the effects and limitations of optimization are crucial.

If C++ is used (which may be an excellent language choice), the developers need to fully appreciate how the language is implemented. Otherwise, memory and real-time overheads can build up and not really become apparent until too late in the project, when a redesign of the software is not an option. Careful selection of C++ tools, with an emphasis on embedded support, is essential.

### User Interface

The user interface on any device is critically important. Its quality can have a very direct influence on the success of a product. With a consumer product, the influence is overwhelming. If users find that the interface is "clunky" and awkward, their perception of not just the particular device, but also the entire brand will be affected. When it is time to upgrade, the consumer will look elsewhere.

So, getting it right is not optional. But getting it right is easier to say than do. For the most part, the UI is not implemented in hardware. The functions of the various controls on a digital camera, for example, are defined by the software. And there may be many controls, even on a basic model. So, in an ideal world, the development sequence would be:

1. Design the hardware.
2. Make the prototypes.
3. Implement the software (UI).
4. Try the device with the UI and refine and/or reimplement as necessary.

But we do not live in an ideal world. . . .

In the real world, the complexity of the software and the time-to-market constraints demand that software is largely completed long before hardware is available. Indeed, much of the work typically needs to be done even before the hardware design is finished. An approach to this dilemma is to use prototyping technology. With modern simulation technology, you can run your code, together with any real-time operating system (RTOS, etc.) on your development computer (Windows, Linux, or UNIX), and link it to a graphical representation of the UI. This enables developers to interact with the

software as if they were holding the device in their hand. This capability makes checking out all the subtle UI interactions a breeze.

## Reusable Software

Ask long-serving embedded software engineers what initially attracted them to this field of work and you will get various answers. Commonly though, the idea of being able to *create* something was the appeal. Compared with programming a conventional computer, constrained by the operating system and lots of other software, programming an embedded system seemed like working in an environment where the developer could be in total control. (The author, for one, admits to a megalomaniac streak.)

But things have changed. Applications are now sufficiently large and complex that it is usual for a team of software engineers to be involved. The size of the application means that an individual could never complete the work in time; the complexity means that few engineers would have the broad skill set. With increasingly short times to market, there is a great incentive to reuse existing code, whether from within the company or licensed from outside.

The reuse of designs–of intellectual property in general—is common and well accepted in the hardware design world. For desktop software, it is now the common implementation strategy. Embedded software engineers tend to be conservative and are not early adopters of new ideas, but this tendency needs to change.

### *Software Components*

It is increasingly understood that code reuse is essential. The arguments for licensing software components are compelling, but a review of the possibilities is worthwhile.

We will now take a look at some of the key components that may be licensed and consider the key issues.

## Real-Time Operating System

The treatment of an RTOS as a software component is not new; there are around 200 such products on the market. The differentiation is sometimes clear, but in other cases, it is more subtle. Much may be learned from the selection criteria for an RTOS.

### *RTOS Selection Factors*

Detailed market research has revealed some clear trends in the factors that drive purchasing decisions for RTOS products.

**Hard real time:** "Real time" does not necessarily mean "fast"; it means "fast enough." A real-time system is, above all, predictable and deterministic.

**Royalty free:** The idea of licensing some software, and then paying each time you ship something, may be unattractive. For larger volumes, in particular, a royalty-free model is ideal. A flexible business model, recognizing that all embedded systems are different, is the requirement.

**Support:** A modern RTOS is a highly sophisticated product. The availability of high-quality technical support is not optional.

**Tools:** An RTOS vendor may refer you elsewhere for tools or may simply resell some other company's products. This practice will not yield the level of tool/RTOS integration required for efficient system development. A choice of tools is, on the other hand, very attractive.

**Ease of use:** As a selection factor, ease of use makes an RTOS attractive. In reality, programming a real-time system is not easy; it is a highly skilled endeavor. The RTOS vendor can help by supplying readable, commented source code, carefully integrating system components together, and paying close attention to the "out-of-box" experience.

**Networking:** With approximately one third of all embedded systems being "connected," networking is a common requirement. More on this topic later.

**Broad CPU support:** The support, by a given RTOS architecture, of a wide range of microprocessors is a compelling benefit. Not only does this support yield more portable code, but also the engineers' skills may be readily leveraged. Reduced learning curves are attractive when time to market is tight.

### *RTOS Standards*

There is increasing interest in industry-wide RTOS standards, such as OSEK, POSIX (Portable Operating System Interface), and µiTRON. This subject is wide ranging, rather beyond the scope of this article and worthy of an article devoted exclusively to it.

**OSEK:** The short name for the increasingly popular OSEK/VDX standard, OSEK is widely applied in automotive and similar applications.

**µiTRON:** The majority of embedded designs in Japan use the µiTRON architecture. This API may be implemented as a wrapper on top of a proprietary RTOS, thus deriving benefit from the range of middleware and CPU support.

**POSIX:** This standard UNIX API is understood by many programmers worldwide. The API may be implemented as a wrapper on top of a proprietary RTOS.

## File System

A digital camera will, of course, include some kind of storage medium to retain the photographs. Many embedded systems include some persistent storage, which may be magnetic or optical disk media or nonvolatile memory (such as flash). In any case, the best approach is standards based, such as an MS-DOS-compatible file system, which would maximize the interoperability possibilities with a variety of computer systems.

## USB

There is a seemingly inviolate rule in the high-tech world: the easier something is to use, the more complex it is "under the hood."

Take PCs for example. MS-DOS was very simple to understand; read a few hundred pages of documentation and you could figure out everything the OS was up to. Whatever its

critics may say, Windows is easier to use, but you will find it hard (no, impossible) to locate anyone who understands everything about its internals; it is incredibly complex.

USB fits this model. Only a recollection of a few years' experience in the pre-USB world can make you appreciate how good USB really is. Adding a new peripheral device to a PC could not be simpler. The electronics behind USB are not particularly complex; the really smart part is the software. Developing a USB stack, either for the host computer or for a peripheral device, is a major undertaking. The work has been done for host computers—USB is fully supported on Windows and other operating systems. It makes little sense developing a stack yourself for a USB-enabled device.

USB has one limitation, which restricts its potential flexibility: a strict master/slave architecture. This situation will change, as a new standard, USB On-the-Go (OTG), has been agreed upon and will start showing up in new products. This standard allows devices to change their master/slave status, as required. So, USB gets easier to use and—guess what—the underlying software becomes even more complex. Many off-the-shelf USB packages are available.

### Graphics

The LCD panel on the back of a camera has two functions: it is a graphical output device and part of the UI. Each of these functions needs to be considered separately.

As a graphical output device, an LCD is quite straightforward to program. Just setting RGB values in memory locations results in pixels being lit in appropriate colors. However, on top of this underlying simplicity, the higher-level functionality of drawing lines and shapes, creating fills, and displaying text and images can increase complexity very rapidly. A graphic functions library is required.

To develop a GUI, facilities are required to draw screen elements (buttons, icons, menus, etc.) and handle input from pointing devices. An additional library, on top of the basic graphics functions, is required.

### Networking

An increasing number of embedded systems are connected either to the Internet or to other devices or networks. This may not sound applicable to our example of a digital camera, but Bluetooth connectivity is quite common and even Wi-Fi-enabled cameras have been demonstrated.

A basic TCP/IP stack may be straightforward to implement, but adding all the additional applications and protocols is quite another matter. Some key issues are worthy of further consideration.

#### *IPv6*

IP is the fundamental protocol of the Internet, and the currently used variant is v4. The latest version is v6. (Nobody seems to know what happened to v5.) To utilize IPv6

requires new software because the protocol is different in quite fundamental ways. IPv6 addresses a number of issues with IPv4. The two most noteworthy are security (which is an add-on to IPv4 but is specified in IPv6) and address space. IPv6 addresses are much longer and are designed to cover requirements far into the future (see Figure 1.1).

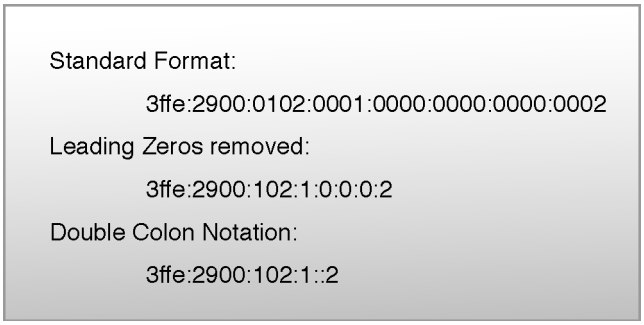



*Figure 1.1: IPv6 addresses*

If you are making Internet-connected devices, do you need to worry about IPv6 yet?

If your market is restricted to nonmilitary/nongovernment customers in North America, IPv6 will not be a requirement for a few years. If your market extends to Asia or Europe or encompasses military applications, you need to consider IPv6 support now. You also need to consider support for dual stacks and IPv6/IPv4 tunneling.

### *Who Needs a Web Server?*

The obvious answer to this question is "someone who runs a web site," but, in the embedded context, there is another angle.

Imagine that you have an embedded system and you would like to connect to it from a PC to view the application status and/or adjust the system parameters. This PC may be local, or it could be remotely located, connected by a modem, or even over the Internet; the PC may also be permanently linked or just attached when required.

What work would you need to do to achieve this?

The following tasks are above and beyond the implementation of the application code:

- Define/select a communications protocol between the system and the PC.
- Write data access code, which interfaces to the application code in the system and drives the communications protocol.
- Write a Windows program to display/accept data and communicate using the specified protocol.

Additionally, there is the longer-term burden of needing to distribute the Windows software along with the embedded system and update this code every time the application is changed.

An alternative approach is to install web server software in the target system.

The result is:

- The protocol is defined: HTTP.

- You still need to write data access code, but it is simpler; of course, some web pages (HTML) are also needed, but this is straightforward.
- On the PC you just need a standard web browser.

The additional benefits are that there are no distribution/maintenance issues with the Windows software (everything is on the target), and the host computer can be anything (it need not be a Windows PC). A handheld is an obvious possibility.

The obvious counter to this suggestion is size: web servers are large pieces of software. An embedded web server may have a memory footprint as small as 20 K, which is very modest, even when storage space for the HTML files is added

#### *SNMP*

SNMP (Simple Network Management Protocol) is a popular remote access protocol, which is employed in many types of embedded devices. The current version of the specification is v3.

SNMP offers a very similar functionality to a web server in many ways. How might you select between them?

If you are in an industry that uses SNMP routinely, the choice is made for you. If you need secure communications (because, for example, you are communicating with your system over the Internet), SNMP has this capability intrinsically, whereas a web server requires a secure sockets layer (SSL). On the other hand, if you do not need the security, you will have an unwanted memory and protocol overhead with SNMP.

A web server has the advantage that the host computer software is essentially free, whereas SNMP browsers cost money. The display on an SNMP browser is also somewhat fixed; with a web server, you design the HTML pages and control the format entirely.

### Conclusion

As we have seen, the development of a modern embedded system, such as a digital camera, presents many daunting challenges. With a combination of the right skills and tools and a software-component-based development strategy, success is attainable. But what will be next?

#### What Is So Simple About SNMP?

SNMP, which stands for "Simple Network Management Protocol," can be described in many ways (e.g., versatile, standardized, secure, and widely used), but simplicity is not one of its attributes. So, why the name?

The answer is that it is not a *simple* network management protocol; it is a *simple network* management protocol. It is designed for the management of equipment on "simple" networks—LANs or maybe even point-to-point links. So the name does make sense.

## 1.2 Memory in Embedded Systems

*I am often asked to explain what embedded systems actually are. This tends to lead to a supplementary question about how they differ from "normal" computers, from a software point of view. The nature and treatment of memory is a fine example of how the two worlds often differ. This is as true today as it was when I wrote an article about the topic in* NewBits *in the early 1990s, upon which this article is based. The concepts have not changed, but some of the numbers have. I observe that, in the original text, I described 4 M as the normal RAM size for a PC! (CW)*

### Memory

When people discuss the advances in microelectronics over the last few years, chances are they are raving about the speed of the latest RISC chip or just how fast a Pentium can go if you keep it cold enough. However, a much quieter revolution has been taking place in the same context: memory has been growing.

Consider some of the implications of this revolution. The PC is just over 20 years old; the original model had just 16 K of memory, whereas 512 M is now considered an average amount. The mainframe I used 20-odd years ago had rather less memory than my palmtop. Where is it all going to end?

I suppose the theoretical limit of memory density would be one bit per atom. I have even read that this limit is being considered as a practical proposition. With that kind of memory capacity, how big of an address bus do you need? Thirty-two bits is certainly not enough. How about 64? Maybe we should go straight to a 128-bit bus to give us room to move. Will that last for long? The answer is easy this time: we will never need an address bus as big as 128 bits because $2^{128}$ is well beyond our current (or any likely future) capacity.

#### *What Is Memory?*

That really should be an easy question to answer, but you would get a different response from different people.

A hardware engineer would respond:

> Memory is a chip in which you can keep bits of data. There are really two kinds: ROM and RAM. These, in turn, come in two varieties each. There is masked programmed ROM and programmable devices, which you can program yourself. RAM may be static, which is easy to use but has less capacity; dynamic is denser but needs support circuits.

A typical software engineer would respond:

> Memory is where you run your program. The code and data are read off of the disk into memory, and the program is executed. You do not need to worry too much about the size, as virtual memory is effectively unlimited.

An embedded systems programmer would respond:

> Memory comes in two varieties: ROM, where you keep code and constants, and RAM, where you keep the variable data (but which contains garbage on startup).

A C compiler designer would say:

> There are lots of kinds of memory: there is some for code, variable data, literals, string constants, initialized statics, uninitialized statics, stack, heap, some is really I/O devices, and so forth.

Four differing answers to the same question! These responses do not contradict one another, but they do reflect the differing viewpoints.

### *Memory in Embedded Systems*

These differing views of the nature of memory are quite likely to come sharply into focus on an embedded system project. The hardware designer puts memory on the board, the C compiler designer provides the software development tools, and software engineers often end up doing the programming. An experienced embedded systems programmer has learned to reconcile the differences, understands what the hardware engineer has provided, and understands how to use the development tools to make the program fit into that environment.

## Implementation Challenges

What problems need to be overcome when implementing software for an embedded system?

### *ROMable Code*

The first and most obvious challenge when implementing software for an embedded system is to arrange for code to be stored in ROM and variable data to be assigned RAM space.

This radically differs from a "normal" computer, where code and data are simply loaded into (read/write) memory as a unit. Data may, therefore, be mixed up with the code, and its values may be set up at compile time. Furthermore, although not a sensible practice, programs may be made self-modifying since the code is stored in read/write memory. A true cross-compiler generates code that is ROMable (i.e., will execute correctly when stored in ROM). In addition, a linker intended for such applications facilitates the independent positioning of code into ROM and data into RAM.

### *Program Sections*

To accommodate the need to treat various types of memory differently, the concept of a program "section" evolved. The idea is that memory could be divided into a number of

named units, called sections. While coding in assembly, the programmer could specify the sections where code, data, constants, etc. are placed.

The actual allocation of memory addresses to sections takes place at link time. The linker is provided with start addresses for each section or the start address where a sequence of sections, in a specified order, may be placed. Contributions from a number of object modules may be made to a given section. Normally, these are concatenated and placed at the address specified.

For an embedded system, at the simplest level, just two program sections are needed: one for code and constants (ROM) and one for data (RAM). In each module, the programmer takes care to indicate the appropriate section for each part of the program. At link time, all the code and constants are gathered together and placed in ROM, and all the data is placed in RAM.

### *Static Variables*

In C, any variable that is not automatic (i.e., on the stack or held in a register) is stored statically. A static variable has memory allocated for it at compile time. When such a variable is declared, there is the option of providing an initial value. If no value is specified, the variable is set to zero.

Clearly there is a potential problem with the use of static variables in an embedded system. The values set up at compile time would be lost, as only code (and constant data) is blown into ROM. There are three possible solutions:

1. Do not use initialized static variables. Use explicit assignments to set up their values and never assume an unassigned variable contains zero. Although this approach is possible, it is inconvenient.
2. Map the initialized statics into ROM. This means that although they do have the required initial value, they cannot be changed at all later. While this may sound restricting, it is often useful to have look-up tables of variables (more likely structures), which are actually treated as constants.
3. Map the variables into RAM, if the compiler package in use permits, with their initial values being mapped into ROM. It is a simple matter to copy one area of (ROM) memory to another (RAM) at startup, before `main()` is called.

## When All Goes Wrong

When developing software for an embedded microprocessor, the reconciliation of the various views of memory may not be easy. If the compiler simply divides the code and data into two sections, destined for ROM and RAM, respectively, the possibilities for resolving the initialized statics problem are limited. Furthermore, the compiler may treat literal text strings as data, which would be inconvenient because they need to reside in ROM with the code.

### When All Goes Right

A true embedded toolkit can make dealing with memory in an embedded system very straightforward. The use of named program sections are employed to the full. This support begins with the compiler, which generates a number of sections. A typical list is as follows:

- `code`—The program code
- `zerovars`—Uninitialized static
- `initvars`—Initialized statics
- `const`—Variables declared `const`
- `strings`—Constant text strings
- `literals`—Compiler-generated literals
- `tags`—Compiler-generated tags

The names of the sections may be changeable and will vary from one compiler to another. Using the assembler, any assembly code may contribute to these sections or create others.

The final stage is the use of the linker, which enables each section to be placed in the appropriate part of the memory map. In addition, there may be a command that permits initialized data to be set up from values held in ROM at startup, with minimal effort.

So the story has a happy ending and the problems of diverse memory in an embedded system may be solved by the use of the right tools.

## 1.3 Memory Architectures

*The perception of what a memory address really is can be challenging for many engineers, who are unaccustomed to thinking in such "low-level" terms. I wrote a piece for* NewBits *in 1992 that investigated this topic, and it was the basis for this article. (CW)*

I remember when the term "architecture" always seemed to refer to buildings—their style and design. I guess this viewpoint was initiated, or at least reinforced, by the letters "RIBA" (Royal Institute of British Architects) on my father's business card. Nowadays, everything seems to have an architecture. Chip architecture is a good example, where it has spawned the term "silicon real estate," which carries the analog with the construction industry a little bit further. In this article I will give some consideration to memory architecture, as implemented by microprocessors used in embedded systems.

### The Options

In general, memory architectures fall broadly into five categories:

- Flat single space
- Segmented
- Bank switched
- Multiple space
- Virtual

Figures 1.2–1.5 illustrate diagrammatically the first four schemes. Every microprocessor may employ one or more of these possibilities. Each chip offers a different combination, and each option may be more or less appropriate for a specific application. Cache memory, while not strictly a memory architecture, is also discussed.

### Flat Single-Space Memory

Flat memory is conceptually the easiest architecture to appreciate. Each memory location has an address, and each address refers to a single memory location. The maximum size of addressable memory has a limit, which is most likely to be defined by the word size of the chip. Examples of chips applying this scheme are the Motorola 68 K and the Zilog Z80.

Typically, addresses start at zero and go up to a maximum value. Sometimes, particularly with embedded systems, the sequence of addresses may be discontinuous. As long as the programmer understands the architecture and has the right development tools, this discontinuity is not a problem.

Most programming languages, like C, assume a flat memory. No special memory-handling facilities need be introduced into the language to fully utilize flat memory. The only possible problems are the use of address zero, which represents a null pointer in C, or high addresses, which may be interpreted as negative values if care is not exercised.

Linkers designed for embedded applications that support microprocessors with flat memory architectures normally accommodate discontinuous memory space by supporting scatter loading of program and data sections. The flat address memory architecture is shown in Figure 1.2.

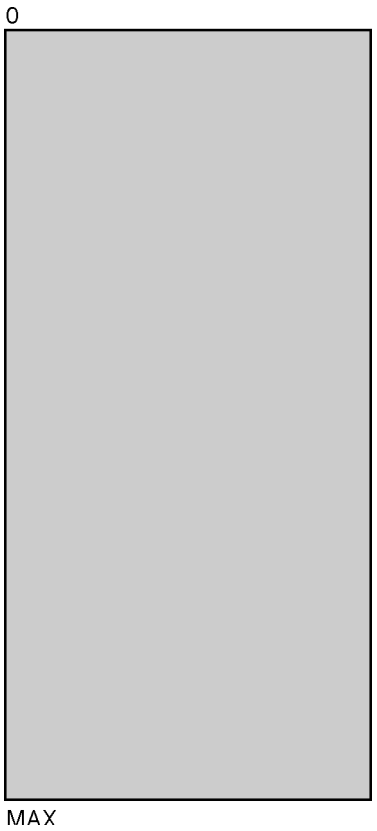



*Figure 1.2: Flat address memory architecture*

### Segmented Memory

A perceived drawback of flat memory is the limitation in size, determined by the word length. A 16-bit CPU could only have 64 K of memory and a 32-bit architecture may be considered overkill for many applications. The most common solution is to use segmented memory (see Figure 1.3). Examples of chips applying this scheme are the Intel 8086 and the Hitachi H8/500.

The idea of segmented memory addressing is fairly simple. Addresses are divided into two parts: a segment number and an offset. Offsets (usually 16 bits) are used most of the time, where the additional high-order bits are held in one or more special segment registers and assumed for all operations. To address some memory over a longer range, the segment registers must be reloaded with a new value. Typically, there are individual segment registers for code, data, and stack.

The use of segmented memory necessitates the introduction of the concepts of "near" and "far" code and data. A near object may be accessed using the current segment register settings, which is fast; a far object requires a change to the relevant register, which is slower. Since segmented memory is not immediately accommodated by high-level languages, `near` and `far` (or `_near` and `_far`) keywords must be introduced. With these keywords, you can specify the addressing mode used to access a code or data item. Using a "memory model," you can specify default modes. For example, a "large" model would access all objects as "far," and a "small" model would use "near" for everything. With segmented memory, the size of individual objects (e.g., modules or data arrays) is generally limited to the range addressable without changing the segment register (typically 64 K).

Compilers for chips with segmented memory typically implement a wide range of memory models and the `far` and `near` keywords.

### Bank-Switched Memory

Another approach to extending the addressing range of a CPU is bank switching. This technique is a little more complex than segmented memory, but it has the advantage that

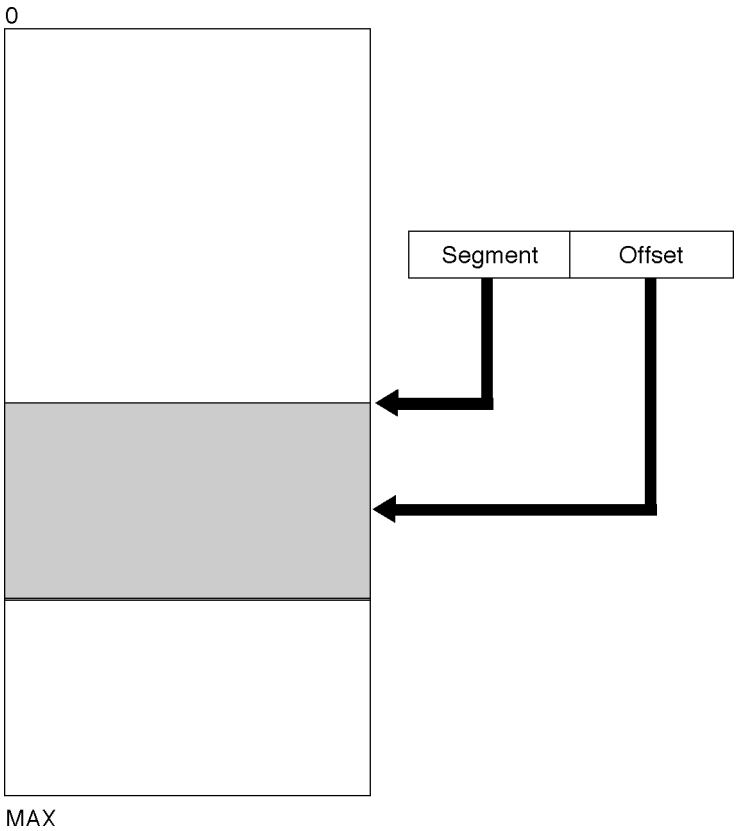



*Figure 1.3: Segmented memory architecture*

such memory may be implemented with a processor that does not itself support extended memory.

A bank-switched memory scheme comprises two parts: a range of memory addresses, which represent a "window" into a larger memory space, and a control register, which facilitates the moving of this window (see Figure 1.4). Accessing the bank-switched memory area requires the control register settings to be verified and adjusted, if necessary, before the required location is accessed within the window.

Little, if anything, can be done with a C compiler to accommodate bank-switched memory. However, a linker may provide an "overlay" scheme to enable a number of data items to exist apparently at the same address. Better still, the linker could implement the concept of "logical views"—that is, groups of modules within which a certain setting of the control register(s) may be held constant. Transfers (jumps or calls) between logical views are performed by code that performs the necessary bank switch.

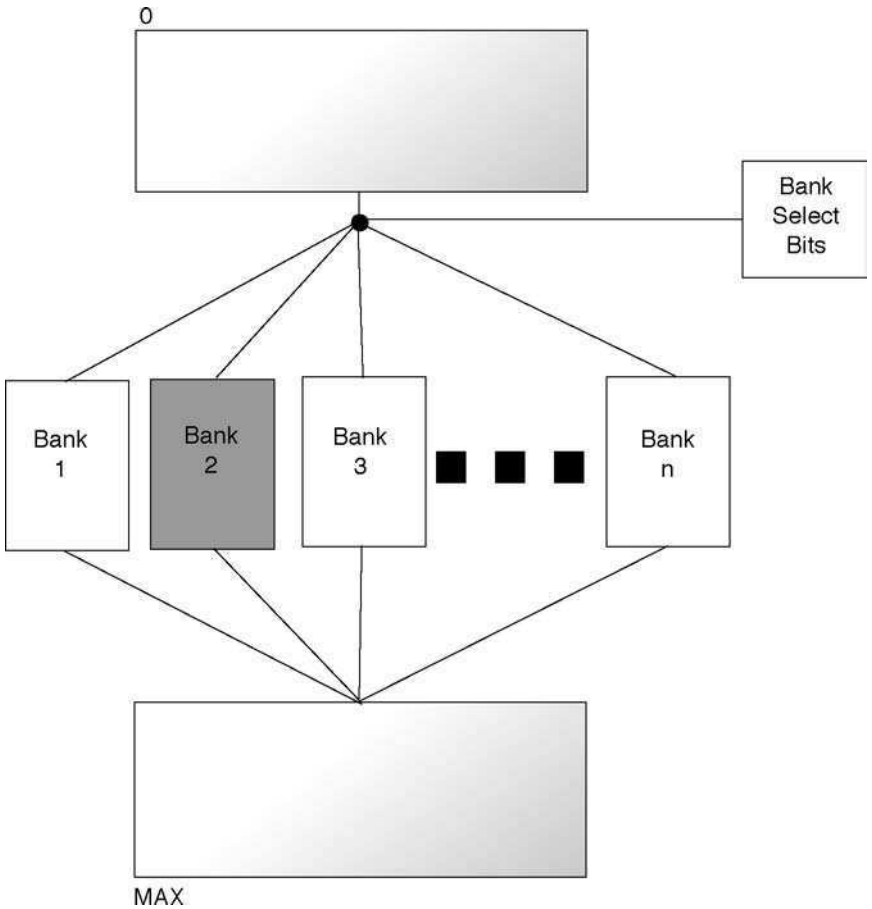



*Figure 1.4: Bank-switched memory architecture*

### Multiple-Space Memory

Yet another approach to increasing the memory-addressing range of a limited number of addresses is the use of multiple memory spaces (see Figure 1.5).

A Motorola 68000, for example, may optionally have four address spaces: user code and data and supervisor code and data. Since the functions of these spaces are essentially noninterchangeable, no particular provision is required within a high-level language.

The Intel 8051 family employs multiple address spaces, which map to different types of memory: on-chip and off-chip ROM and RAM, etc. Since such memory may not be addressed unambiguously (as a given address may correspond to several actual memory locations, any one of which may be validly addressed), special data type modifiers need to be added to high-level languages, like C, to qualify variables appropriately.

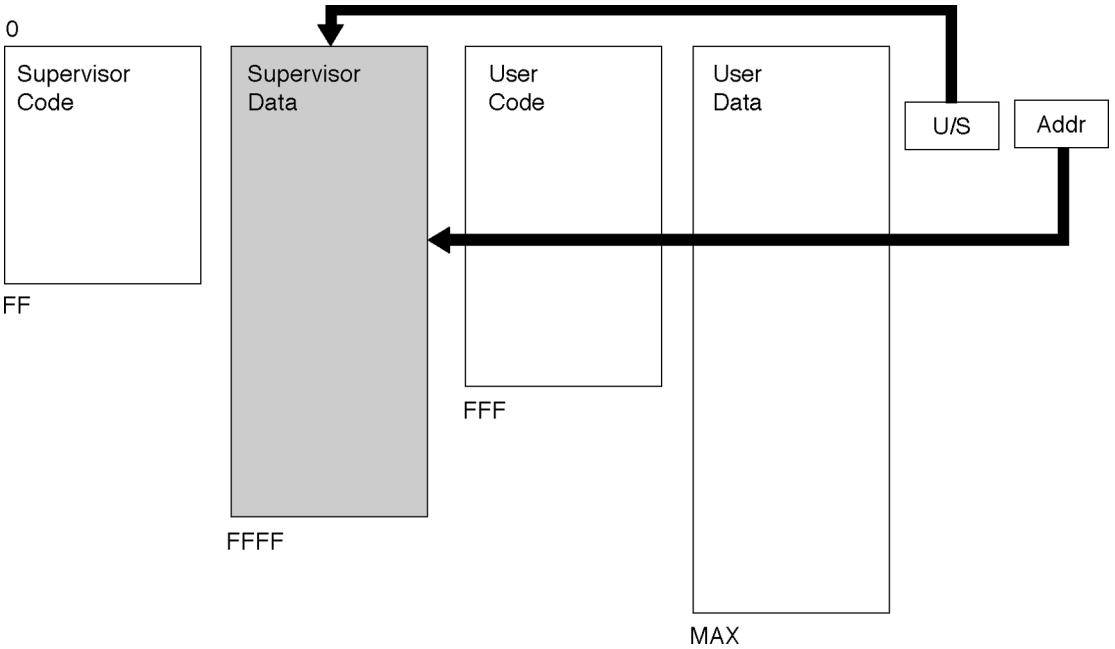



*Figure 1.5: Multiple-space memory architecture*

### Virtual Memory

The virtual memory scheme has a long history but has remained appropriate to modern computer systems, where memory demands continually escalate. Since it is very rarely employed in embedded systems, little space will be devoted to its consideration here. Virtual memory gives you the impression that memory is of indefinite size. This impression is achieved by automatically swapping areas of memory on and off of the hard disk using specialized hardware facilities. A delay occurs whenever a memory location that is currently swapped out must be read in. The delay reduces the usefulness of virtual memory for real-time systems.

### Cache Memory

Although not strictly a memory architecture by the definition of those described previously, memory caches are becoming a common feature of many modern, high-performance microprocessors. A full discussion of memory cache design and implementation would fill an entire article or more by itself. Bearing in mind that the presence of a cache may generally be ignored by a programmer, only a brief comment is appropriate here.

A memory cache is a special area of high-speed memory within or adjacent to the CPU. The cache controller reads from regular memory into the cache when a memory location is accessed. Subsequent accesses to the same or nearby locations can then be

performed from the cache, without accessing the slower main memory. The performance of a small code loop may often be significantly increased when the entire code can reside in the cache.

By careful sequencing of instructions, the best performance may be extracted from a cache facility. Recent compilers for chips with such an architecture often take advantage of this technique.

### Memory Management Units

Many 32-bit microprocessors, which are used for embedded applications, include a memory management unit (MMU). It is either built in or may be included as an option. An MMU provides a means to protect memory from corruption. Typically it is used with a real-time operating system (RTOS), where each task and the OS itself can be protected from malfunctioning code in another task. This may be done by simply blocking (or write-protecting) memory areas or through the implementation of a "process model," where each task apparently has a complete private address space starting at zero. From the embedded applications programmer's point of view, very little action needs to be taken to accommodate the use of an MMU. It is largely an issue for the RTOS.

### Conclusions

Understanding the memory architecture of a chip is essential to determine its appropriateness for a specific application. For very large or very small applications, flat memory is usually best. For small programs with a lot of data, bank-switched memory may be particularly suitable.

## 1.4 How Software Influences Hardware Design

*There is always a tension between hardware and software specialists. Even today, when the boundaries are becoming increasingly indistinct, the labels persist. I have always felt that I am on a crusade to bring the two sides together. This is what I had in mind when I wrote a piece for* NewBits *in 1992, which was the basis for this article. (CW)*

At the time of writing, the Festive Season still seems mercifully far off. I do not like to think about Christmas at all until a couple of weeks before. You may well ask: what has this got to do with embedded systems design? The answer lies in an old English Christmas custom, which has an interesting parallel in the electronics world that I will discuss further. In the nineteenth century, December 26 was designated "Boxing Day"; the day upon which tradesmen received their "Christmas Boxes" (i.e., bonus money or another gift). This seems slightly illogical, as I am sure they would rather have had the extra money before Christmas than after, but I digress. The tradition on Boxing Day was for masters and servants to change places for the day; the same practice in the armed forces resulted in officers and men swapping roles. The objective, in both cases, was to enable each "side" to see the other's point of view more clearly. Could the same idea be applied in the embedded system development lab?

### Who Designs the Hardware?

In a small company, or even a small development department of a larger operation, there may well be a limited number of engineers (or even just one) who design embedded systems in their entirety. Imagine the answer to your question about the staff deployment: "Fred over there does our embedded stuff." The definition of "stuff" would cover both hardware and software. This situation is fine for the development of small systems. Fred probably understands all the aspects of the application and can optimize the interaction of hardware and software accordingly, assuming that he is sufficiently expert in both disciplines to evaluate the trade-offs properly.

In a larger lab, there is much more likely to be a sharper differentiation between hardware and software, with just a small number of "gurus" who "have a foot in both camps."

Decisions made at the start of an embedded systems project have implications that last through its design and often into its production. Skimping on hardware to reduce cost or power, for example, could cost you extra programming time at the end. Splurging on hardware by overestimating your needs, however, burdens your design with extra cost on each unit shipped. Designers usually have hardware or software expertise, but not both.

### Software Leading Hardware

Typically, the design of an embedded system starts with the hardware. Only when that has been (irreversibly) designed is software taken into consideration. The efficiency of the implementation would be ultimately enhanced if software were considered at a much earlier stage. It would be ridiculous to actually suggest that the software engineers should do the hardware guys' work, but their early involvement would be useful.

If nothing else, it would minimize the "finger pointing" during the integration phase, when each "side" blames the other for any problems that arise.

## Software/Hardware Trade-Offs

The consideration of software requirements and capabilities has a real influence at many points during the hardware definition.

### *Microprocessor Selection*

A factor in microprocessor selection is the availability of advanced software development tools. Although low-power chips are specified for many applications, the saving of a few microwatts of CPU power consumption is often used up by the additional memory needed to store inefficient code.

### *Memory Size and Mix*

A decision on the exact amount of memory and the mix of ROM (flash, etc.) and RAM should be made as late as possible in the design cycle, since this is often hard to predict. Cramming code and data into too small a memory area is a problem, and having too much memory has price and power implications. Sometimes memory sites can be designed to take either RAM or ROM, which means extra traces for the Write Enable signal. This strategy allows the memory type mix be determined late in the design cycle and offers the ability to replace most of the ROM with RAM for debugging purposes (see the section, "Debug Hardware," that follows).

### *Peripheral Implementation*

The inclusion of other peripheral devices in the design should be carefully considered. Timers and serial I/O, for example, can be implemented in software when necessary. There is a greater load on software when each bit of serial communication has to be processed instead of having hardware handle a whole byte. However, if CPU processing power is available and hardware cost is a concern, having the software do the work is often a reasonable solution. After all, you pay for the software development just once, but an extra chip adds cost to every unit shipped. This trade-off must be made carefully, because the considerations of a low-volume application (where software development cost is dominant) differ from those for a product shipping in large volume (where unit cost is the issue).

## Debug Hardware

Adding hardware to help debug software does not occur to many hardware designers. It is a matter worthy of further thought.

### *In-Circuit Emulators*

Historically in-circuit emulators (ICEs) were the instrument of choice for embedded software development. They provided a totally unintrusive way to debug code at full

speed on a real target. But as processor complexity increased and, more importantly, clock speeds escalated, ICEs became more and more expensive and their availability declined. Now they tend to be available for only low-end devices.

If an ICE is available and the cost is not a problem, then this may offer the very best debugging tool. However, in reality, a full team of software developers rarely can be equipped with ICEs for software debugging. These instruments are ideal for addressing complex problems resulting from the close interaction of the software and the hardware, and if they are in short supply, should be reserved for this kind of job.

### *Monitor Debuggers*

A good solution, in many cases, is to make use of a monitor debugger. This requires some provision in the target hardware: ROM must be replaced temporarily by RAM and an additional serial port or other I/O device must be available at debug time. Neither of these requirements presents a problem in most circumstances if the need is considered early in the design phase. The result is the ability to debug the code at full speed on the target hardware. This does not have all the facilities of an ICE, but quite enough for most purposes. In particular, a monitor-based debugger can be the basis of a run-mode debug facility with a real-time operating system.

### *On-Chip Debug Support*

Most modern microprocessors provide on-chip support for software debugging—generically termed "on-chip debug" (OCD). The Freescale 683xx microcontroller family (those based on the CPU32 core; not the 68302), for example, has background debug mode (BDM). Many other devices use a JTAG interface with a similar result.

OCD is generally implemented in the microcode of the chip and offers a unintrusive debugging capability. To use OCD, an adaptor is needed between the target and the host machine to handle the special synchronous protocol and manage the interface to a high-level language debugger. Connecting this adapter to the target board requires an appropriate connector, but the cost and space to add it are negligible. However, this must be considered during the design phase.

## Self-Test Support

Most embedded systems have a degree of built-in self-testing. Sometimes self-testing is executed only at power-up to check for any chip failures that may have occurred. In other systems, the self-test code may be run as a background task when no other processing is required.

The programming techniques employed for self-testing really deserve an article to themselves, so, for the moment, we will concentrate on the provisions that can be incorporated into the hardware design.

### I/O Circuits

The self-test facilities that may be incorporated into I/O circuits are device-specific. A typical example is a serial line (RS232) interface. To confirm that the serial interface chip is transmitting and receiving correctly, a "loopback" can be implemented. This capability results in each transmitted character being sent straight to the receiver. The self-test code first sends a sequence of characters and then verifies that they are being received correctly. The loopback channel in the I/O circuit must be controllable by the software.

### On-Board Switches

An on-board switch (or, better still, a small switch bank), jumpers, or press buttons can provide test commands to the software. With a four-way switch bank, for example, one switch could activate the self-test mode, and the other three could select one of eight possible baud rates for a serial line.

### Status Displays

A display could be as complex as a line of LCD characters upon which messages can be written, or as simple as a single LED, which can be turned on or off by the software. A surprising amount of information can be conveyed by such a single LED. It can be in one of three simple states: on, off, or flashing. A good approach is to use flashing to indicate the software is OK, because this requires action to maintain the flashes and is, therefore, fail-safe. Of course, the instructions to turn on and off the LED should ideally be embedded in the main software loop. The fixed on and off states can then be used to indicate two possible failure conditions. Just like with lighthouses, the way the LED flashes can indicate a status. There are two variations of flashing that can be employed: bursts of flashes separated by a pause (where the number of flashes in the burst can indicate a particular status) or variable duty cycles (ratio of on and off times) can convey different states. Each of these has its attractions, but the duty cycles method is probably a little easier to implement and understand.

## Conclusions

Addressing software early, while the hardware design is still fluid, is the way to avoid software/hardware mismatches in an embedded systems project. A serial port or other I/O device on the target hardware, for example, lets a monitor-based debugger work at full speed on the final hardware design to debug optimized code. Since debugging hardware may be absent from the production boards or may have other uses in the application, there may be no additional cost at all.

Making the right decisions at the start of an embedded systems project is important because skimping or splurging on hardware costs time and money.

## 1.5 Migrating Your Software to a New Processor Architecture

*In the mid-1990s, Motorola (now Freescale) dominated the high-end embedded processor market, but things were changing fast. It was with this change in mind that, in 1996, I wrote a piece on migrating between processors for* NewBits*, which was the basis for this article. Two other articles in this book extend on the concepts: "PowerPC Assembler" (Chapter 3) gives more information on EABI; "On The Move—Migrating from One RTOS to Another" (Chapter 7) expands on issues only touched upon here. (CW)*

A few years ago, the basic decision involved in developing a high-end embedded system—"which processor should I use?"—was fairly easy to answer. It could, more often than not, be translated into "which 68 K family device should I choose?" Things have changed. Freescale offers two other architectures for high-end applications: PowerPC and ColdFire. At the same time, a plethora of other 32-bit devices compete for attention, all with strong differentiating factors that can make them an attractive choice.

Having chosen the processor for the next project, the next issue to address is the development of new expertise in programming the device and porting existing code. There are three major topics to cover:

- Avoiding target-specific code
- Real-time operating system (RTOS) issues
- Open standards and how they aid target processor migration

### Target Specifics

The primary aspects of an embedded application, which have a degree of target specificity, are the code, data, and real-time structure.

#### *Code*

Code written in C or C++ will itself tend to be portable. The only real concern is the precise dialect of the language that has been used. If the change of processor necessitates a change of compiler supplier, there may be some issues. In particular, extensions to the language (e.g., interrupts) may be implemented differently or may include a proprietary set of keywords.

A nonlinear change is likely in the runtime performance of the code. Assuming, for example, that the new chip is supposed to deliver a 2X performance increase, this will not be the case with all language constructs; some will be even faster than this, others will benefit less from the higher processor performance. Some architectures lend themselves more readily to a particular program structure or language feature.

Assembler code will inevitably require a rewrite. Even the migration from 68 K to ColdFire requires a very careful review of the code to ensure that it is confined to the

instruction subset supported by these devices. This is an ideal time to review whether some of the assembler code can be replaced by a (portable) high-level language implementation.

### Data and Variables

Since the definition of the storage allocation for data types in C/C++ is, by definition, target-specific, a change in this area may be anticipated. For example, an `int` may be allocated 32 bits instead of 16 bits. This is quite likely to affect the performance and functionality of the code as well as the RAM requirements of the application, which may increase or decrease unexpectedly. In practice, the storage allocation schemes for most 32-bit target compilers are essentially identical, so little or no difficulty should arise.

To completely eliminate problems, some engineers implement a defensive strategy, which renders their code more portable. This approach is very simple: use `typedef` statements to implement some bit-size-specific data types in unsigned and signed variants: `U8`, `U16`, `U32`, `S8`, `S16`, and `S32`. Porting the code to a new device and/or compiler is simply a matter of making a slight edit to a header file.

Structure layout and alignment may change, as will bit field allocation. This problem is unlikely to be encountered, because it is bad practice to design code that relies on these factors.

According to the language definitions, `enum` variables should be allocated the same space as `int`. However, many compilers optimize the space, allocating 1, 2, or 4 bytes, depending upon the range of values required. Again, this could result in an unexpected change in the performance or memory requirements for an application.

Native compilers, used for the development of desktop applications, are solely focused on the generation of fast code, at the expense of compactness if necessary. For embedded systems software development, such an assumption about a user's requirements would be inappropriate. For a program to achieve maximum speed, data must be laid out in memory such that it can be accessed efficiently. This necessitates waste; extra bytes are added for "padding" to facilitate the correct alignment. Cross-compilers generally have a facility to optionally pack data—i.e., they eliminate the padding and generate any extra code that is required to access the data. This may be implemented differently for a different processor, leading to code incompatibility and an increase or decrease in memory requirements.

### Data and Function Parameters

The conventional method for function parameter passing is to push each value onto the stack, from right to left. Any code that relies on this mechanism will not be portable. Here is an example:

```
fun(int n)
{
      int *p, x;
      p = &n;
      x = *++p;
      ...
}
```

High-performance devices tend to have a RISC architecture, with a large set of registers. These may be more efficiently employed for parameter passing, which would break this code.

For straightforward C or C++ code, the details of parameter passing are unimportant. Parameter details are important when assembler code is involved or when libraries of code built with a different compiler are to be used.

### Data and Register Usage

A modern optimizing compiler will make effective use of CPU registers for storing automatic variables in C and C++ programs. High-performance RISC devices, with much larger register sets, will benefit even more from this optimization. Also, a "smart" optimizer will perform "register coloring," which makes even more efficient use of the available registers. This technique is even a benefit to performance when the microprocessor itself performs register renaming. In this example, `i` and `j` are likely to be allocated to the same register:

```
color()
{
       int i, j;
       for (i=0; i<4; i++)
             ...
       ...
       for (j=0; j<4; j++)
             ...
}
```

Although the use of registers to hold variables is transparent to the C or C++ programmer, except possibly at debug time, it may affect the runtime performance and stack requirements unexpectedly. Also, the use of the `register` keyword, which advises the compiler that a particular variable is heavily used, may be more effective than usual.

### Data and Endianness

Memory is normally just a series of byte-size storage locations. The way these are grouped together to form 16- or 32-bit words is arbitrary. There are broadly two

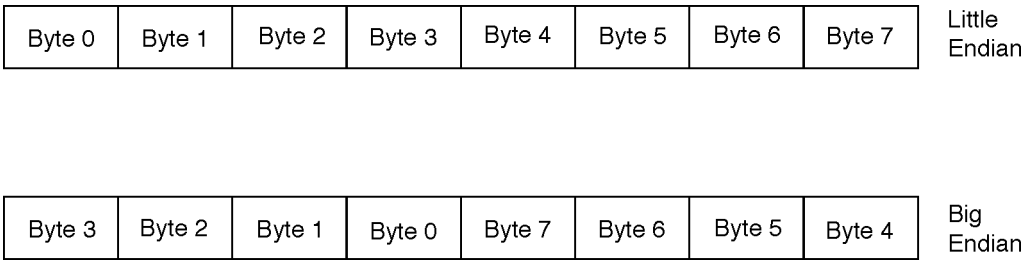



*Figure 1.6: Endianness and byte allocation*

approaches: the most significant byte of the word can be allocated the lowest address (which is "big-endian") or it can have the highest address ("little-endian"). These two approaches are illustrated in Figure 1.6.

For most code, the "endianness" (or "endianity") is irrelevant. Certain program constructs are sensitive, and they may lead to nonportable code. They should be avoided.

There can be a problem when a big-endian processor exchanges data with a little-endian machine. It is necessary for one or another format to be established as the standard for such an interchange and any necessary byte-swapping performed in software.

Many modern microprocessors, like the PowerPC, are flexible and may be operated in either big- or little-endian mode. Most compilers will generate code for either model, but care may be required because libraries may be available in only one of the two formats.

### *Real-Time Concerns*

Moving to a higher-performance microprocessor will obviously increase the speed of the application. As mentioned previously, this performance increase may be nonlinear for various reasons. This is rarely a source of difficulty. Of more significant concern are constructs that characterize the real-time structure of the program. The most significant examples are interrupts.

The interrupt mechanism on various chips is different. Also, it is likely that the design of the hardware around the CPU will differ in a new design. The net result is that a rewrite of the interrupt code is very likely to be necessary.

To maximize the possibilities for portability, interrupt service routines should be written in C or C++ (assuming that the compiler supports this option). The interrupt vector can most likely be coded in high level too; an array of pointers to functions is usually sufficient.

### RTOS Issues

For higher-end embedded systems, the use of a real-time operating system (RTOS) is common. For many 68 K-based designs, an in-house kernel has been utilized for reasons

of ready availability and perceived cost advantages. This situation complicates issues when migration to an alternate processor architecture is planned. An RTOS often has a high assembler code content, requiring a rewrite, and specific architectural features of the device are frequently exploited, which may necessitate a total redesign.

As designs become more complex, the need for an RTOS becomes even more apparent. Also, time-to-market pressures are always increasing. As a result, three other requirements emerge:

- More comprehensive tool support is essential. A debug technology, which is capable of tackling the multitasking environment, and the special problems that can result, is no longer optional.
- To speed application development, it is very beneficial to use an RTOS that has been designed for a specific environment: telecoms, handheld equipment, automotive, etc.
- The requirements do not end at the RTOS itself. More complex applications, associated with short times to market, demand that, wherever possible, standard application code is brought in, rather than developed in-house. This "intellectual property" may be as simple as hardware drivers or may be more sophisticated middleware or application-enabling technology, such as TCP/IP, SNMP, or Wi-Fi.

These factors point to a requirement to employ a commercial RTOS product, where its availability for the new architecture may be assured and the selection of support products verified. In the longer term, when the next change in processor architecture is necessitated, the vendor is most likely to be motivated to provide support in good time.

### Processor Migration and Open Standards

Although changing microprocessor architecture is part of the problem, many of the headaches associated with migration to a new device come from the necessity to adopt different programming techniques and new tools. The way a compiler makes use of an architecture can strongly affect the way an engineer works. In most cases, compiler design is at the discretion of the tool developer. The result is that a change of processor may require the use of very different tools. Also, an embedded systems developer is likely to find very little interoperability between different vendors' tools.

An open standard governing the application program's interface to the chip architecture was defined with the launch of the PowerPC: the Embedded Application Binary Interface (EABI).

#### *EABI—Introduction*

EABI is based upon the UNIX SVR4 desktop Application Binary Interface for PowerPC, optimized for memory usage, while retaining runtime performance. Compliance to EABI facilitates a number of kinds of tool interoperability:

- Generally any compiler may be used with any debugger.
- Code from multiple compilers may be mixed.
- Third-party binary libraries may be used.

The key matters addressed by the standard are:

- Function-calling conventions
- Register usage
- Data types and alignments
- File/debug formats

Although this article reviews some aspects of EABI, a detailed discussion of the standard is beyond its scope.

### *EABI and Register Usage*

EABI dictates the way the machine registers are applied by a compiler. Three categories of register are defined:

- **Dedicated registers** are used for one specific purpose at all times; for example, `gpr1` is the stack pointer.
- **Volatile registers** may be corrupted through a function call; the calling function must preserve them if their values are required later.
- **Nonvolatile registers** are preserved through a function call; the called function must preserve them if their use is required.

The following table illustrates the allocation of registers and condition register (CR) fields:

| Register | Type | Application |
|---|---|---|
| `grp0` | Volatile | Language-specific use |
| `grp1` | Dedicated | Stack pointer |
| `grp2` | Dedicated | Read-only data area anchor |
| `grp31/Ngrp4` | Volatile | Parameter passing/return values |
| `grp51/Ngrp10` | Volatile | Parameter passing |
| `grp111/Ngrp12` | Volatile | |
| `grp13` | Dedicated | Small data area anchor |
| `grp141/Ngrp31` | Nonvolatile | |
| `fpr0` | Volatile | Language-specific use |
| `fpr1` | Volatile | parameter passing/return values |
| `fpr21/Nfpr8` | Volatile | Parameter passing |
| `fpr91/Nfpr13` | Volatile | |
| `fpr141/Nfpr31` | Nonvolatile | |
| fields `CR21/NCR4` | Nonvolatile | |
| Other `CR` fields | Volatile | |

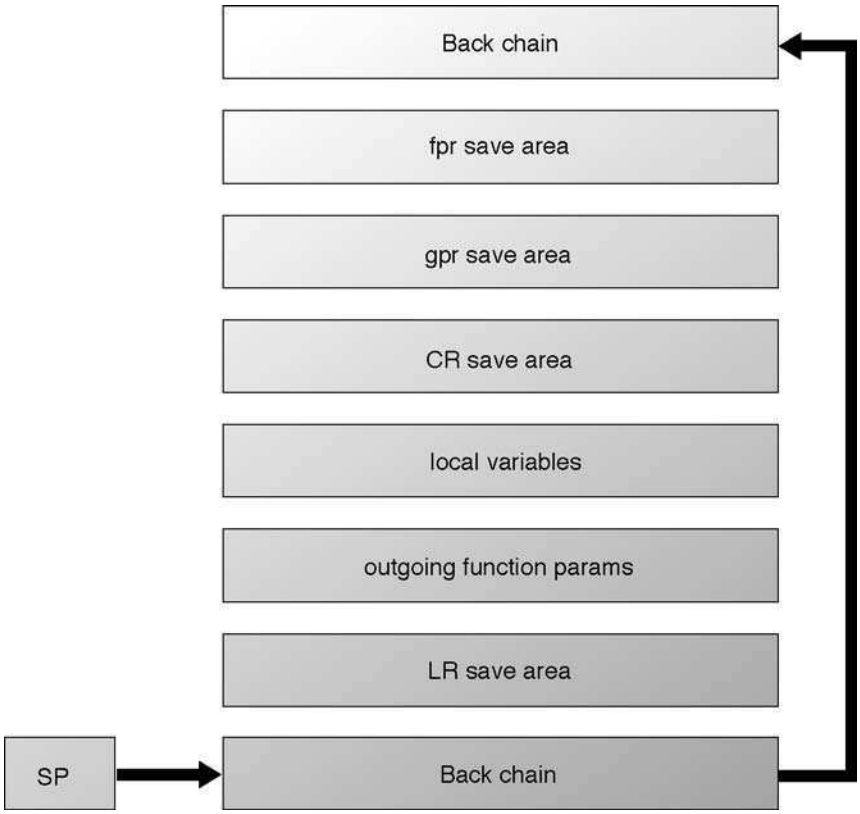



*Figure 1.7: EABI stack-frame format*

### *EABI and Stack Frames*

The C function requirement for local storage is often beyond what can be accommodated in registers. For this purpose, EABI describes the format of stack frames. In Figure 1.7, the format of a complete stack frame is illustrated.

In practice, it may not be necessary to employ all the components of the frame. A minimal stack frame may consist of just 8 bytes. A true leaf function (i.e., a function that does not call any others) may have no need for its own stack frame at all.

### *EABI and Data Types*

Data types, which correspond to those in common high-level languages, are outlined by EABI: byte, half word (2 bytes), word (4 bytes), double word (8 bytes), and quad word (16 bytes). Alignment rules for data structures are also specified, which ensures that the use of memory is consistent from one compiler to another.

The EABI definition encompasses ABIs for both big-endian and little-endian operations. However, these two ABIs are not interoperable.

### EABI and File/Debug Formats

To permit tool interoperability, compatibility of file formats is essential. EABI specifies two formats:

- For object files, *Executable and Linking Format* (ELF) is used. This format is well specified and widely understood.
- For debug information, *Debug With Arbitrary Record Format* (DWARF) is used. This format has been extended by different vendors in different ways, which unfortunately reduces interoperability.

## Conclusions

Migrating to a new processor is an inevitable challenge faced by embedded developers. With careful planning and a good understanding of the issues, the problems may be minimized. Furthermore, a forward-looking strategy can result in code and development procedures that are ready for porting again, when the need arises.

## 1.6 Testing Computers on Wheels

*In recent years, electronics in cars has moved from being the prestige functionality of top-of-the-range models to a significant part of the build cost of most vehicles. A large part of the electronics is, of course, embedded systems. Other articles in this book address some of the technicalities of automotive embedded systems ("Embedded Software for Transportation Applications" in this chapter and "OSEK: An RTOS Standard" in Chapter 7 ). This article is based upon a short web essay by Robert Day, director of marketing at Accelerated Technology, and it illustrates the scope of this growing application area. (CW)*

As telematics systems in automobiles proliferate, they are also becoming more complicated. Their electronics must be designed to ensure that the components work together and that the system is scalable to meet the requirements of various automobiles. Since the devices must connect to each other, to the outside world, and to existing automobile systems, they must be designed using a "system" approach. This introduces the world of connectivity standards such as CAN or COM for internal communications, cellular systems (CDMA, GSM) for voice communications, and potentially 802.11 (Wi-Fi) and Bluetooth for external data communications.

With the explosion in telematics functionality, more embedded software is entering cars. What makes a good and usable telematics system depends more on software than on the hardware design. The software teams need a small, dependable RTOS at the heart of the system. Although OSEK (which stands for "open systems and the corresponding interfaces for automotive electronics" but in German), the automotive standard for powertrain systems, is not a requirement for telematics, it is gaining ground as the de facto standard. The specification of the OSEK operating system standard represents a uniform environment that supports efficient utilization of resources for automotive control unit application software.

Additional components—networking stacks, file systems, and graphical interfaces—are also needed. The business model for this "software IP" must be flexible for the automotive industry. As the volumes increase, so does the attractiveness of a royalty-free software licensing scheme.

Considering the large amount of embedded software, the required new hardware design, and time-to-market pressures, the system must be verified, validated, and tested as early as possible in the development cycle. Software validation can be done without hardware by using either "native" simulation environments or "instruction-set" simulators:

- Native simulators are very fast and run system software directly on a workstation. Peripheral and GUI modeling, available on advanced native simulators, allows for a comprehensive test of system software.
- Instruction-set simulators (ISS) simulate processor instructions, thus giving a more accurate environment in which to run the embedded software but at a reduced speed. Coverification tools provide the most accurate simulation environment by linking a software simulator to a hardware simulator. The embedded software is tested against a full hardware simulation—although this further

reduces the execution speed. For larger systems, a hardware emulator can be used to accelerate execution.

The more that telematics systems enter our cars, the more automotive designers and suppliers must contend with embedded software and connectivity. Having the right set of software IP with a comprehensive set of verification tools ensures that our in-car systems make it to the showrooms on time.

## 1.7 Embedded Software for Transportation Applications

*Interest in automotive electronics and, hence, embedded software has escalated drastically in recent years, as such systems have become more extensive. Other forms of transportation make similar demands upon software, which is why embedded software is discussed in more general terms in this article, based upon a piece I did for* NewBits *in early 2003. (CW)*

The extent and complexity of electronics in all modes of transportation (automobiles, trains, airplanes) has been steadily increasing in recent years, and no end is in sight for this trend. Although, in essence, such systems have much the same characteristics as other embedded, real-time systems, a number of specific tools, technologies, and techniques have been developed. This article reviews the topic, covering programming techniques, with an emphasis on the appropriate use of C and real-time issues, including an introduction to OSEK/VDX.

### Introduction

Consideration of the nature of embedded software in automotive and transportation applications quickly reveals a key difference from other types of systems: failure really matters. Many such systems, perhaps not all, have a safety-critical element, and this factor necessitates an appraisal of the programming tools and techniques. We will initially consider the approach to writing the code, then move on to consider real-time issues and how they relate to the use of a real-time operating system (RTOS).

### Transportation System Characteristics

It is possible to identify some common characteristics of embedded systems used in transportation and automotive applications. It is very common for such systems to be distributed—multiple microprocessors or microcontrollers are deployed, with appropriate interprocessor communication facilities. For example, interaction in a car may occur between the engine management, antilock braking, and driver information systems. The devices employed may be of widely differing architectures, depending upon the functionality demanded of them; they could be 8, 16, or 32 parts and may be standalone processors or highly integrated microcontrollers. This diversity has significant impact upon programming techniques and RTOS selection. Both of these issues will be considered further.

Almost all transportation-oriented embedded systems can be described as "real-time," a common definition of which is as follows: "A real-time system is one in which the correctness of the computations depends not only upon the logical correctness of the computation, but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred." In other words, a real-time system may not be fast (although it often will be)—its key characteristics are predictable response and reliability. This definition also implies that system failure is not unacceptable, but it should be anticipated and its behavior well defined. These parameters place constraints on the approach to the system design and limit the

selection of a real-time operating system, which will be addressed in more detail in the sections that follow.

### Programming Issues

Many matters must be addressed before planning the development of software for a safety-critical system. The first thing to consider is the choice of programming language.

It is widely accepted that a high-level programming language, as opposed to an assembly language, not only enables the programmer's time to be used more efficiently, but will most likely result in more accurate, reliable, and maintainable code. The choice is likely to be limited to C, C++, and Ada, at best. Most other languages are either unavailable for the type of processor used in such an application or would be unsuitable for other reasons. As we have seen, a given project may require programming multiple devices of widely differing architectures.

All embedded systems tend to have limited resources (particularly memory and CPU power), but, with many transportation and automotive applications, this limit is even more acute, as cost issues are often paramount. Processors are selected that are just powerful enough and sufficient memory (but no more) is provided. The result of such limitations tends to eliminate C++ and Ada because their demands on resources tend to be too great.

The result is that almost all such systems are programmed in C. This is a concern because C is not really 100% suitable for such applications. It can generate compact, fast code, which is good. It may also be written in a clear maintainable style, if the right guidelines are followed. However, being a very powerful language means that it is possible to write "dangerous" code, which could produce unpredictable results. Also, the definition of the language standard contains ambiguities, which can also lead to unexpected behavior. These two factors indicate the need for some very firm guidelines if C is to be used successfully.

It is useful to use a "style guide" to ensure that code is written in a consistent manner within an organization. There are many publications that address this topic. The need for guidance in writing safer, more secure code is much more than a question of a style and methodology. Using the complete C language is very inadvisable, and the use of a carefully selected subset is the best approach. Books on writing "safer" C are available (see the "References" section that follows), but the clearest and most concise guidance is provided by the Motor Industry Software Reliability Association (MISRA), who have specified "MISRA C," a well-defined C subset, avoiding all the major pitfalls. MISRA C is really just a set of 127 rules; 93 of these are "required," and the remainder are "advisory." There are accommodations for "deviations," but the rule set is well suited to real programming situations, based upon actual experience in developing safety-critical systems. Applying MISRA C does not put any particular constraints upon the choice of programming tools. Any suitable

C compiler which supports ISO/ANSI C and includes accommodation for embedded systems requirements, is acceptable. Various tools are also available to analyze MISRA C compliance.

## Real-Time Operating System Factors

An understanding of the characteristics of embedded systems for transportation applications yields some definitive requirements for an RTOS. It is also worthwhile to consider available options.

### *RTOS Requirements*

We can highlight key features required for an RTOS used for this type of application:

- **Statically defined data structures:** Most RTOSs on the market offer the capability to create objects dynamically. This functionality is rarely required in practice and provides a potential opportunity for failure and unexpected behavior. An RTOS with objects defined at build time would be more suitable.
- **Multiple processor architecture support:** Since transportation systems tend to be distributed, with many different microprocessor architectures employed, it is useful to consider an RTOS that is supported on all (or as many as possible) of the devices. This simplifies the selection process and staff deployment.
- **Interprocessor communications:** An RTOS that incorporates a well-defined interprocessor communication mechanism is attractive for distributed systems.
- **Certifiability:** Systems used in safety-critical applications commonly require certification by relevant authorities. Using an RTOS which is known to be certifiable reduces risk. It is not possible to have an RTOS that is itself "certified."
- **Standard API:** If the RTOS supports a standardized application program interface, the learning curve is shortened and staff deployment is simplified.

### *OSEK/VDX*

The OSEK/VDX standard was developed by a group of European automobile manufacturers and is finding broad acceptance across the industry worldwide. The standard addresses most of the issues previously identified but goes further, including standards for:

- The operating system (OS)
- Communications (COM)
- Network management (NM)
- OS implementation language (OIL)

The OSEK/VDX standard has been implemented for a wide variety of chips by various RTOS vendors. The application of this standard permits coding to a standard API and methodology, which results in portability of both code and expertise. Details may be found at www.osek-vdx.org.

## Conclusions

The successful development of embedded systems for transportation applications depends upon many factors. Understanding the architecture and unique demands of such systems is a start. Care with the code-development process is essential, and MISRA C is a useful guideline for the use of this language. Real-time issues are also important, and the OSEK/VDX standard for RTOSs directly addresses the requirements of this kind of application.

## References

"Guidelines for the Use of the C Language in Vehicle Based Software," the original version of MISRA C. Publication details are available at www.misra.org.uk.

"Safer C" by Les Hatton; McGraw-Hill (1995).

"Programming in the OSEK/VDX Environment" by Joseph Lemieux; CMP Books (2001).

### Examples of the 127 MISRA C Rules

#### Advisory Rules

**Rule 66:** "Only expressions concerned with loop control should appear within a `for` statement." Good style, which leads to more readable code.

**Rule 86:** "If a function returns error information, then that error information should be tested." Good practice, but checking input parameters is more effective.

**Rule 93:** "Function should be used in preference to a function-like macro." Macros have no type-checking and are, hence, less secure. Many compilers will inline small functions anyway to improve speed.

#### Required Rules

**Rule 21:** "Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier." Good practice for clear code.

**Rule 71:** "Functions shall always have prototype declarations and the prototype shall be visible at both the function definition and call." A common-sense rule that should be followed everywhere.

**Rule 76:** "Functions with no parameters shall be declared with parameter type `void`." Although the syntax may be a little odd, this avoids ambiguity. It lines up with a rule that disallows variable numbers and/or types of arguments (Rule 69).

## 1.8 How to Choose a CPU for Your System on Chip Design

*The choice of CPU for any embedded design is interesting, but often surprisingly arbitrary. Stephen Olsen wrote a paper considering the matter in the context of an SoC design, where many of the factors are particularly critical, but mostly turn out to be just as applicable to conventional, board-based designs. This article is based upon that paper. (CW)*

There are several factors to consider when choosing a CPU for your next system on chip (SoC) design. If you consider that the CPU is to the SoC what an engine is to an automobile, you would not put a Volkswagen engine into a Hummer and expect it to perform. Similarly, a Ferrari engine would also be unsuitable in a Hummer. Although it may deliver similar horsepower to the Hummer engine, it would fail due to a lack of torque. Simple assessments of "horsepower" are just as misleading in CPU selection as they are in the automobile world. There is an optimal solution for the desired functionality. The same holds true for the CPU choice in an SoC. Many times the CPU is chosen based purely on the system architect's knowledge of, and past experience with, a particular device. The decision of which CPU to use should also consider the overall system metrics: complexity of overall design, design reuse, protection, performance, power, size, cost, tools, and middleware availability.

### Design Complexity

The design's complexity is critical to the choice of CPU. For example, if the design calls for a single-state machine to be executed with interrupts from a small set of peripherals, then you may be better off with a small CPU and/or microcontroller such as the 8051 or the Z80. Many systems may fit this category initially. An example might be a pager. The memory footprint is small, the signal is slow, and battery consumption is required to be extremely low.

The algorithms and their interaction will dictate the complexity. They may or may not also dictate the need for an RTOS. Typically, as the application complexity increases, the need for a greater bit-width processor increases.

### Design Reuse

Designs are continuing to be reused and are growing in complexity; that pager designed in 2000 may have to be upgraded to play MP3s in 2005. Now the 8-bit CPU may not be enough to keep up with the task at hand. How many interfaces a design contains is a good indicator of the amount of processor power required. In our pager example, initially there were two main interfaces: the user interface and the radio link. For the new design, which adds an MP3 player, we will need to add a memory interface for storing and transferring the data, and an audio interface for playing the data. Now the system complexity is greatly increased from its initial conception, and if we have taken a forward-looking approach to the design, we can reuse much of this earlier work.

Make sure that you have room for growth. Today your 8-bit design may be good for the MP3 player, but when the design gets reused and placed in a set-top box application,

which has a much higher bandwidth peripheral set, you may need to reengineer the complete solution to migrate to an ARM-, MIPS-, or PowerPC-based architecture to deal with the new constraints.

### Memory Architecture and Protection

The system may need to protect itself from outside attack or even from itself. This causes us to look at CPUs that include (or can include) memory management units (MMUs) to address this issue. Virtual memory will allow trusted programs access to the entire system, and untrusted ones to access just the memory they have been allocated. A 3-G cell phone—a phone with Internet connectivity—is a prime example of the need for protection. No longer can you use a CPU that lacks an MMU, since a rogue program will crash your phone. Although an MMU does not eliminate the possibility of system crashes, it reduces the likelihood of hard-to-resolve system failures.

Three main CPU architectures center around 8-, 16-, and 32-bit data registers with 16-, 24-, and 32-bit address buses. The main difference between these CPUs is how much information one particular register can hold and how much it can address directly:

- 8-bit data/16-bit address = (0 . . . 256), with 64-K address space
- 16-bit data/24-bit address = (0 . . . 65536), with 16-M address space
- 32-bit data/32-bit address = (0 . . . 4 billion), with 4-G address space

Why would an embedded system ever need to access 4 G of address space? The answer is simple: as the system is asked to perform more complex tasks, the size and complexity of the code it runs increases. In the early days, CPM on a Z80 utilized a process of banking memory and page swapping in order to run more complex programs on an 8-bit machine. Space of 64 K was not sufficient, and a solution was to make the system more complex by overlaying memory and pages to get more out of the CPU.

It seems like a 24-bit address bus would be adequate for many designs. A couple of factors drive us to a 32-bit address space: protection and pointers. For protection, the CPU with virtual memory can use the entire address range to divide up the physical memory into separate virtual spaces, thus providing protection from bad pointers. And the ability for any register to become a pointer to memory without the need for indexing simplifies the software.

### CPU Performance

The performance of the overall system will be greatly impacted by the selection of CPU. Specifically, features like cache, MMU, pipelining, branch prediction, and superscalar architecture all affect the speed of a system. Depending on the needs of the SoC, these features may be necessary to achieve system performance.

### Power Consumption

The end use of the SoC will determine how much power your design can consume. If your design is battery operated, the CPU will need to be as power conscious as possible.

For instance, some CPUs have the ability to sleep, doze, or snooze. These modes allow the CPU, when idle, to suspend operation and consume less power by shutting down various parts of the CPU. Different CPUs perform the same task with different results.

### Costs

The cost of the CPU can be measured in several ways. First the intellectual property (IP) cost, which is the cost to acquire the IP for your SoC and any derivative products. Then there is the system integration cost. Which tools are available for design and implementation of your SoC? Finally, is the CPU variant silicon proven, and is it available on the bus architecture that your SoC is utilizing?

### Software Issues

The availability of an RTOS and middleware may dictate your choice as well. For instance, in designing a PDA, you may want the middleware that is available for Linux, but the choice of a virtual operating system will dictate that you migrate away from small non-MMU CPUs.

Is there a graphics system or a file system necessary in the design? If so, then the choice of RTOS will dictate the type of CPU that is needed as well. Many RTOS vendors target specific families, leaving others untouched. Most 8-bit CPUs have simple schedulers that are adequate for small designs that utilize little outsourced code. They are not likely to be adequate for designs that consume any quantity of outsourced code. The outsourcing of the solution will strongly influence the RTOS choice, which, in turn, dictates what types of CPU are possible.

The tools necessary to do the design: are they available for the standard ANSI C/C++ compiler that you may use? How will you debug your design, either in the hardware/software cosimulation environment or on the SoC after it exists? Does a JTAG port exist, and is the CPU using this channel for debug, or is a dedicated serial port necessary? The choice of a higher level language like C++ or code generated from a design in UML may also dictate the need for a higher bus width and clock frequency to deal with the code size and complexity.

### Multicore SoCs

The SoC may be better off if partitioned into several processor subsystems that communicate via a loosely connected FIFO or serial channel. Many designs incorporate a DSP (digital signal processor) and a RISC CPU to share the workload and simplify the design of each processor domain. But this further complicates the CPU choice, which may now be multiplied several times over.

### Conclusions

Modern SoC design has presented new challenges for the system architect. No longer is the choice of CPU trivial. Utilizing metrics such as the complexity of overall design, design reuse, protection, performance, power, size, cost, tools, and middleware availability can simplify the decision.

## 1.9 An Introduction to USB Software

*The success of USB on the desktop is undeniable and has led to a surge of interest in its incorporation into embedded devices. This interest, in turn, resulted in numerous questions about just how USB worked and how it might be implemented. In response, C.C. Hung and I wrote a tutorial in* NewBits *in the summer of 2004. This article is based on that tutorial. (CW)*

However you look at it, USB is a good thing.

If you have worked with PCs for a few years, you will remember how it used to be. You would buy a new peripheral device, spend lots of time and effort ripping apart your PC to install the card, and then start worrying about software. It took forever. By the time you'd finished, any excitement you had about this cool new device had long since evaporated.

USB changed all that. Nowadays, you just plug a standard USB cable into the back of the computer and into the device and switch on. Sometimes the computer will take a moment or two to figure out drivers and so forth, but in no time, you are up and running.

USB highlights an interesting phenomenon in the high-tech world: the simpler something is to use on the outside, the more horribly complex it is on the inside! We will take a look at how USB works, and you will see what this means.

### What Is USB?

USB was designed as an alternative to replace the plethora of different serial and parallel interfaces used to connect peripherals to PCs. It is a single standard, which minimizes the number of different interfaces, cables, and connectors. The whole input/output system is simplified, and it offers the potential for real plug and play.

There is also plenty of capacity. A USB system may support up to 127 devices. A wide range of performance options are available to support a selection of requirements. The original USB specification allowed 1.5 Mb/s or 12 Mb/s; the latest spec increases this by up to 40X.

The structure of a USB system is hierarchical. It is termed a "tiered star" topology. A USB host includes the "root hub," which forms the nexus for all device connections. The host controls the bus, and there may only be a

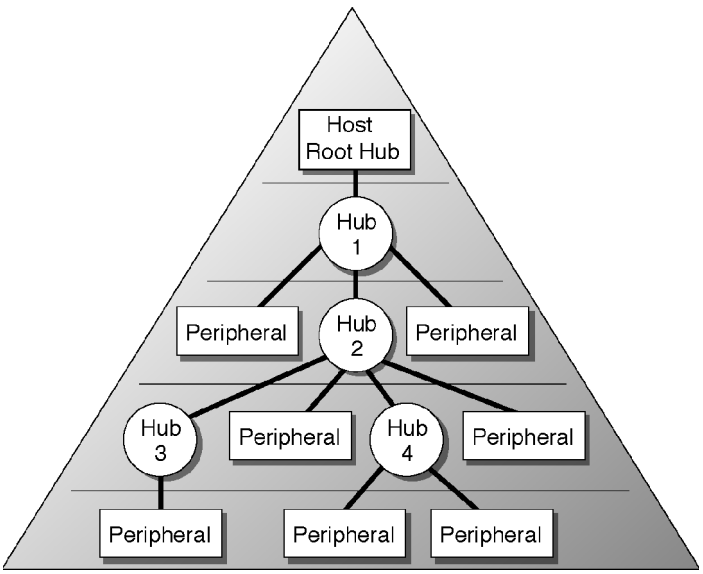



*Figure 1.8: USB topology*

single host in any USB system (see Figure 1.8). Hubs have a single upstream connection but may have many downstream. Hubs increase the logical and physical fan out of the network. "Peripherals"—the devices controlled by the host or hub—are also referred to as "functions" in the USB world.

### A USB Peripheral

The typical architecture of a USB peripheral device is shown in Figure 1.9. The USB peripheral controller is interfaced to the controlling microprocessor or microcontroller. This, in turn, is interfaced to the rest of the electronics of the peripheral device.

Communication to and from the host or hub is transmitted serially in the form of frames. All communications are sent and received via addressable buffers called "endpoints."

Transfers take place through logical channels called "virtual pipes," which connect the peripheral's endpoints with the host. Each connection always has a control pipe (endpoint zero) and one or more data pipes (endpoint 1, 2, etc.), which may be configured as "IN" or "OUT" endpoints. IN denotes device to host communication; OUT is host to device.

### USB Communications

When establishing communication with the host, each endpoint returns a descriptor. This is a data structure that tells the host about the peripheral's configuration, protocol, message transfer type, packet size, data transfer interval, and so on. USB is a pure master/slave architecture. The host initiates and controls all communication. The host sends a control packet to the peripheral device for "enumeration"—the process in which the peripheral describes itself to the host. The host chooses which peripherals are allowed to connect.

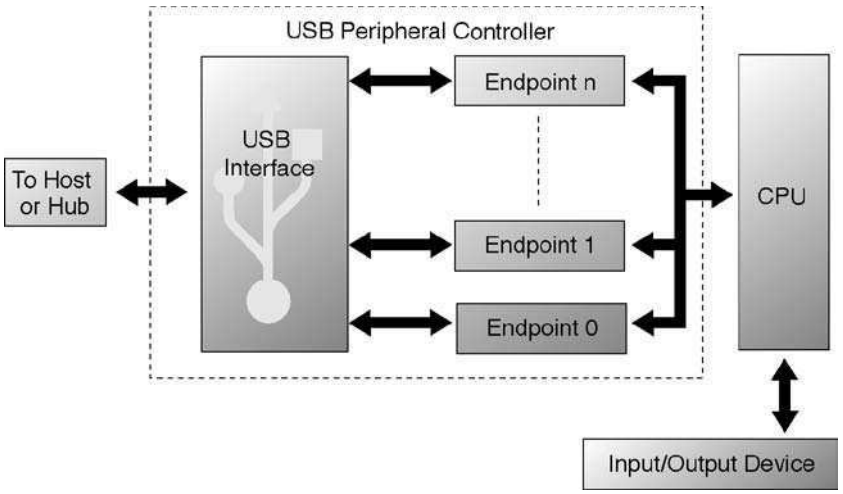



*Figure 1.9: A USB peripheral*

Four types of data transfer are supported:

- **Control transfers** exchange configuration, setup, and command information between the host and device. This is required for every connection and is associated with endpoint zero.
- **Bulk transfers** move large amounts of data, when timely delivery is not critical. There is no guaranteed throughput. Typical applications include printers and scanners.
- **Interrupt transfers** are confusingly named, as they have nothing to do with interrupts in the CPU-diverting sense. They are used to poll devices to ascertain whether they require service. Mice and keyboards are typically handled this way.
- **Isochronous transfers** handle streaming data and offer a guaranteed throughput. They are typically used for audio and video applications.

## USB Software

From a software perspective, the implementation on host and peripheral—or function or device, as you prefer—is quite symmetrical and comprises a series of layers, as shown in Figure 1.10.

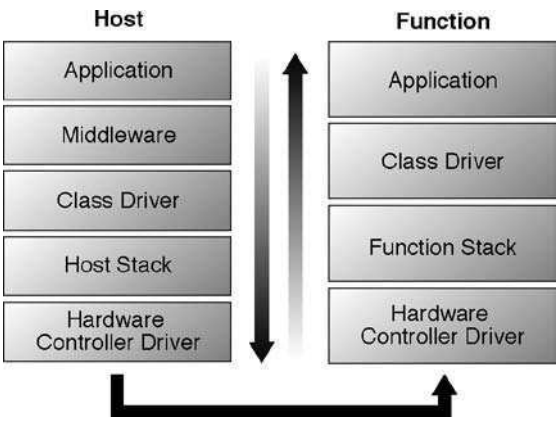



*Figure 1.10: USB system architecture: software layers*

As with most communications protocols, USB can most easily be described as a series of discrete layers:

- At the top is the **application.** On the host, this is the program that is utilizing the device. In the function, this is the embedded software that controls the device.
- **Middleware** enables the easy integration of the application with the underlying components.
- **Class drivers** implement capabilities such as storage, printing, audio, imaging, and human interface. They provide the real *characterization* of the device. They may be implemented according to the USB standard specifications, or they may be vendor-specific. This will depend upon the kind of device being supported. Here's some useful advice: if you are building a USB device, making it look like a standard peripheral—complying with a standard class driver—will make life much easier, obviating the need to develop host (Windows) drivers.
- The **Stack** layer processes the USB bus protocols and manages all data transactions for devices on the bus.
- The **Controller driver** layer manages USB device power, enumeration, and other low-level control functions.

## USB and Embedded Systems

USB is generally thought of as being a means to connect devices to a PC. In this case, the PC is the USB host and the devices or peripherals are USB functions. But we are interested in embedded systems, which requires a different kind of perspective.

The obvious application of USB in an embedded system is when you are building a device—a peripheral that may connect to a PC. This is a USB *function* and requires software support for that end of the bus. Examples of USB functions include hard drives, printers, audio devices, medical equipment, cameras, keyboards, mice . . . the list just goes on. You name it nowadays, and it has a USB interface. There are times when the USB *host* need not be a PC and is an embedded system of some kind. Possibilities here include set-top boxes, point-of-sale devices, and health-care monitoring systems. Of course, the interface board that goes in a PC could well be an embedded system in its own right. These are USB hosts and need software support for that end of the bus, which, as we have seen, has distinct differences from the function end.

It is possible to identify examples of embedded systems that may act as both USB host and function, at different times. The example shown in Figure 1.11 is a camera, which may be a function, when a PC is the host, for uploading pictures. It can be the USB host when it connects directly to a printer to get a hard copy of images. In this context, the camera is actually supporting two separate USB systems. It needs software support for both ends of the bus.

Recognizing this kind of situation, an extension to the USB 2.0 specification is emerging. This extension is called "USB On-the-Go" or "OTG." This capability enables a device to behave as a limited USB host and connect to other USB functions. The primary limitation is the inability to support hub functionality. It is ideal for portable devices, such as our example. For this to work, both the devices must be OTG-compatible.

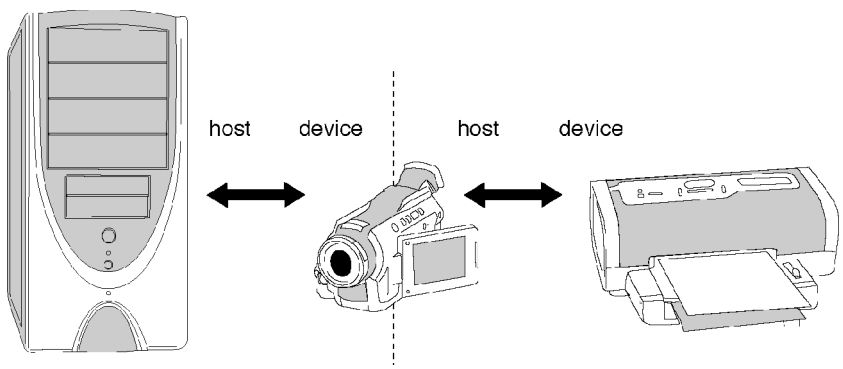



*Figure 1.11: Both sides of the bus*

## Conclusions

Although familiar to almost any user of a modern computer, the underlying functionality of USB is not widely understood. It is well defined, but complex. Its successful application requires absolute adherence to published standards, which may most readily be achieved by licensing a commercial embedded USB stack. The standards are evolving rapidly, as higher speed interfaces become standard and the OTG standard becomes widely accepted.

## 1.10 USB On-the-Go

### Its Use in System on Chip Designs

*With the surge in interest in USB, I was very pleased to receive this article from Richard Vlamynck (a consultant at Nebula E.D.A.). His article puts the new USB OTG specification into perspective. In addition, Richard takes the opportunity to highlight the close relationship between embedded software and the various technologies employed in electronic hardware design and verification. (CW)*

#### What Is USB OTG?

USB OTG is "Universal Serial Bus, On-the-Go." It is the latest specification from usb.org that has evolved from the original PC-centric specification. The original USB specification was tuned to applications that were usually centered around a personal computer.

USB, as first defined, envisioned multiple data pipes from many different types of peripheral devices all connected to a single master controller called the "host." The host controller is the bus master at all times. The scheduling and synchronization of all data transfers are initiated by the PC-centric host controller.

The OTG specification extends USB to less PC-centric application domains.

#### Why is USB OTG such a hot topic today?

The convergence of digital instrumentation coupled with the desire to remove the need for the desktop PC to perform many tasks has created an opportunity for USB OTG. The first and simplest example that comes to mind is the ability to print digital photographs directly without the need to use some kind of personal computer.

#### Many devices already have serial ports and parallel ports; why can't I just use those ports to connect devices together? Why do I need USB?

USB offers standardization so that the different devices have a protocol that they all understand. If not for the USB protocol, then you would have to write a custom driver for every combination of devices that you need to connect. For example, if you were building digital cameras and had three models of your camera that you wanted to connect to two different models of printers from three different vendors, then, at worst case, you would need to write $3 \times 2 \times 3 = 18$ different device drivers. With its standard protocol, USB means that you only have to write a device driver for your own device.

#### What makes up a USB OTG device?

There are two halves to a USB OTG device, the hardware and the software.

The hardware consists of at least three major components: a USB controller core that handles the USB protocol, a USB PHY to transmit and receive the signals, and a microcontroller to run the USB software protocol stack. The microcontroller includes its associated RAM, flash-ROM, and bus structure.

The software consists of at least three major components: a high-level application program that knows what to do with the data being sent and received over the USB bus, a USB OTG software protocol stack with device drivers that can program the USB controller, and a real-time operating system (RTOS) to synchronize and schedule application programs.

It is worth noting that earlier, simple USB devices frequently used 8-bit microcontrollers and sometimes needed no RTOS. That is no longer the case with the higher speed and more complex applications targeted for USB OTG. You will most likely find a 32-bit microprocessor controlling a device in this context.

### How do you build custom hardware for a USB OTG device?

There are three different ways to build the hardware. The first way is to select commercial off-the-shelf USB hardware components and design a custom printed-circuit board that uses those components. The second way is to create a system based around a field-programmable gate array (FPGA). The third way is to build a custom system on chip (SoC).

### What is a custom SoC?

When you are building a custom SoC, it means that you are going to create your own chip design, have it fabricated on silicon wafers, and have it packaged in ceramic or plastic suitable for use on a printed circuit board. You will have to select a wafer foundry and a target process at that foundry.

### How do you build a custom SoC that incorporates USB OTG?

You will need to either create or select the hardware and the software for your device.

The software can be purchased from an RTOS vendor, or you can write it yourself. The pros and cons of writing your own RTOS are discussed in other articles and publications (in Chapter 7 of this book, for example). The same is true of the protocol stack for USB OTG. In each case, it is conceptually simple to design and write your own RTOS or your own protocol stack, but the devil is in the details.

The three main hardware components of your USB OTG-enabled device are collectively termed the "CPU subsection." Therefore, you will need to either design or purchase a microprocessor, a USB controller core, and a USB PHY. For example, if you have an experienced design team, then you can get them to design a microprocessor for your SoC. If you do not have the design team, then you can simply purchase the design of a microprocessor from a "fabless semiconductor vendor." Similarly, you can either design the USB controller, or you can purchase a preexisting design (IP—intellectual property) from a third party.

This is an oversimplification of a very complicated subject, but at least it provides a good head start by revealing the tip of the iceberg.

### What are the advantages of using third-party IP in a custom SoC?

The primary advantages of using third-party IP are faster time to market and less verification effort. It takes several man-years of effort to create a viable microprocessor subsection complete with USB controller using Verilog or VHDL hardware description language. The verification effort will be at least as big as the design effort, but realistically the verification effort for the SoC will be twice as large as the design effort. Therefore, it makes good sense to use predesigned and preverified third-party IP from fabless semiconductor vendors when creating a custom SoC.

### How do I create an SoC using a third-party IP?

For the hardware, make a list of the vendors that offer microprocessor IP, USB controller IP, and the IP for the USB PHY. Select a set of hardware IP that meets your criteria and purchase that IP.

In a similar fashion, select an RTOS with full and complete support for USB OTG and purchase that RTOS.

### How much does a single mistake in SoC hardware actually cost?

A single respin of a production wafer will cost $400 K to $900 K in NRE engineering mask costs, depending on the wafer fab vendor and the process technology that you selected. The cost of lost "time to market" will depend on your application and market opportunity. For example, a three-month delay in a mid-range laser printer selling for $1,200 that ships 1000 units per month would be $3.6 million in gross revenue. If the lost time to market also includes the loss of becoming the first to market a new technology, then the loss of market share from not becoming the market leader could translate to the loss of tens of millions of dollars over the lifetime of that product.

### I cannot actually afford a million dollars on an SoC respin; how do I verify that the device will work correctly before I actually build the hardware?

The answer is that you have to fully validate the hardware design before "tape out." There are a few different ways to validate an SoC design such as emulation and formal verification, but the preferred method is called "hardware/software co-verification."

### What is hardware/software co-verification?

Hardware/software co-verification, or co-modeling, is the methodology of using a software test bench to augment the hardware test bench. In normal hardware design flows, the hardware design team performs exhaustive testing on the hardware on a module-by-module basis. Some large modules such as custom DSP (digital signal processor) blocks and USB controllers may take many hours or a few days to run exhaustive hardware verification tests. The verification test bench takes longer and longer to execute as more modules are added to the system-level test bench.

### How can my RTOS actually run in a co-modeling environment if I have not even built the hardware yet?

The secret to the co-modeling environment is to get a very accurate simulator for your microprocessor that is also capable of driving hardware bus cycles. Your hardware team has already created exact models of the hardware that is going to be in your custom SoC chip, and your software team has created, or is in the process of creating, the software that is going to run on the chip. So, if you had a simulator that could run your software on the hardware models, then the software is like another test bench that the hardware team gets for free.

But it is even better than that, because the hardware team usually does not have a system-level test bench that exercises the SoC in a fashion that is representative of the way the chip is going to run after it has been manufactured.

### Are you saying that the device drivers that I create for the co-modeling environment will run unmodified on the real silicon?

Yes. Among the many advantages of the co-modeling environment is that the software team can create the exact boot code, power on reset code, the RTOS startup code, and all the device driver code in the co-modeling environment exactly like it will be on the actual device. The code is the same; the timing is the same; the execution order is the same. You use the same software tools (compiler, assembler, and linker) to produce binaries for the co-model as you would for the real chip.

Using co-modeling, it is possible to create the entire software load binary for an SoC long before the chip is even built. The software team can have fully debugged software up and running before the chip comes back from the wafer fab.

### How can I find a good co-modeling environment?

An engineering design manager would be well advised to find a co-modeling environment that ranges across the full spectrum of performance versus model fidelity. This would signify a company that offers a range of simulation and emulation technology. More specifically, you should look for a company that offers these three main product lines:

1. RTOS software, networking and file systems, and especially USB protocol stacks
2. VHDL and Verilog simulators
3. Hardware IP such as microcontrollers and USB controllers

### How can I find a good commercial off-the-shelf RTOS and USB OTG protocol stack?

You should look for a package deal that includes not only the RTOS itself, but a good design and support team to back it up. You should look for a USB OTG protocol stack that has been tuned to match the RTOS. Ensure that you purchase a prequalified and pretested RTOS for your SoC. With today's global competition and extreme pressure against time to market, the days of "mix and match" and "do it yourself" are over.

# 2

# *Design and Development*

*Finding a logical flow for the chapters in this book proved challenging. I more or less gave up and simply grouped articles that seemed to belong together. Design and development methodologies and tools seem to fit alongside one another. But where should they all go in a sequence? After all, you do design first, but design methodologies are seen as "high level," so shouldn't the "low-level" stuff come before it? See what I mean?*

*What did prove interesting is that there are two "buzzwords" that are evoking a lot of interest in the embedded world at the beginning of the twenty-first century: UML and Eclipse. Both are addressed in articles in this chapter.*

## 2.1 Emerging Technology for Embedded Systems Software Development

*Crystal ball gazing is a hazardous occupation. No matter how well you know a technical subject, new developments will arise that you were unable to foresee. I wrote an "agenda setting" piece for* NewBits *in the late 1990s, and while reviewing it for use in this book, I was surprised at how much had "gone according to plan." This article for the 1990s needed surprisingly little adaptation to be developed into an article for the twenty-first century. The only key new technology is the UML, which is addressed in more detail in other articles in this chapter. (CW)*

It is easy to think of embedded systems development as state of the art and leading edge. However, since microprocessors were first introduced in the early 1970s and the business has been developing over 30 years—more than a quarter of a century—it is now a mature technology. By "mature," I do not mean "stagnant" or "boring." Embedded systems software development is far from boring. It is hard to identify any other business that is more dynamic, fast moving, and forward looking.

That maturity may be used to real advantage. After 30 years of growing, it is possible to identify a number of clear trends in the evolution of embedded systems development as a whole. Those trends point to the emergence of key technologies, upon which we may confidently focus to address the challenges ahead.

In this article, we endeavor to identify some of those trends and single out the technologies that they drive, resulting in an agenda for our attention over the coming months and years.

### Microprocessor Device Technology

The earliest microprocessors were 4- and 8-bit devices. As fabrication techniques became more sophisticated, integrated 8-bit microcontrollers began to appear and the first 16-bit microprocessors came into use. Once again, silicon technology moved on, and 16-bit microcontrollers were introduced and widely applied, as demand grew for more sophisticated embedded systems. Devices with 32-bit architecture gradually took hold in higher-end applications, and these too were complemented by highly integrated microcontrollers. The first 32-bit devices were all CISC architecture, but increasingly RISC chips are providing even higher performance.

It would be easy to interpret this "potted history" of the embedded microprocessor, as illustrated in Figure 2.1, as a description of a timeline: 8-bit micros were yesterday; 32-bit RISC is today. However, this is not the case. As the more powerful devices have become available and found application, they have not, for the most part, replaced the earlier parts but have augmented the range of options available to the designer. An embedded systems designer has a wider choice of microprocessors than ever before and must make a choice based upon functionality, specification, support, availability, and price.

This increasingly wide range of devices has a number of possible impacts on the software designer. Obviously, suitable programming tools must be available to support this array of processors; it is preferable that the tools are consistent from one device to another. More importantly, the necessity of migrating both code and programming expertise from one device to another is becoming commonplace. This need not present major problems. By careful code design, use of off-the-shelf components, and adherence to recognized standards, porting may be quite straightforward.

### System Architecture

As microprocessors have evolved, the architecture of the systems in which they are used has progressed as well. The earliest systems were comprised of the CPU and a selection of logic devices. More highly integrated devices reduced the chip count, and higher-performance devices presented many design challenges to the hardware developer. From the software engineer's point of view, nothing really changed. For many years, the same debugging techniques could be employed as the system became more complex: in-circuit emulation, on-chip debug, ROM monitors, and instruction set simulation. This situation began to change.

As embedded systems become more powerful, with ever-increasing levels of demanded functionality, many designers are taking a fresh look at their use of microprocessors and microcontrollers. In many cases, instead of following the obvious path of simply incorporating more powerful devices, an alternate choice is made: the application of multiple processors. This choice may be driven simply by a desire to distribute the processing power (which would be typical in a telephone switch, for example). Alternatively, an

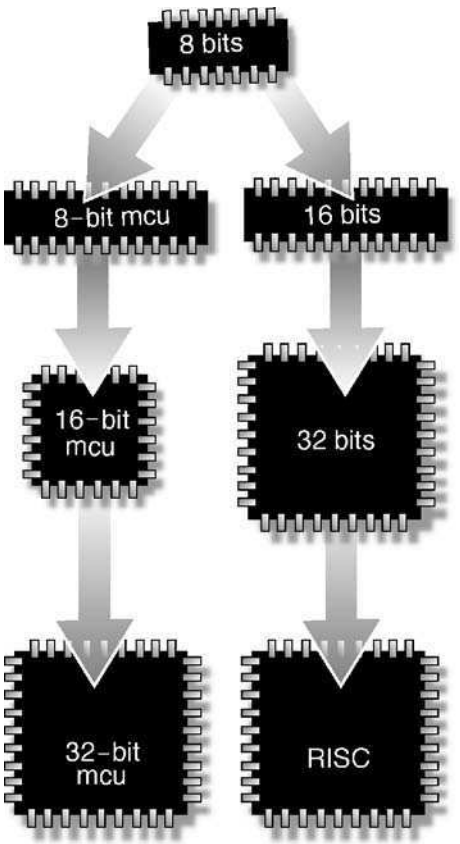



*Figure 2.1: Microprocessor technology*

additional processor may be added to provide specific functionality (e.g., a DSP—digital signal processor—in a mobile phone).

One of the biggest challenges faced by software developers when confronted with a multiprocessor system is debugging. It is, of course, possible to simply run one debugger for each device. However, that is not really addressing the problem. What is needed is the means to debug the system; the functioning of each processor and the interaction between them needs to be debugged. The requirement is for a debug technology that supports multiple processors in a single debug session, even when a variety of architectures are represented.

### Design Composition

In the earliest days of embedded systems, all of the development—both hardware and software design—was typically undertaken by a single engineer. The software element represented a small part of the entire effort: perhaps 5% to 10%. As illustrated in Figure 2.2, over time, the proportion of the engineering time dedicated to software development increased substantially. By the mid-1980s, this work was done by software specialists and comprised more like 50% of the development effort.

In the last few years, although hardware design has become more complex, the amount of software has grown drastically, now often being 70% to 80% of the total design effort. The result is that teams of software engineers are involved, and new challenges arise. Among these is the availability of hardware to facilitate software testing. Since more software needs to be developed (in a shorter time), an environment for testing is required sooner. Various solutions are available including native code execution prototyping environments, instruction set simulation, and the use of standard, low-cost, off-the-shelf evaluation boards. In addition, low-cost host-target connection technologies are becoming common, typically using a JTAG interface.

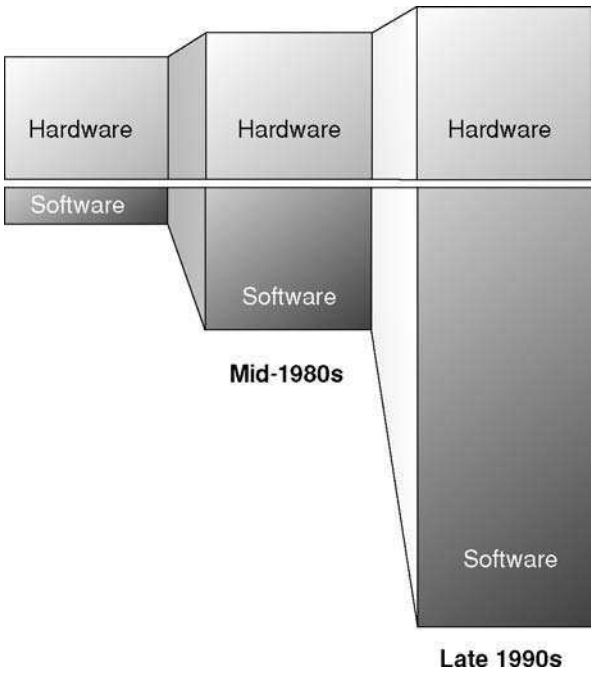



*Figure 2.2: Design composition*

This climate represents an ideal opportunity for hardware and software teams to work together. By using codesign and, in particular, coverification techniques,

software engineers can test on "real" hardware sooner, and the hardware designers are able to prove their designs earlier, with less prototyping cycles.

### Software Content

The proportion of development time dedicated to software has been increasing. Meanwhile, under pressure from worldwide trade and truly global competition, time to market has been decreasing. This has radically influenced the design strategy. The earliest designs were quite simple, being comprised solely of in-house designed applications code. As systems became more complex, a multitasking model was widely adopted for software development, and many developers opted for standard, commercial real-time operating system (RTOS) products.

As shown in Figure 2.3, the proportion of bought-in software, or "intellectual property" (a term borrowed from the hardware design world), has steadily increased, as further standards are adopted.

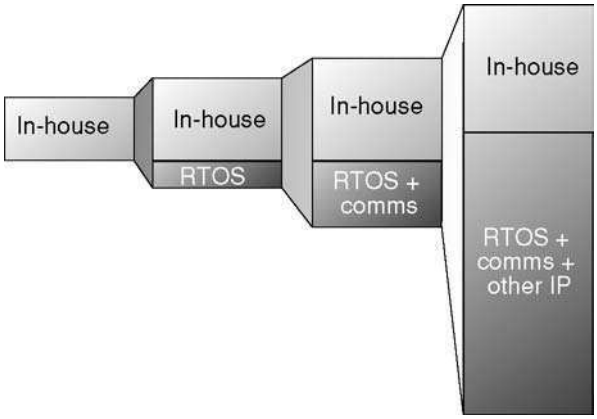



*Figure 2.3: Software content*

This trend has a number of implications for the software developer. The integration of standard software components—with the applications code and with one another—is a matter of concern. Debugging in a multitasking context is another issue. The business decision associated with the selection of intellectual property is particularly complex; future (e.g., migration to different processors) as well as immediate requirements must be take into consideration.

### Programming Languages

For the first 4- and 8-bit microprocessors, there was no choice of programming language. Assembler was the only option. Since the applications were relatively simple, this was not a big problem.

As 16-bit technology became viable, the need for a practical high-level language became apparent, and several options emerged. Pascal and C were both in use on the desktop, and these languages were adapted for embedded systems. Intel developed PL/M specifically for this kind of application. Forth was also very popular for certain types of systems. Over time, with the increasing use of 32-bit technology, the two languages that persisted were C and Ada. The latter is prevalent in defense-oriented systems.

It has been known for some years that C++ would start to replace C for embedded software development. Now, between one-quarter and one-third of embedded systems code

is written in C++. What was not anticipated a few years ago was the emergence of new languages and approaches, which are set to play a strong role in applications development in the future. The Java language was developed specifically for embedded applications and has found a niche where runtime reconfigurability is demanded. The Unified Modeling Language (UML) has become the most popular choice for a higher-level design methodology.

### Software Team Size and Distribution

As discussed earlier, the initial embedded system designs were one-man efforts. In due course, specialization resulted in engineers being dedicated to software development. The next step was the establishment of embedded software development teams. Managing software development is challenging in any context; embedded systems development is no exception and brings its own nuances. Using conventional programming techniques—procedural languages like C and assembler—most members of the team need to have a thorough knowledge of the whole system. As the team grows, this becomes less and less feasible. Typically, specific members of the team have expertise in particular areas. To manage the team effectively, a strategy must be in place that permits the encapsulation of their expertise. It must be possible for the work of an expert to be applied by the nonspecialist in a safe, secure, and straightforward manner. Object-oriented programming techniques find application in this context.

With many very large companies, the software teams are not simply growing; they are becoming distributed. Some members of the team are located at one site, while others are elsewhere. The sites may even be in different countries. This arrangement is common in Europe, where (spoken) language may be a concern. Elsewhere, time zones may be an issue (or an advantage, as a distributed team can work around the clock). This is increasingly the case as emerging technology centers (e.g., in India) are widely utilized. The need for reusable software components becomes even more apparent in this context.

### UML and Modeling

The UML has become a key design methodology in recent years, which goes hand in hand with increasing embedded software team size. There are broadly two ways to use a design tool: either as a guide to writing the actual code or as a means of generating the code directly. Code generation is controversial for embedded software, as it may be argued, quite validly, that every system is different and has very specific needs in this respect. This is where xtUML (executable and translatable UML) is attractive because it enables the application and architecture to be clearly separated. This follows the same philosophy as object-oriented programming—leveraging expertise through tools and technology.

### Key Technologies

All of these trends, which have become established over 30 years of embedded systems development, point to some key technologies:

**Microprocessor technology:** Leading to a proliferation in devices which involved the consideration of migration issues and writing portable code. That, in turn, drives a requirement for **compatible tools** and **RTOS products** across microprocessor families.

**System architecture:** Progressing so that multiprocessor embedded systems are becoming commonplace. This drives a requirement for a **debug technology** that addresses these needs.

**Design composition:** Changing, with a much greater part of the design effort being expended on software. This drives a requirement for **instruction set simulator technology** and **host-based prototyping** and the application of **on-chip debug facilities** and **hardware/software coverification**.

**Software content:** Moving from entirely in-house design to the wide use of intellectual property. This drives a requirement for **standards-based RTOS technology** and appropriate **debug technology**.

**Programming language:** Narrowing choices somewhat. Although a strong requirement for **C tools** still prevails, compatible **C++ products** are in strong demand.

**Software team size and composition:** Changing from one engineer (or less) to the employment of large, evenly distributed, teams. This drives a requirement for tools to support **object-oriented programming** and **RTOS technology with a familiar or standard API.** There is also an increasing demand for **modeling and design tools**.

## Conclusions

Tracking all the emerging technologies, which are driven by the ongoing trends in embedded systems development, is no easy task. Taking any one in isolation is also fruitless because of the many inter-relationships. For example, multitasking and multiprocessor debugging go hand-in-hand; standards-based RTOS technology is a real boon to processor migration; using a design methodology that flows naturally toward an implementation makes complete sense.

## 2.2 Making Development Tool Choices

*Developing embedded systems software is a complex matter, and the tools are necessarily complex. Although today the focus is heavily on integrated development environments, the selection of the tools to be used in that context is just as important as it was when I wrote a detailed review of the topic for* NewBits *in the mid-1990s. That piece is the basis for this article. Many of the topics highlighted here are expanded upon in other articles in this book, notably in Chapters 4, 5, and 7. (CW)*

This article is a review of available tools and techniques for program development in embedded systems, and it also discusses the implications of the availability of development tools on selection of a target microprocessor and real-time operating system. This article addresses the following questions regarding the selection process:

- What build tools will be needed?
- What features should be sought?
- What about the debugging parameters and options?
- What about tool integration?

### The Development Tool Chain

A useful way to view software development tools for this purpose is from the perspective of a tool chain, where each component forms a tight link to the next: from a high-level language, through the assembler, linker, and so on, to one or more debugger variants. (See Figure 2.4.)

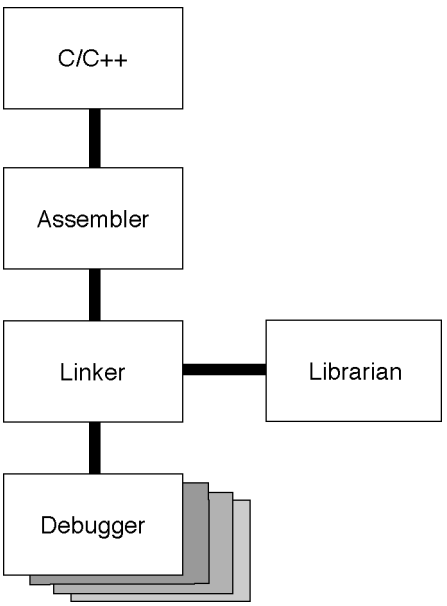



*Figure 2.4: The development tool chain*

There are two distinct parts to the tool chain:

- The body of the chain consists of the tools that take source code and generate an executable: the *build tools*.
- The base of the chain includes the tools used to execute and verify the executable program: the *debug tools*.

Almost without exception, the use of the build tools (options, controls, formats, etc.) should be quite unaffected by the proposed execution environment and the variant of debug tool employed. For example, it should not be necessary to build using a special library in order to use a particular debug tool. The clear requirement is to test exactly the same code at all stages of the development process.

The options, with respect to the execution environment, offered by debug tools are numerous. These options will be reviewed in turn, but let us first consider the build tools.

## Compiler Features

Most commonly, software for embedded systems is written in C. However, C++ is increasing in popularity, as object-oriented design becomes the norm. The main parameters governing the choice of cross-compiler are similar to those applied to native products, but other factors must be taken into consideration, such as:

- **Programming language accepted**
  The primary requirement in a compiler is that it accept the programming language in use. For both C and C++, full compliance with the ANSI specifications is essential.

- **Libraries provided**
  An ANSI-compliant C compiler need not, according to the specification, include a full runtime library. In reality, such a library is very useful, and its absence would hinder efficient program development. Unfortunately, a number of the standard library functions, as specified by ANSI, are intrinsically nonreentrant, which may be a problem for some embedded system designs. Because particular demands may be placed on library code by an embedded system, access to the library source code is particularly desirable.

  A common reason for using C++ is to facilitate code reuse and to be able to employ standard class libraries. It is, therefore, reasonable to expect a compiler to be supplied with such a library as standard.

- **Build tools that support an entire microprocessor family**
  Typically, an engineer selects a cross-compiler to support development for a specific target microprocessor that will be used for the current project. It is quite likely that future projects will use a different device, but it is commonly another member of the same family. With this in mind, choose build tools that support an entire microprocessor family. This support should, of course, go beyond the generation of code for the "baseline" device. For example, a Freescale 68000 family toolkit should feature specific code-generation capabilities for, say, 68332, 68020, and 68040 and not generate "plain vanilla" 68000 instructions at all times.

- **Manufacturer support**
  Beyond the technical requirements of the build tools, it is at least as important to look at the "pedigree" of the build tools: consider the reputation of the company who produces them, their technical support facilities, and the size of the current user base.

## Extensions for Embedded Systems

A cross-compiler is intrinsically a more complex tool than its native equivalent. This primarily comes about because very few assumptions about the target environment may be made by the compiler developer. To maintain the appropriate level of flexibility, the compiler manufacturer must implement a number of special features.

In particular, embedded systems almost always have complex memory configurations. The simplest have read-only memory (ROM) for code and constant data and random access (read/write) memory (RAM) for variable data. To accommodate this, the minimum required of the compiler is the generation of ROMable code, with the data clearly separated from the code. In most systems, a greater degree of control is needed, and being limited to this simple memory model would be a serious restriction.

A further implication of the memory structure of embedded systems is a clash with a language construct in C. In C, a static variable may be given an initial value. This was intended to avoid the necessity for initialization code for variables whose location in memory could be predicted at compile time. The intention was that such variables would be preset to their starting value in the executable file (memory image) on disk and loaded into memory with the program. For an embedded system, where the program is already in ROM, this mechanism does not work. This situation has three possible outcomes:

- Static variables cannot be initialized.
- Initialized statics can only be used as constants because they must be stored in ROM.
- The build tools must readily accommodate the copying of data from ROM to RAM at startup.

Since the C and C++ languages permit direct access to specific memory addresses, these languages are often useful for embedded system development, particularly for code that is closely associated with the hardware. Naturally, the compiler should not restrict this capability in any way.

As for assembler code, even though nowadays a high-level language is almost always chosen for software development, it is inevitable that, at some time, the programmer will write some assembler code. The use of assembler code may be necessary to permit the programmer to extract the last ounce of performance from the target chip, but, more likely, the programmer uses assembler code to access some microprocessor facility that does not map into C (for example, enabling or disabling of interrupts). In the interests of efficiency and code portability, the use of assembler should be minimized, and the facilities for its development should be as flexible as possible. The ability to write a complete assembler module should be augmented by the means to include one or more lines of low-level code among the C language.

### *Impact of Real-Time Systems*

The majority of embedded microprocessors are employed in real-time systems, and this puts further demands on the build tools. A real-time system tends to include interrupt service routines (ISRs); it should be possible to code these in C or C++ by adding an additional keyword `interrupt` declaring that a specific function is an ISR. This performs the necessary context saving and restoring and the return from interrupt sequence. The interrupt vector may usually be defined in C using an array of pointers to functions.

Furthermore, in a real-time system, it is common for code to be shared between the mainline program and ISRs or between tasks in a multitasking system. For code to be shared, it must be reentrant. C and C++ intrinsically permit reentrant code to be written, because data may be localized to the instance of the program (since it is stored on the stack or in a register). Although the programmer may compromise this capability (by using static storage, for example), the build tools need to support reentrancy. As mentioned previously, some ANSI-standard library functions are intrinsically nonreentrant, and some care is required in their use.

Since memory may be in short supply in an embedded system, cross-compilers generally offer a high level of control over its usage. A good example is a language extension supporting the `packed` keyword, which permits arrays and structures to be stored more efficiently. Of course, more memory-efficient storage may result in an access time increase, but such trade-offs are typical challenges of real-time system design.

One of the enhancements added to the C language during the ANSI standardization was the `volatile` keyword. Applying this qualifier to a variable declaration advises the compiler that the value of the variable may change in an unexpected manner. The compiler is thus prevented from optimizing access to that variable. Such a situation may arise if two tasks share a variable or if the variable represents a control or data register of an I/O device. It should be noted that the `volatile` keyword is not included in the standard C++ language. Many compilers do, however, include it as a language extension. The usefulness of a C++ cross-compiler without this extension is dubious.

### Optimizations

All modern compilers make use of optimization techniques to generate good quality code. These optimization techniques can result in software that rivals handcrafted assembler code for size and efficiency. Optimizations can be local to a function or global across a whole module.

Many optimizations may be applied in a generalized way, regardless of the target. More interesting are those that take specific advantage of the architectural characteristics of a specific microprocessor. These include instruction scheduling, function inlining, and `switch` statement tuning.

Instruction scheduling is a mechanism by which instructions are presented to the microprocessor in a sequence that ensures optimal usage of the CPU. This technique is a common requirement for getting the best out of RISC architectures. However, CISC devices can benefit from such a treatment.

The inlining of functions is the procedure whereby the actual code of a (small) function is included instead of a call. This is useful to maximize execution speed of the compiled code. Some compilers require specific functions to be nominated for this treatment, but automatic selection by the compiler is preferable. The optimization can yield very dramatic improvements in runtime performance.

In C, `switch` statements lend themselves to optimal code generation. Depending upon the values and sequence of the `case` constants, quite different code-generation techniques may be appropriate. Explicit tests, lookup tables, or indexed jump tables are all possible, depending upon the number and contiguousness of the `case` constants. It may even be efficient to generate a table with dummy entries if the constants are not quite contiguous. Since the compiler can "rewrite" the code each time it is run, efficiency rather than future flexibility can be the sole priority. This example gives a compiler a distinct advantage over a human assembler code writer.

Manufacturers of development tools for embedded systems have very limited knowledge of the architecture of individual configurations—every system is unique. As a result, fine control over the optimization process is essential. At a minimum, there should be a user-specified bias toward either execution time or memory usage.

### Build Tools: Key Issues Recapped

In selecting build tools for embedded system software development, consider these two key issues:

- Do the tools provide extensive accommodation for the special needs of embedded system development?
- Does the compiler perform a high standard of optimization, with extensive user control of the process?

### Debugging

Having designed and written a program and succeeded in getting it compiled and built, the programmer's next challenge is to verify the program's operation during execution. This is a challenge for any programming activity and never more than when working on an embedded system. In this context, many external influences on the debugging process and stringent requirements dictate the selection of tools.

#### *Debugger Features*

Some debugger features are desirable or vital in any context; others are specific to embedded systems work. We will concentrate on the latter.

A key capability of a debugger is the ability to debug fully optimized code. Although this sounds quite straightforward, it is not a facility offered by all debuggers. Often, there is a straight choice: ship optimized code or fully debugged code. It is common to select a microprocessor on the basis of its performance and to rely upon the compiler to deliver this performance. This is particularly true of high-performance RISC devices. It is unacceptable to be limited by available debugging technology.

In reality, debugging fully optimized code may be challenging for the programmer. The results of some optimizations (e.g., code motion and register coloring) can make it difficult to follow the execution process. It is, therefore, common to perform initial debug-

ging with optimization "reigned in." However, for the final stages of testing, the debugger should not preclude the use of maximum optimization.

Programmers write software in a high-level language (usually C or C++) primarily in the interests of efficiency. The debugger should pursue this philosophy fully and operate entirely in high-level terms. Code execution should be viewed on a statement-by-statement basis; line-by-line is not good enough. Data should be accessible in appropriate terms. Expression evaluation, structure expansion, and the following of pointers should all be straightforward. On the other hand, low-level access to code and data should also be available, when required.

C++ presents additional requirements: function names should always be shown "unmangled" and constructors and destructors should be visible, for example.

Since the suppliers of tools for embedded systems development cannot predict exactly what a given embedded system is like, they are unable to predict the precise functional requirements of the debugger. In the same way as with the build tools, this problem may be circumvented by providing enough flexibility to the user. For a debugger, this flexibility is manifest in the availability of a powerful scripting language. This might permit I/O device modeling, test automation, and code patching, for example.

The user interface of a debugger is of primary concern, because its design can directly affect the user's productivity. It must be powerful, consistent, and intuitive, which is particularly important when debuggers are to be used in a variety of execution environments. It is clear that if a single debugger family can fulfill all the differing requirements, hours of operator training time can be saved.

### *Development Phases*

Before considering how code debugging can be performed, it is useful to review the total development cycle of the embedded system. This process can be divided into five phases, as illustrated in Figure 2.5. At each phase, work on the software may progress, but the scope for progress and the techniques employed change as the system development continues.

In phase 1, although the system hardware is undefined, initial work developing algorithms and trying ideas can proceed. At this stage, it is wise to train the engineers who are going to use the debugger in the use of the software development tools.

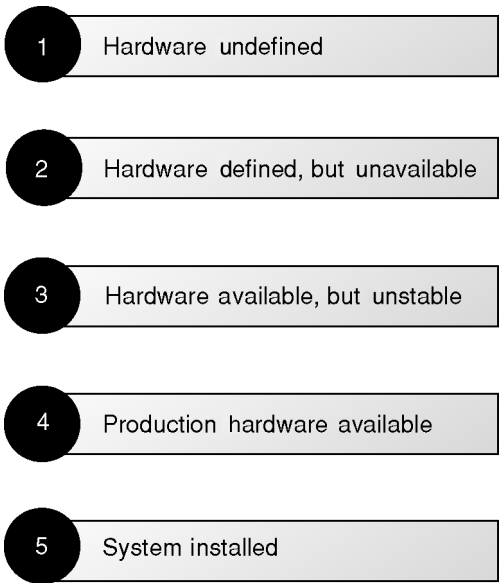



*Figure 2.5: Development phases*

At phase 2, since the hardware configuration is known, the engineer performs detailed software design and a large part of the implementation.

In phase 3, although hardware is available, a software engineer may often wish that it was not because the hardware will probably be unstable. However, the engineer can now begin the software/hardware integration.

In phase 4, the availability of stable hardware, maybe in multiple units, permits the engineer to complete final integration and testing.

Some development projects can be completed entirely within the factory, without requiring phase 5. Commonly, however, on-site installation requires final tuning of the software. At some later time, enhancements to the system may necessitate on-site work on the software.

Each development phase calls for a particular type of debugger, as described in the sections that follow.

### *Native Debugger*

At first sight, a native debugger (i.e., one running on the host computer, executing code in that environment) seems inappropriate for embedded systems development. However, there are two contexts in which such a tool may be useful.

During phase 1 of the project, with no clear idea of the target hardware configuration, a native debugger can provide a useful environment in which to develop ideas and formulate algorithms, particularly for sections of code that are not time critical. This idea can be extended further if a host-based prototyping environment is available. This permits a significant amount of development to proceed on parts of the application that interact with the hardware.

If a native debugger is available, one that has the same (or very similar) user interface to debuggers being used at later stages of the project, the native debugger can offer an ideal training ground, since even if the target hardware is available for training purposes, it may not be the safest place to "play around." The worst that can happen with a native debugger is to crash the computer. The consequences of some embedded systems going out of control may be more dire.

### *Debugger with Simulator*

The simulation of the target chip, instruction by instruction, on the host computer provides a very useful environment for software testing at almost any phase of the project. In particular, at phase 2, when the hardware is known but unavailable, a simulator will make rapid progress possible.

A simulator allows very detailed debugging to be performed. Although not running at anything like full speed, the simulator keeps track of execution time and permits accurate timings to be taken. This means the engineer can fine-tune critical code sections

early in the development cycle. Since the simulator can effectively add functionality to the microprocessor it is simulating, the execution of the code may be monitored in great detail without any intrusion at all. This facilitates 100% performance analysis and code coverage, which is not possible using other techniques.

Of course, a simulator limited to the simulation of just the core CPU would be of limited utility. The simulator must also address the interrupt and I/O systems.

### *Debugger with ICE Interface*

An in-circuit emulator (ICE) for the microprocessor is a very powerful tool for software/hardware integration, particularly when the hardware is exhibiting instability. An ICE enables the software to be run at full speed on the target, while permitting a real-time trace and the specification of complex breakpoint conditions.

Unfortunately, while they were once a ubiquitous tool in any embedded development lab, ICEs are no longer available for most high-end processors. The clock speed of processors made the devices more difficult and expensive to produce, and other technologies have become accepted alternatives.

If one is available, the usefulness of an ICE is influenced critically by the user interface, whose operation in high-level language terms is assumed. However, an interface that is compatible with other debuggers in use during the project is a real bonus. An important parameter in the selection of a debugger is support for industry-standard ICEs.

### *Debugger with Monitor*

Once stable and fully-functional hardware is available, the exceptional power of an ICE is less necessary. This is partly because ICEs can be overkill once the hardware is working reliably. Additionally, a cost-effective means of performing on-target debugging for large teams is increasingly required.

This situation led to the development of monitor debuggers where the target hardware is connected to the host computer by a communications link (serial line, Ethernet, etc.) and the target runs a small (<10 K) monitor program that provides a debug environment to the debugger itself, which runs on the host. The result is a low-cost, highly functional debugging solution that enables code to be run at full speed on the target with very little overhead. The ICE may be retained for use in particularly tricky situations.

For a monitor debugger to be viable, the monitor itself must be highly configurable. Standard boards (VME cards and evaluation boards) should be supported “out of the box.” Tools and services must be available to facilitate the rapid accommodation of custom hardware.

Although the use of a monitor debugger is most common during phase 4 of a project, it can also be used in phase 5. If the target monitor is included in the shipped software (after all, its memory overhead is likely to be very small), on-site debugging may be possible using just a laptop computer running the debugger.

### Debugger with Hardware Assist

As the speed and complexity of microprocessors increases, the likely cost (and lack of feasibility) of in-circuit emulators increases. As a result, semiconductor manufacturers are increasingly adding debug facilities to the silicon itself. This may vary from the provision of hardware breakpoints (address/data comparators), which should be supported by a monitor debugger, to a special "debug mode" that requires specific debugger support.

An early example of such a debug mode is background debug mode (BDM), which is featured in Freescale 683xx (CPU32) series devices. Most commonly, devices use a JTAG connection to provide on-chip debug (OCD). Assertion of OCD mode stops the processor and enables a debugger to read and write information to and from the machine registers and memory. To utilize OCD, an appropriate connector must be included on the target board, but this low-cost connector does not represent a significant overhead. Between the host computer and the target board, an OCD adapter is required. Like a monitor, a debugger with OCD (also termed "hardware assist") provides some ICE functionality at a much lower cost. Unlike a monitor, such a technique does not require an additional debug communications port(s) or code on the target.

### Debugger with RTOS

As embedded applications become more complex, the use of a real-time operating system (RTOS) is increasingly common. Debugging such a system has its own challenges, and they dictate specific requirements in a debugger. Two particular areas of functionality are required in an RTOS debugger:

- Code debugging must be "task aware." Setting a breakpoint on a line of code should result in a break only when the code is being executed by the task being debugged. Code shared between tasks is very common, so this requirement can easily arise. Similarly, data belonging to a specific task instance must be accessible to the engineer.
- Information about the multitasking environment (system data) is required: task status, queues, inter-task communications, and so on.

It is clearly desirable that both these requirements are addressed in the same debug tool.

If an in-house designed RTOS is used, particular debug challenges arise.

RTOS awareness may be implemented using all of the previously mentioned debug technologies. In particular, OCD and monitor debuggers are most commonly adapted. It is, however, quite possible to enhance simulators or even native debug environments to be RTOS aware.

## Debug Tools: Key Issues Recapped

In selecting debug tools for embedded systems software development, there are two key issues:

- Does the debugger permit the use of fully optimized code?
- Do the tools provide support for a wide selection of execution environments used in various phases of the development?

### Standards and Development Tool Integration

When selecting development tools, attention to standards is essential. For build and debug tools, it is worth investigating the tools that colleagues and associates are using. Industry standards are likely to enjoy long-term support and "grow" with the target chip. Apart from the development tools themselves, integration with standard version management systems is increasingly a requirement with larger project teams. Similarly, clear links to design techniques must be sought.

Beyond industry standards, attention should be paid to the adherence to "real" standards; that is, those set by international standards bodies. An obvious starting point is the programming language itself. Although the use of pure ANSI C/C++ is desirable, in reality a few specific language extensions are essential to make the language useful and efficient for embedded systems development. Such extensions are provided by suppliers of appropriate compilers (i.e., compilers specifically designed for working with embedded systems), and their use is, of course, very reasonable. A good example of an essential extension to the C language is the keyword `interrupt`, which enables a C function to be declared an interrupt service routine. Then the compiler can take care of the necessary context saving and restoring. However, some nonessential language extensions, provided by a few suppliers, should be avoided to aid code portability between compilers. Similarly, the use of a standard object module format (OMF) for relocatable and absolute binary files may remove the necessity of using build and debug tools from a single source.

In broad terms, choosing tools developed with open interfaces ensures interoperability with other products now and in the future.

### Implications of Selections

Although selection of the software development tools is important in itself, it is one of a number of such selections that must be made during the development of an embedded system. Other selections include the target microprocessor, the development host computer, and the RTOS. It is important to appreciate the interaction between these various selection processes, some of which may be less obvious than others.

#### *Target Chips*

Many reasons can be cited for the selection of a particular microprocessor:

It has the right range of features.

The price was right.

Low power consumption.

It is fast.

I have used it before.

A colleague is using it.

I liked the salesman.

These reasons are all valid, and a combination of them may be justification for selecting a device. However, another criterion should also be applied:

A good range of software development tools is available.

Purchasing something from a single, unique source rarely is an acceptable decision. Why should it be the case with software tools? If a microprocessor is supported by a very limited range of tools—perhaps from a single vendor—its use should be called into question.

### *Host Computers*

The choice of development platform is largely driven by the local culture. It is likely to be a PC (Windows or Linux) or a UNIX workstation. Software tools vendors offering support on an incredibly wide selection of hosts may be guilty of redefining the word "support." Often, on the less-popular platforms, the product versions on offer are extremely old and have not been maintained.

### *RTOS*

The choice of an RTOS (along with the decision to use one or not) is influenced by a number of factors. This topic is worthy of an article by itself; however, the availability of development tools is a significant factor, which I address here.

An RTOS with a suitably open architecture makes the most sense. It should accept the output generated by a range of build tools. Suitable debugging tools must also be available.

An option, which is considered under some circumstances, is the use of an in-house developed RTOS. This often represents the worst case in terms of tool availability and compatibility.

## Conclusions

The selection of development tools for embedded systems software is not an easy task, with many vendors offering partial or even complete selections of products. A good appreciation of the possibilities and a checklist of questions to pose to vendors are key prerequisites.

## 2.3 Eclipse—Bringing Embedded Tools Together

*In 2004 interest in Eclipse for embedded development applications snowballed. Late in the year Sarah Bigazzi wrote a piece for* NewBits *discussing the technology and its application, which I have adapted for this article. (CW)*

### Introduction

Development tools are widely known to be key to the success of microprocessors. Although powerful embedded tools have been developed over the last two decades, little progress has been made in integrating multivendor tools on multiple hosts. Without good integration, communication between tools is restricted, and the full potential of the tools is untapped.

Proprietary IDEs (integrated development environments) limit integration and prevent use of best-in-class or preferred tools. This inflexibility frustrates developers and curbs productivity. De facto proprietary standards partially address this problem but are restricted to a single host. Thus, embedded developers have long wished for a host-agnostic open IDE that they can enhance with their own or third-party tools.

The new Eclipse platform, an open host-independent, industry-standard base, makes this possible.

On the desktop, the Eclipse platform is already noted for its excellence and is used in numerous business applications. The benefits seen on the desktop—a common tool interface and integration platform—can be brought to the embedded world.

In this article I introduce Eclipse and describe how it can be adapted and enhanced to make an extensible embedded platform without compromising key Eclipse concepts—standard interface and plugability.

### Eclipse Platform Philosophy

During the Internet boom days, the availability of tools mushroomed for the various Internet business applications. Since these tools were built by diverse organizations, most of them had their own GUI paradigms and rarely worked well with each other. It became apparent that a standard IDE and framework were required. To address this need, IBM started the Eclipse project to build a well-designed tool integration platform so that independently built tools could be part of a single environment. The result was the Eclipse platform.

Originally, IBM released the Eclipse platform into Eclipse Open Source, and later, on February 2, 2004, the Eclipse Foundation reorganized into a not-for-profit corporation. "Eclipse became an independent body that would drive the platform's evolution to benefit the providers of software development offerings and end-users. All technology and source code provided to this fast-growing ecosystem will remain openly available and royalty-free."

Unlike other open source organizations, the Eclipse Foundation is driven by business needs; hence, it is also known as the "directed" open source organization.

A major goal for Eclipse is to provide a well-planned and secure platform for commercial tool vendors. In addition, the Eclipse Foundation constantly works to remove hurdles in licensing the platform for commercial use. Contributed code is thoroughly scrubbed before it is committed; to ensure ease of licensing, there are plans to replace the existing CPL (Common Public License), which is already much simpler than the GPL (General Public License), with a more relaxed EPL (Eclipse Public License).

### Platform

The Eclipse design focuses on a new paradigm—an open platform to integrate tools. In the old paradigm, individual tools are integrated, one at a time, either into an IDE or with another tool. This is a workable patch for a small set of proprietary tools but fails to scale in the larger multivendor context.

To address scalability, the Eclipse platform uses the innovative plug-in architecture. The platform, developed from the ground up, comprises well-defined GUI and framework mechanisms that provide a standard interface, facilitate integration, and are extensible. Tool developers, who no longer have to worry about GUI and framework issues, can concentrate on their tool-specific advancements—for example, multicore debug.

Extension points, extensions, and plug-ins form the underlying mechanisms of the plug-in architecture. Plug-ins are the smallest functional entities. Eclipse plug-ins from any source can be plugged into the platform for a single integrated environment. Except for the platform runtime, Eclipse itself is implemented as a set of plug-ins as shown in Figure 2.6.

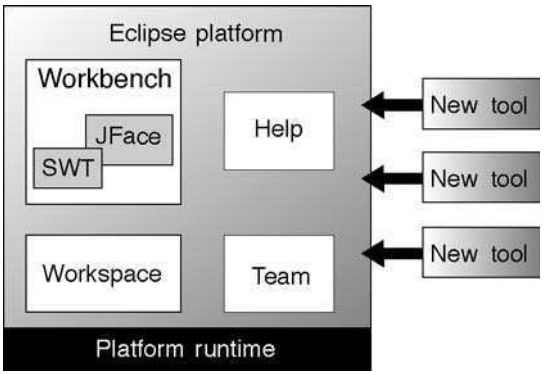



*Figure 2.6: Eclipse platform architecture*

The core of Eclipse is its user interface made up of the Workbench, JFace, and the Standard Widget Toolkit (SWT). The combination of these plug-ins is known as the Rich Client Platform (RCP).

- **SWT and JFace:** SWT and JFace take care of the windowing system in an OS independent way allowing portability across hosts.
- **Workbench:** The Workbench is the Eclipse UI. It is a collection of editors, views, perspectives, and dialogs provided as a common base for tools to use and extend.
- **Workspace:** Resources—projects, folders, and files—reside in the Eclipse Workspace where you can navigate them at will. The manipulation of resources, of course, provokes automatic incremental builds.

- **Team:** The Team plug-in takes care of source control. CVS (Concurrent Versions System) is the default, but other source control systems, including ClearCase, Source Integrity, and Visual SourceSafe, can be plugged in.
- **Help:** The Help plug-in does what the name implies. Integrated tools can extend Help for tool-specific needs.

### How Eclipse Gets Embedded

As many in the Eclipse community have said, "the Eclipse platform by itself is an IDE for everything and nothing in particular." CDT (C/C++ Development Tools) and JDT (Java Development Tools) are open source incarnations of the Eclipse platform for the desktop C++ and Java developer, respectively. They do not address the complexities of embedded development.

An Eclipse-based embedded IDE is a powerful, self-contained environment for building and debugging embedded systems, integrating program management and debug tools, and allowing users to drop in their favorite Eclipse plug-ins—for example, an editor or source control system.

It is important that such a development strictly adheres to Eclipse principles and is itself implemented as a set of plug-ins. This methodology allows the inheriting of today's key features in Eclipse, but also future ones as they become available. For example, platform runtime changes made in Eclipse 3.0 would be automatically reflected in the embedded IDE.

The key embedded development technologies that need to be made available as Eclipse plug-ins include:

- **Build tools:** To accommodate the great variability between embedded systems, compilers, assemblers, linkers, and so on, build tools tend to be significantly more complex than their native development counterparts. The return—in terms of improved usability—of their incorporation into an IDE is very significant.
- **Debug:** Software engineers spend more time debugging than anything else, and embedded programmers tend to use a variety of debug tools, which may accommodate different execution environments, RTOS awareness, or multicore debug capabilities.
- **Target connection:** An embedded development environment generally consists of a host computer with one or more target devices connected to it. These targets may be local, or they may be located remotely and reached via a network; they may be "real" targets (i.e., actual boards) or "virtual" (i.e., provided by some kind of simulation or emulation facility). Selecting and configuring the connections to multiple targets is another complex matter that may be readily simplified in this way.
- **Simulation:** Availability (or, rather, nonavailability) of hardware is an increasingly difficult challenge for embedded software developers, as the beginning

stages of software development are brought forward to an earlier point in the project cycle. A number of simulation tools may be employed—native execution, instruction-set simulation, and hardware/software coverification are all options. These also need to be brought into the IDE.

- **Profiling:** Ensuring that an embedded application functions correctly is the first priority, but, since resources are always scarce, profiling tools are employed and need to be contained within the IDE. These tools analyze how resources—time and memory primarily—are used by the application (and any associated RTOS).

## Conclusions

The need for an IDE for embedded software is apparent. The use of Eclipse as the basis for such an environment is clearly a very flexible approach, which is gaining ground across the embedded software development industry and will yield benefits for both suppliers and users alike.

## 2.4 A Development System That Crosses RTOS Boundaries

*At the end of 2004, Robert Day wrote a paper that looked at the use of Eclipse for embedded software development from a different perspective. That paper was the basis for this article. (CW)*

In the embedded software world, the choice of real-time operating system has typically dictated the choice of development tools. In the 1980s and 1990s, close relationships were established between RTOS vendors and tool providers.

At the end of the 1990s, industry consolidation resulted in major RTOS vendors and device providers owning their own tools technology. With the increase in software in every new embedded design, the reuse of software is becoming critical, making it very unattractive to keep switching embedded software tools.

This leaves us in an interesting tools conundrum. Which tools do you choose if you are using a given RTOS and device combination? And how can you protect your investment in tools (and code generated using those tools) if you switch your RTOS and/or devices? This situation is further exacerbated if more than one RTOS and/or more than one device are used in the same design—a common occurrence in the system on chip (SoC) era.

### Are Standards the Solution?

Standards are often an effective way of dealing with code reusability. The widespread use of ANSI C as a development language across embedded systems gives developers a chance for reuse. However, there are no real standards for tools and RTOSs. Therefore, each design has to use a specific compiler, which generates code for a specific device, to work with a specific RTOS. The RTOS and the tools have a proprietary API from compiler flags, through debug GUI, through IDE interface. This means that each project involves a substantial amount of relearning and retooling.

What is really needed is an environment that is standard across all embedded systems tools. This would allow the tools and RTOS vendors to plug and play with each other, and it would give users the benefit of a single tool to manage their projects and code. The environment would also provide standard interfaces and interoperability across the multiple tools and operating systems that are available today. This same environment could also support embedded projects that use a proprietary RTOS, or no RTOS at all.

One of the issues associated with creating such an environment touches on embedded systems politics. If either a device company or an RTOS company created this environment, it would become very difficult for their competitors to embrace it as their standard, because it would leave them somewhat at the mercy of their competitor. So having an environment developed and maintained by an agnostic third party is very appealing.

Looking to the desktop or enterprise world is an interesting exercise, because it inherits the benefits of a huge developer network, and the ability to use desktop productivity tools (such as standard editors and version management systems). The problem with

this approach is that the desktop world is very host-specific (e.g., Microsoft Visual Studio). In addition, if embedded features are required, the desktop tools providers are not readily open to develop those features because they would have a relatively small base of embedded customers.

### The Eclipse Solution

The good news for the embedded world is that a solution exists. While it may not be well known in the embedded world yet, Eclipse is set to revolutionize the embedded software developer's environment. It is as much a culture as it is a product, and it has been embraced by major embedded solution providers as their standard tools environment.

Eclipse is an open platform for tool integration built by an open community of tool providers. To quote the Eclipse web site: "The Eclipse Platform is an open IDE for anything, and for nothing in particular." It was developed by IBM and first released in 2001. In 2004, it was spun out of IBM into a nonprofit corporation called the Eclipse Foundation.

The technology is an open framework that is available in source code. The framework is written in Java and is highly portable across host environments. To date, it is available under Windows, Linux, Solaris, HP-UX, Mac OS, and IBM AIX.

### Eclipse Plug-Ins

A key factor that makes Eclipse useful is the notion of plug-ins. Each tools provider can build their tools according to a certain set of rules and APIs that allow them to plug-in to the Eclipse framework. In the embedded world, this enables embedded tools providers to build true embedded products that will simply plug into the IDE, and it allows them to focus on their core competencies without worrying about developing IDEs.

If the Eclipse plug-in rules are adhered to, the embedded tools will track the latest versions of the Eclipse framework, as well as plug into other Eclipse-based environments.

### Eclipse Licensing

Another key factor that makes Eclipse useful is the licensing model under which the Eclipse framework is provided. The Common Public License (CPL) provides royalty-free source code and world distribution rights and allows tools developers to offer the Eclipse framework and their plug-in products without putting their own intellectual property (IP) back into the community.

This model makes for a very viable business by allowing an open source framework to be a common vehicle, with a common look and feel. The open source framework can also be maintained by an RTOS- and device-agnostic organization and contain detailed embedded functionality provided by the embedded companies.

### Eclipse User Advantages

Many of the embedded RTOS companies are now providing an Eclipse-based solution. This means that the embedded user will have a common platform for the common development functions such as project management, editing, file navigation, and build management. The user will also have a common interface to compilation and debugging tools.

If the RTOS providers have implemented their plug-ins correctly, then this one environment can host multiple operating systems without having to change environments. Each RTOS vendor can provide tools that will have a common Eclipse look and feel, but with specific features that help build and debug applications using that RTOS.

An example of where this is particularly beneficial is in companies with a large product portfolio that serve the same market but necessitate different user requirements—such as cell phone companies.

#### *Cell Phone Example*

High-end cell phones may need a PDA-like operating system such as PalmOS, Windows CE, or SymbianOS, whereas middle- and low-end phones may use more of a true embedded RTOS. Much of the software IP will be the same, so having it under a single project manager is very beneficial. The devices are also likely to be the same, so the compilation system can be consistent. Only the RTOS environment and high-end applications will change, which Eclipse can facilitate.

When considering a multicore architecture, the cell phone example also applies, because most cell phones use at least one standard processor core and at least one DSP. Eclipse can host the different compilers, debuggers, and RTOS choices for each processor in one environment.

### Perspectives

Eclipse, as do engineers, operates using the notion of perspectives. When building, engineers use a build perspective, and when debugging, they use a debug perspective, and so forth. A nice feature of Eclipse is that plug-ins can cross perspectives. For example, when debugging, the editor and project manager can be made visible and used to make changes to the code if bugs are found and to navigate around the project to fully understand the context of the debugged code. An example of an Eclipse debug perspective is shown in Figure 2.7.

### Nonembedded Plug-Ins

Another advantage of Eclipse is the ability to plug in development productivity tools that are not specific to embedded systems. Currently available Eclipse plug-ins provide the following functionality:

- Modeling
- Bug tracking

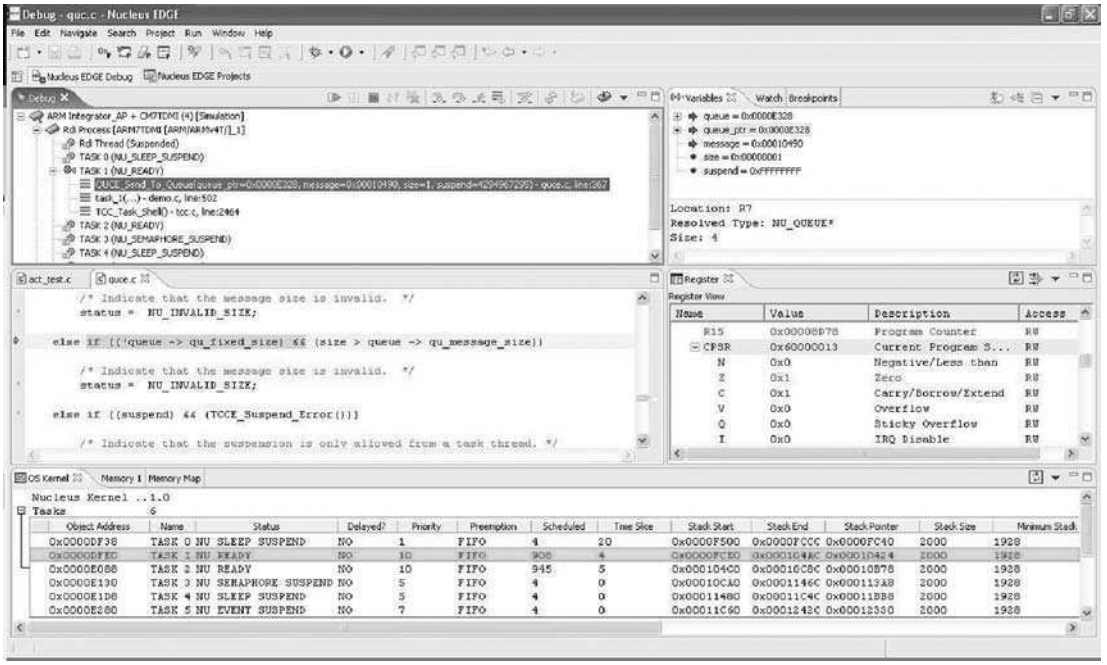



*Figure 2.7: Eclipse debug perspective*

- Code generation
- Graphics/drawing
- Source analysis/testing
- Project/team management
- Source control (such as CVS, ClearCase, and Visual SourceSafe)

Eclipse provides many advantages for embedded developers; it is now in the hands of the embedded solutions providers to make it a reality.

## 2.5 Embedded Software and UML

*Embedded software is a fashion-conscious business. From time to time, different products, technologies, and methodologies are "flavor of the month." That's not to say that these fashionable ideas don't last—it's just that they generate a surge of interest and then settle into their place. As the twenty-first century began, the UML showed that surge, and everyone is interested in what it can do. It appears to address real problems that many developers are experiencing on real projects. Clearly a practical approach to its use for embedded software is needed—embedded developers are very definitely pragmatic people. In a series of articles in* NewBits *in 2004 and 2005, Stephen Mellor (with some help from Alasdar Mullarney) described an approach that makes practical sense. This article is based upon that series. (CW)*

### Why Model in UML?

Yes—why? For all the usual reasons: to reduce costs, time to market, and unnecessary redevelopment and to increase productivity, quality, and maintainability. How can UML models do all that?

That depends on what a model is, and how it relates to the systems development process. There are at least three meanings of "model," and each meaning has different uses and implications. Let's take a look at each meaning.

One meaning for the word "model" is a "sketch." For example, we might sketch out a hardware configuration on the back of a beer mat, showing a few boxes for processors and lines for communication or adding a few numbers to indicate bandwidth or expected usage. The sketch is not precise or complete, nor is it intended to be. Often, a sketch of this nature is "talked to" by pointing at various boxes to explain what is happening there and how it relates to other elements. The purpose is to communicate a rough idea, or to try one out just to see if it will work. The sketch is neither maintained nor delivered.

A second meaning for "model" is "blueprint"—a classical example is the set of plans for a house. The blueprint lays out what must be done, describing properties needed to build the real thing, as determined by an architect. Because blueprints are intended to be plans for construction, they often map closely to the artifact that is to be built, so for each important element in construction, there is a "symbol." Because software is a complex beast, the set of symbols—the vocabulary—can become quite large, and without standards, chaos can ensue.

Enter the Unified Modeling Language (UML), which is a language that can be used for building software blueprints. (It has other uses too, as we shall see.) The UML is the result of an effort to reduce needless differences between different systems development methods and establish a common vocabulary for software modeling.

Why would you want to use UML? For all the reasons we outlined previously. Thinking about what you intend to build carefully—to the point of defining it exactly so that someone else can build it—will reduce costs and decrease defects, similar to the efficiency of writing a detailed, reviewed shopping list that avoids all the effort involved in returning a wrong item and getting the right one.

But as anyone who has built a house knows, the blueprint is rarely followed to the letter. Instead, as the builder (in contrast to the architect) constructs the house, the facts on the ground cause some modifications to be made.

The same argument can be applied to models: as we write code, we discover that our design wasn't as clever as we thought. This critique has led to the deprecation of models as "paper mills" that deliver pictures but not working systems. Instead, it has been argued, we should just hack—sorry, write—code because it executes.

Execution is important because it closes the verification gap between a concept on paper and a reality that either works or not. Code either runs right or it doesn't. You can't be certain of that one way or the other with a blueprint, even with the best review team in the world.

The third meaning for "model" then, is an "executable." When we build an executable model, we have described the behavior of our system just as surely as if we had written a program in C. Indeed, when you have a software model that can be compiled and executed, there's no need to distinguish between the model and the "real thing." It is the software.

So does this mean we should "program in UML"? And, if so, why should that reduce costs, time to market, and unnecessary redevelopment, as well as increase productivity, quality, and maintainability?

The answer to the first question is "Yes, but at a higher level of abstraction." For example, when you declare an association between two classes, you do not say whether that will be implemented by a pointer, a reference, or a list (just as when you program in C, you don't think about allocating registers). So while you are "programming," when you build an executable UML model, you don't have to think about a lot of things you normally worry about when programming in a language at a lower level of abstraction.

This approach reduces costs (the first of our reasons for modeling) because the cost of writing a line of code is the same irrespective of language. Studies as far back as 30 years ago showed that, on average, a developer produces 8 to 12 lines of assembly code, or C, or FORTRAN, or whatever per day. These numbers are "fully loaded," meaning that we're taking into account the time we spend in meetings, unjamming the printer, dealing with performance reviews, fighting the configuration management system, running tests, and all that other stuff. Although some programmers are much more productive, their productivity is also the same irrespective of language.

When we program in an executable UML, we write at a higher level of abstraction, thus reducing costs and increasing productivity. One user of executable UML generates 7 to 10 lines of C++ for each line of logic written in UML; the amount of code would be greater if this user's projects were written in C. Because the number of lines of code per day is the same, this translates directly into a decrease in time to market and an increase in productivity.

For these reasons, we developed higher-level programming languages as sketched in Figure 2.8.

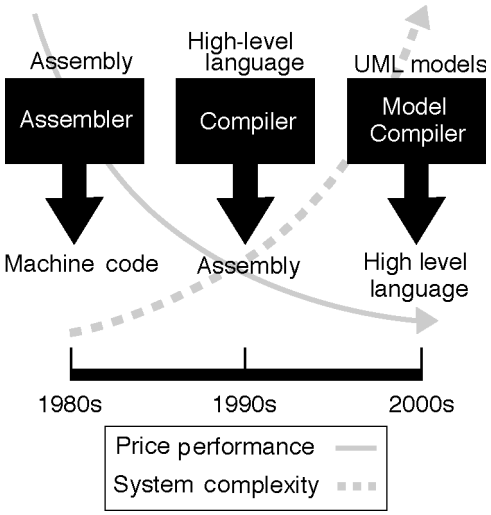



*Figure 2.8: The evolution of software development*

Using an executable UML also increases quality. Not only is the number of defects reduced, but the errors are found earlier, providing time to react. It is better to know you have a problem when you have six months to go on the project than six weeks! Figure 2.9 shows the effect of applying this methodology.

Early error identification is achieved by building test cases and running them against the executable model. Because the model is executable, we can provide real values and get real results immediately, using a model executer that interprets the models. You and other experts can see immediately whether the model is doing the right thing. If it is not, you make the change in the models, then and there, and run the tests again.

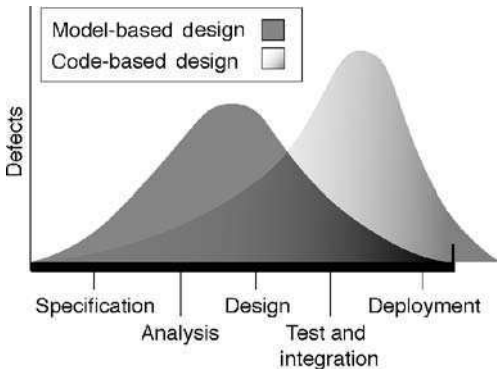



*Figure 2.9: Model- versus code-based design*

We must emphasize that model testing occurs early in the life cycle, thus removing downstream defects. In turn, this reduces the effort involved in implementing the wrong thing, just as with that shopping list. The combination of removing defects early and avoiding wasted effort implementing the wrong thing reduces costs and time to market, while increasing productivity and quality.

Models are also more maintainable than code because it is easier to manipulate concepts at a high level of abstraction than a lower one. The careful reader will have noted that we have discussed all of our reasons to model except reduction of unnecessary development, or—putting it in the positive—maximizing reuse. While it is certainly easier to reuse models than code (that higher level of abstraction argument again), the main reason you can reuse models is the same reason you are more likely to reuse a C program than one written in M68000 assembly code—namely, you can port the C program across multiple hardware platforms.

The same concept applies to executable models. When we built an association, we did not specify whether it was implemented as a pointer, a reference, or a list. This allows us

to decide later, once we better understand the speed and performance constraints of our system. In other words, executable models confer independence from the software platform, just as writing in C made us independent of the hardware platform. We can then redeploy the executable model onto different software platforms and implementation environments. This is actually something of an understatement. Models can be translated into just about any form, so long as their application behavior, as defined in the executable model, is preserved.

This brings us to a concern. When we moved from assembly code to C, we lost control of, for example, register allocation, which could lead to a reduction in the performance of the system—a killer concern in an embedded system. The keyword here is "control." If the compiler does a good enough job we don't care, but if the compiler doesn't know enough about our environment to make sound decisions, and we have no control over those decisions, we're in trouble.

For this reason, models need not only to be executable, but also to be translatable onto any software platform, and you, the developer, have to have control of how that translation process takes place, reducing performance concerns to zero. After all, if you can write the code, you can also describe how to go from a concept, as expressed in an executable model, to that code. A translatable UML also offers complete control of how that code is produced.

It is for this reason that we support executable and translatable UML, or xtUML, for short. xtUML models are both executable and translatable to any target software platform in an open manner. We do this by using a trick that differentiates blueprint models from executable models: separation of application from architecture.

## Separating Application from Architecture

The separation of the application from the architecture differentiates blueprint-type models from executable ones. To understand that, we first need to understand how "blueprint" model-driven developers do their work.

### *Blueprint Development*

After some initial requirements work, which can be supported by models such as use cases, the blueprint developer builds an analysis model, in UML, that captures the problem domain under study. This model will use various elements of the UML, but there is no universal agreement as to what those elements should be. As a simple example, the UML allows for attributes to be tagged with a visibility (`public`, `private`, etc.). Should an analysis model include this information? That depends upon taste—some do, some don't. Everyone is agreed that an analysis model should not contain design details, but there is little agreement on what that means exactly.

The next stage is to build a design model that does incorporate all that design detail. The design model is a blueprint that captures the software structure of the intended implementation. The work of transforming the analysis model to a design model

exercises embedded systems design expertise. For example, we know, as embedded system designers, that a good way to store fixed-size data elements in a memory-limited environment is to pre-allocate memory—or whatever your expertise tells you to do. This expertise is applied to the analysis model to produce the design blueprint.

The next step is to code it up from the blueprint. Putting aside possible errors in the design, this means filling in code bodies. There are two ways to do that. One is to add the code directly to the model, and have a tool generate code according to the software structure. Another way is to code up the software structure suggested by the blueprint, adding in coding details. The process is sketched in Figure 2.10.

### *What's Wrong with That?*

Nothing, if you like doing all that work over and over every time the technology—and therefore the software structure—changes. And if you like reinventing and reapplying the same programming constructs when you add new system functionality.

This approach to software development is rather like using C-like pseudo-code to outline your design, then hand-coding the assembler. Each time you add new application functionality, you have to decide over again how to pass parameters to a function, how to allocate registers to compute an expression, and so on. Each time you port to a new hardware platform, you have to work out what the assembly code meant (you wouldn't trust the pseudo-code, would you?) and rewrite it for a new processor.

At root, you have failed to leverage and capture the embedded systems design expertise represented by going from analysis to design to code. Or, to use the pseudo-code analogy, you have failed to leverage and capture the expertise involved in assembly coding.

### *Model Compilers*

The solution, of course, is to build a compiler from a more formalized pseudo-code (which we may call C) for each of the various processors. Certain parts of the compilers are common, such as building an abstract syntax tree. Others are specialized to the target processor, though they may share common techniques for register allocation, expression ordering, or peephole optimization. The expertise is captured in an artifact (a compiler) which can be reused as required.

The same concept applies to xtUML model compilers. We can build model compilers for each software platform. (Note the adjective: each *software* platform.) A software platform is simply that set of technology that defines the software structure, such as choice of data structure and access to it; concurrency, threads, and tasking; and processor structure and allocation. All these details are filled in by the model compiler, just as a programming language compiler fills in all the details of register allocation, parameter passing, and so on, as determined by the hardware platform.

This approach captures the expertise involved in making embedded software design decisions and allows you to leverage it across a project and across many projects. Model compilers, like programming language compilers, can be bought.

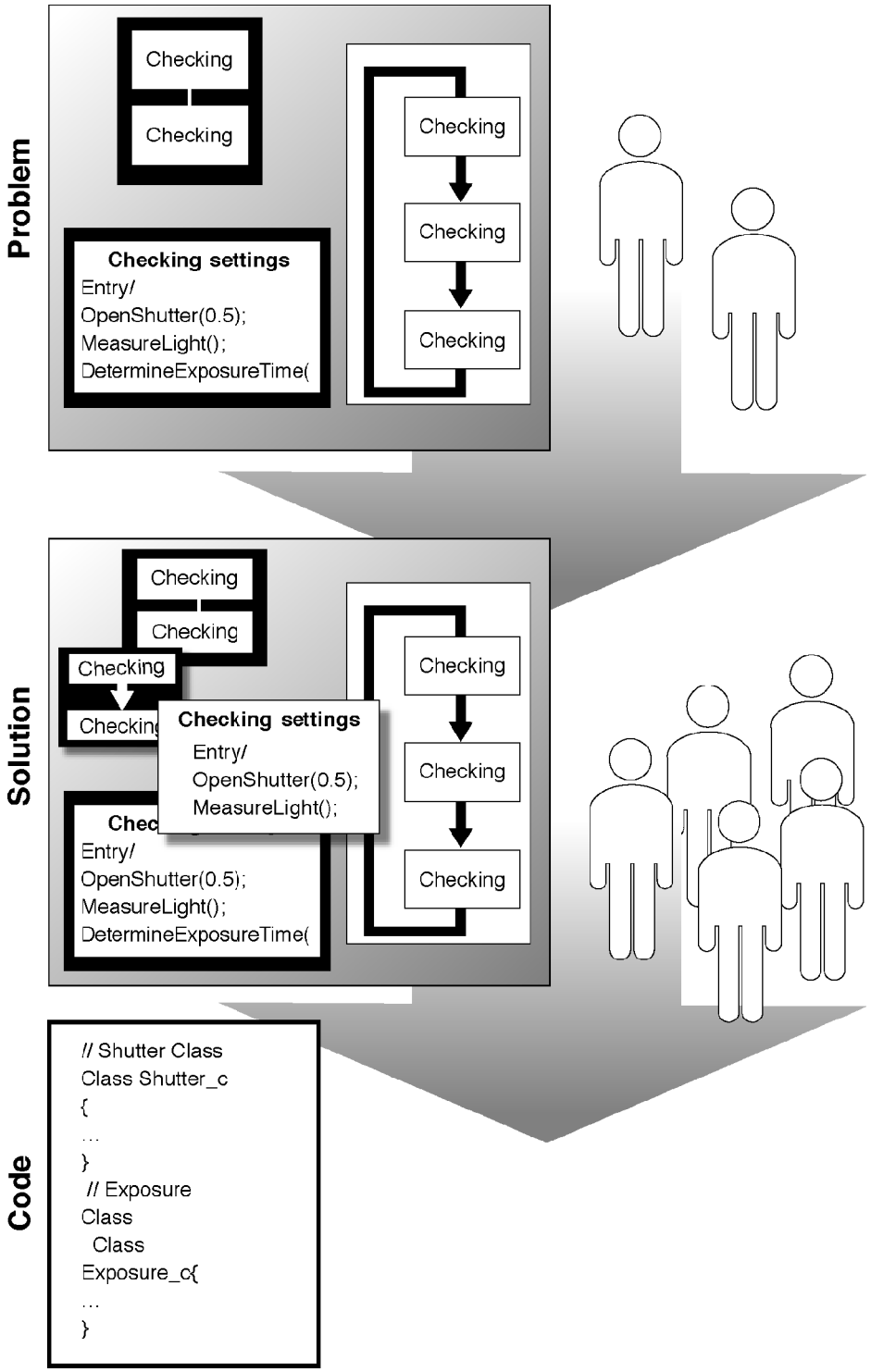



*Figure 2.10: The blueprint development process*

### *Sets, States, and Functions*

Figure 2.11 illustrates the separation between application and architecture. The element to focus on is the dotted line that separates the two.

When we build an xtUML model, it is represented in a simple form for translation, as sets of data that are to be manipulated, states the elements of the problem go through, and some functions that execute to access data, synchronize the behavior of the elements, and carry out computation. The UML is just an accessible graphical front end for those simple elements. When you build a "class" in xtUML, such as `CookingStep` in a microwave oven, it represents a set of possible cooking steps you might execute, each with a cooking time and power level. Similarly, when you describe the life cycle of

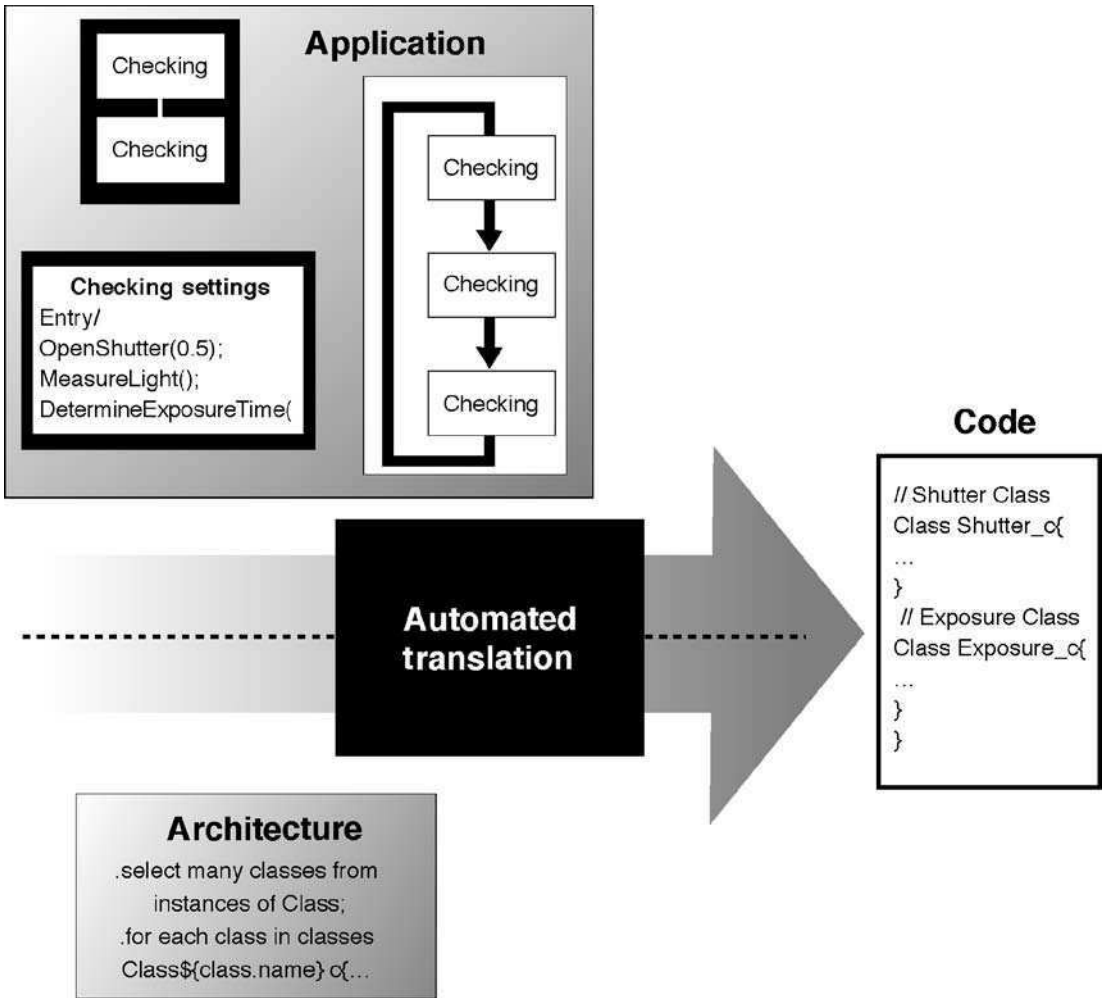



*Figure 2.11: Separation of application and architecture*

a cooking step using a state chart diagram, it follows a sequence of states as synchronized by other state machines (when you open the microwave door, it had better stop cooking!), external signals (such as a stop button), and timers. And in each state, we execute some functions.

Naturally, it's a bit more complicated than that, but the point is that any xtUML model can be represented in terms of these primitive concepts. And once that's done, we can manipulate those primitive concepts completely independently of the application details.

### *Rules*

The ability to perform this independent manipulation allows us to write rules. One rule might take a "class" represented as a set `CookingStep(cookingTime, powerLevel)` and produce a C++ `class` declaration. Crucially, the rule could just as easily produce a `struct` for a C program, or even a `COMMON` block in FORTRAN. Similarly, we may define rules that turn states into arrays, lists, `switch` statements, or follow the state pattern from the Design Patterns community. (This is why I put "class" in quotation marks. A "class" in an executable model represents the set of data that can be transformed into anything that captures that data; in a blueprint-type model, a class is an instruction to construct a class in the software.)

These rules let us separate the application from the architecture. The xtUML model captures the problem domain graphically and represents it in terms of sets, states, and functions. The rules read the application as stored in terms of sets, states, and functions, and turn that into code. This leads to the process shown in Figure 2.11.

The value here is that application models can be reused by applying different sets of rules (a different model compiler) to target a new software platform. Similarly, the model compiler can be reused in any project that requires the same architecture. The applications and the model compilers can each evolve separately, reducing costs, and increasing productivity and reuse.

### *Open Translation*

There is one critical difference between today's programming language compilers and model compilers. With a programming language compiler, you have limited control over the output. Sure, you can apply a few flags and switches, but if you truly dislike the generated code for any reason, you're out of luck unless you persuade the vendor to make the changes to the compiler you require. With a model compiler, the translation rules are completely open. If you can see a better way to generate code because of the particular pattern of access to data, say, you can change the rule to generate exactly what you want. This completely removes any concerns about optimization. It is totally under your control.

I should emphasize that you rarely need to change the model compiler, still less write one of your own. But the knowledge that you can change it should increase your confidence in the technology. Another analogy to programming languages: when they were new, people were concerned about the quality of the output and having some control over it. Over time, of course, those concerns have diminished, even in the embedded space.

### xtUML Code Generation

We will now take a look at the code that will be produced from xtUML models. Obviously, what we want is executable code. For an example, we will look at the safety-related logic of a simple microwave oven. The oven components are the door, which must be closed while cooking, and the actual cooking element. There will be some code to manage the cooking times and power levels.

Take a look at some representative code for such a microwave oven:

```
struct Oven_s
{
      ArbitraryID_t OvenID;

      /* Association storage */
      Door_s *Door_R1;
      Cooking_Step_s *Cooking_Step_R2;
      Cooking_Step_s *Cooking_Step_R3;
      Magnetron_s *Magnetron_R4;

      /* State machine current state */
      StateNumber_t current_state;
};
```

The C `struct` captures information about the oven, which has an arbitrary ID (an identifier to distinguish a particular instance) and some pointers that reference its components. The oven `struct` also has a `current_state` that captures the—well—current state of the oven.

The "Cooking step" in Figure 2.11 allows the microwave oven to be programmed to cook in steps, each at a different power level for a certain time. Each step describes cooking parameters for the oven. Typical uses are to program a cooking step to defrost by pushing one button (Time 1, say) with low power and a long time, followed by pushing another button (Time 2, for a second unimaginative name, say) to cook ready to eat at high power for a shorter time. There are twin steps because there are two buttons.

Here is the code for `Cooking_Step_s`:

```
struct Cooking_Step_s
{
      i_t stepNumber;
      i_t cookingTime;
      i_t powerLevel;
      Timer_s *executionTimer;

      /* Association storage */
      Oven_s *Oven_R2;
      Oven_s *Oven_R3;

      /* State machine current state */
      StateNumber_t current_state;
};
```

This structure includes a step number (used also as an identifier), cooking time, and power level. In the scenario described previously, there could be two instances of this `struct`, say:

| Step Number | Cooking Time | Power Level |
|---|---|---|
| 1 | 10 mins | 20% |
| 2 | 3 mins | 100% |

An execution timer is also used to refer to one of several potential timers. This reference is required so the timer can be interrogated, reset, or deleted. In addition, association storage refers back to the oven. There are two associations because we can program two cooking steps.

Again, the cooking step has a current state attribute to capture whether the cooking step is ready (i.e., has been programmed), executing, or complete.

We need to understand the conceptual entities in the problem and how they are described by data. Figure 2.12 shows a so-called "class diagram," which does just that.

This diagram declares four conceptual entities (the oven and the cooking step we have already discussed, plus a door interlock and a magnetron tube), with several associations identified with "R numbers." The numbers are simply a way to identify each association uniquely; the "R" comes from the real-world relationship captured by the associations. The associations also have names that capture the real-world relationship. `R2`, for example, is read: `Oven` executes first `CookingStep` and `CookingStep` describes first cooking parameters for `Oven`.

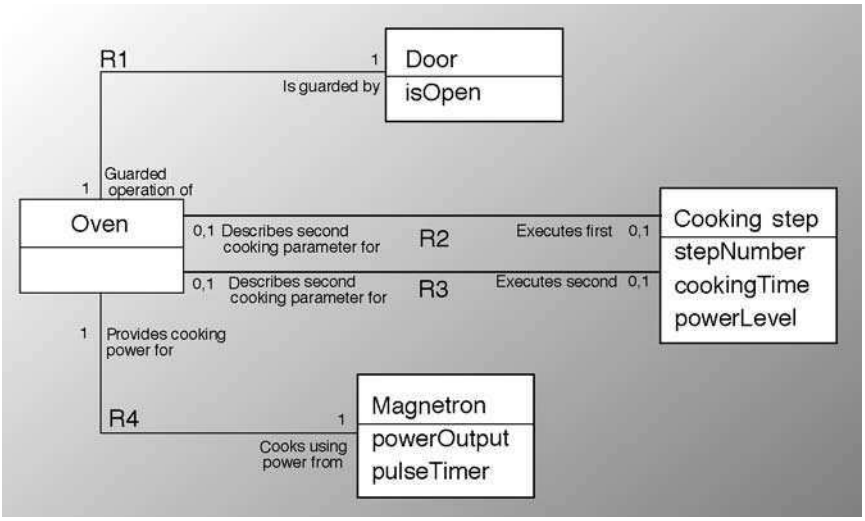



*Figure 2.12: Microwave oven class diagram*

The associations also have a multiplicity that indicates how many instances participate in each association. For `R2`, each oven may have zero or one (0, 1) first cooking steps. And each cooking step may or may not be the first cooking step for this oven (hence the 0, 1 again). The association (`R4`) between the oven and the magnetron is "1" in both directions because one `Oven` *houses* one `Magnetron`, and one `Magnetron` *is housed in* one `Oven`. In general, an association can have many instances, as when a dog owner owns (one or more) dogs, which would be written "1..*". A person, on the other hand, owns "0..*" dogs, because you have to own at least one dog to be considered a dog owner, but a person is free to choose not to own any.

Let's compare now the "class diagram" of Figure 2.12 with the declaration of `Oven_s`. We can see that the name of the `struct` is the same as the name of the box but with a suffix `_s`, and the `OvenID` has a type `ArbitraryID` with a suffix `_t`; both these coding conventions remind us of the purpose of the name symbols.

The second section of the code, marked "Association storage," has a pointer of type `Door_s`, named `Door_R1`, which implements the `R1` association. Remember, the `_s` indicates the pointer is to a `struct` capturing information about the door. Similarly, the other pointers implement the other associations, `R2`, `R3`, and `R4`.

The `current_state` attribute is part of an underlying state machine mechanism; it's a little special and deserves its own type—`state_number_t`. It happens to be an integer, but because we know something about its likely values, we may choose to implement it as an `unsigned char`, for instance.

The primary observation to make here is the close correlation between the oven diagram and the corresponding declaration. Note that the declaration of `Cooking_Step_s` and the Cooking Step "class" have the same close correspondence too.

So, why do we put quotation marks around "class"? We do so because the class `oven` isn't a class at all. It's a C `struct`! This goes back to the separation between application and architecture we discussed earlier. The application model describes the fundamentals of the solution, while the architecture defines the mapping to the implementation—just as we showed here in this extended example.

Moreover, the C we showed illustrates the point we made right at the beginning. The models we build are executable and translated into the implementation. This is in contrast to "blueprint-type" models that are intended to direct the implementation. There, a "class" means that we should build a `class` in the code. Here a "class" simply declares important data and houses behavior as defined by a state machine, the trace of which is the current-state attribute.

### Conclusions

Modeling and the use of the UML mean different things to different people; the terminology can be confusing and is often misused or abused. There are various possible goals that come out of the use of modeling, but an approach that reduces rework and leverages the diverse expertise of the embedded development team must be a clear winner.

#### A Note on Terminology

A **software platform** is analogous to a hardware platform. It is the set of technologies that we rely on to make our software work: linked lists, middleware, operating systems, IPC mechanisms, and so on. These technologies don't know about an application, although the choice of software platform will have an impact on system performance.

An **architecture** (more precisely, an application-independent software architecture) is an abstract representation of how to map an application to a specific target platform. (It includes the software platform by reference.)

A **model compiler** is the physical realization of the architecture; it is a program like a programming language compiler.

These terms have analogies in the programming language compiler world: a compiler embodies choices about the architecture based on, and relying on, the facilities provided by the hardware platform.

## 2.6 Model-Based Systems Development with xtUML

*Stephen Mellor and John Wolfe wrote a white paper, considering the longer term possibilities of the application of xtUML not just to embedded software, but to the design of the whole system. That paper was the basis for this article. (CW)*

Developers of embedded systems have always faced a number of challenges. From determining the hardware/software partition to meeting performance and cost objectives, the job of building these systems has never been easy. With ever-increasing demand for more functionality packed into smaller spaces consuming less power, building embedded systems definitely is becoming more complex every day. Add to these demands the need to shrink development cycles and reduce the overall cost of the system, and you have the state of the embedded systems industry today.

This brief paper provides a glimpse at a new way to overcome many of the challenges confronting embedded systems developers today.

### Why Is Building an Embedded System so Difficult?

Perhaps because it involves the coordination of at least two very creative and rather complex disciplines: software engineering and logic design. Maybe because it often involves creating something new—something nobody has ever built before. Or it may simply be the multitude of choices to be made: allocation of function among logic and software, which processor to use, how to arrange the bus, which programming language to employ, and whether or not to use an operating system. Most of these choices interact with one another in interesting ways, making the prospect of building a new product more than a little daunting.

#### *Every Project Has at Least Two Fundamental Challenges*

By their very nature, embedded systems meld software and hardware together to form a coherent solution. The user of the resulting product never fully appreciates the effort that goes into getting the partition between the two right. With current development practices, getting it wrong is very costly at best and disastrous at worst.

Verifying the hardware/software partition requires the ability to test the system, and producing a version of the system that can represent the real thing accurately enough to verify the partition requires a huge intellectual investment. First, a hardware/software interface specification must be painstakingly written (or scratched out on a bar napkin, as the case may be). Then, the logic designers and the software engineers must construct, using implementation languages like VHDL and C, behavioral models of the function to be mapped to hardware and software. In most cases, the software engineers are not building models at all, but instead are writing the application code for the system.

Working at a level of abstraction somewhere between that of C and assembly language, the logic designers toil away to produce behavioral models of the hardware, often writing small test drivers and diagnostics to test their models along the way.

### Integration: Where the Fun Begins

After months of effort, the two teams are ready to test their prototype system. Most folks do this in one of two ways: simulation or prototype hardware. With recent advances in FPGA technology, it has become significantly easier to test a logic design on an actual chip rather than resorting to simulation on the development workstation. Regardless of the test bed, the integration effort is always interesting.

Recall the development process started with two separate teams, each with different skills, heading off in parallel. The only thing connecting them is that hardware/software interface specification, written in natural language. You know where this is heading.

Two teams with disparate disciplines working against an ambiguous document to produce a coherent system. Sounds like a line from a cheap novel.

Invariably, the two components do not mesh properly. The reasons are myriad: the logic designers didn't really mean what they said about that register in the document; the software engineers didn't read all of the document, especially that part about waiting a microsecond between whacking those two particular bits; and of course, the most common failure mode of all, logic interface changes that came about during the construction of the behavioral models that (horrors!) didn't make into the interface specification.

### So What's a Few Interface Problems Among Friends?

Nothing really. Just time. And money. And market share. We've been doing it this way for years. It's nothing a few days (well, weeks) in the lab won't solve. Besides, shooting these bugs is fun, and everyone is always so happy when it finally works. It's a great bonding experience.

Eventually, the teams manage to get the prototype running, at least well enough that they can begin measuring the performance of the system. Keep in mind here that "performance" has a number of connotations in this context: along with the obvious execution time and latency issues, memory usage, gate count, power consumption, and its evil twin, heat dissipation, top the list of performance concerns in many of today's handheld devices.

### The Suboptimal Partition

There's nothing like a performance bottleneck to throw a bucket of cold water on the bonding rituals of the integration heroes. Unlike interface problems, you don't fix hardware/software partition problems with a few long nights in the lab. No, this is when the engineers head back to their desks to ponder why they didn't pursue that career as a long-haul truck driver.

Partition changes are expensive, and they are difficult to do correctly. Since the system is represented only in terms of implementation languages, knowledge of the partition is distributed throughout the software and the logic design. While it's simple enough to

say, "Let's move this function into logic," it's quite another matter to make it happen. Remember, we have months invested in the construction of the prototype system. Making fundamental architectural changes cannot be done in a matter of days, at least not with the traditional approach previously described.

Getting the hardware and software to mesh properly and making the right partition between them are two significant challenges faced by all embedded systems developers. Certainly, many others exist, but we promised a *short* paper.

### *The State of the Practice*

So, it takes months of effort to produce a prototype that can be executed, but we need to execute it before we will know whether or not the logic designers and the software engineers agree on the hardware/software interface. We need to run the prototype system before we can measure its performance, but if the performance is unacceptable, we'll spend weeks changing the hardware/software partition. That's the state of the practice.

So, what if we had a way to eliminate completely the hardware/software interface problems that are discovered during the initial integration? What if we also had a way to change the partition between the hardware and software in a matter of hours?

## A Better Solution

We have just such a solution. First, we build abstract models of the system using an executable and translatable UML (xtUML). These models have sufficient detail that they are executable, so we can test the behavior of the system early and continuously before investing in the construction of an actual implementation in C and VHDL. The models are also translatable; that is, they are completely independent of design and implementation concerns. Testing at this level is therefore concerned only with ensuring that the models accurately represent the application to be constructed.

When the models are complete, we can specify an initial partition between hardware and software. A model compiler then translates the models into logic designs (VHDL, Verilog, SystemC, etc.) and software (C, C++, Java, assembler, etc.) according to the specified partitioning.

### *Interface Problems?*

The interface between the hardware and software is defined by the model compiler. Because the implementation is generated, there can be no interface mismatches. Because we no longer have two separate teams of people working from a natural language interface specification, the generated implementation is guaranteed to have exactly zero interface problems. (This does have the unfortunate side effect of reducing the number of opportunities for logic designers and software engineers to spend long nights together in the integration lab working around interface problems.)

### What About the Partition?

Because the xtUML models accurately and precisely represent the application, and the implementation is generated, with absolute fidelity, from these models, the partition can easily be changed, and a new implementation can then be generated. This replaces weeks of tedious manual changes to an implementation with a few hours of automatic generation.

With the ability to change the partition and regenerate the implementation, the developers can explore much more of the design space, measuring the performance of various allocation arrangements that would otherwise be prohibitively expensive to produce.

## Experience to Date

An experimental model compiler has been constructed that generates C++ and VHDL. The generated system was then able to execute within a logic simulation environment. There is still a way to go, but the concept is sound.

## The Future

Our vision is to provide a complete system-level development environment that allows embedded developers to construct abstract models of their systems and then automatically translate those models into an optimized implementation that includes software and custom logic components. Of course, without the interface problems and the partition issues, we'll need to find another excuse to spend long nights and weekends in the lab.

# 3

# *Programming*

*In this chapter, the articles focus on specific programming problems encountered in embedded systems. I guess the "PowerPC Assembler" article is the odd one out, but it had no other logical home. Some appreciation of assembly language programming is essential for an embedded software developer.*

## 3.1 Programming for Exotic Memories

*I wrote the original version of this article for* NewBits *in 1991, when I was working for Microtec Research in the United Kingdom, and it required little revision for inclusion in this book. The issues discussed in this article continue to be of concern. It is interesting that, even back when I first wrote the article, solving a problem by hiding it in a C++ class seemed like a good idea. I was even referring to multichip embedded systems, which were far from common. This article may be usefully read in conjunction with another one I wrote, "Self-Testing in Embedded Systems," later in this chapter. (CW)*

### Exotic Memories

A palm-fringed beach with waves washing tendrils of white foam from the deep blue ocean onto the even whiter sand. The sun beating down on your head and an ice cold bottle of Mexican beer in your hand. The hint of suntan oil in the air and the appetizing aroma of lunch cooking on the barbecue in the nearby bar.

A memory of last summer's vacation? No, mine wasn't like that either.

Anyway, the kind of exotic memory I want to talk about is the much more prosaic kind: embedded systems memory storage, other than regular RAM and ROM. Nonvolatile RAM and shared memory both present specific challenges to the embedded systems programmer.

### Nonvolatile RAM (NVRAM)

In the good old days, real computers (the ones with rows of lamps and switches on the front) had magnetic core memory. We still use the terms "core memory" or "core dump" today, even though the technology has passed into the history books. Core memory had two significant characteristics that differ from those of modern semiconductor memory. The first was size. Core memory was generally packaged in slabs, several inches square, which typically held 4 K words. (I heard it suggested that a 4 by 4 matrix of slabs i.e., 64 K—could be termed a "patio," but I'm not sure this terminology ever escaped the lab in which I worked at the time.) The second feature was nonvolatility; the data was still there even after the power had been cut. This tended to be exploited during debugging. At the end of the day, the engineer would shut down the computer and go home. The following day, he would switch on the computer and carry on his work from where he left off. (This assumes that the core had not experienced any significant

mechanical shock—such as being dropped—because it tended to corrupt the memory, which was a source of concern for military and aerospace applications.)

Nowadays, most RAM is volatile and engineered to be nonvolatile only for specific applications. Without assistance, the data is lost on power down, and the RAM is filled with garbage on power-up. Typically you could get nonvolatility by providing a battery backup power supply to low-power RAM chips. In an embedded system, nonvolatile RAM is typically used to hold setup or configuration data, which must be retained across power shutdowns.

Why should these requirements present programming problems? Can't we treat nonvolatile RAM just like regular RAM? The answer to the second question is generally yes; while you can treat it like normal RAM, a little care is needed in two specific respects.

First, it is common practice to carry out initialization and self-test of RAM on startup. It may be obvious that initialization of NVRAM (which should contain useful data retained from earlier) may well be illogical, but self-test sounds reasonable. After all, failure of RAM chips (like most electronic devices) almost always occurs on power-up, so a test before beginning to use them seems only prudent. Of course the battery-backed NVRAM hasn't been powered up, but it has been pulled out of a low-power "sleep" mode, and that amounts to the same thing.

The problem is that a self-test of RAM necessarily corrupts (normally only on a temporary basis) the RAM locations under test. If you carried out a test of NVRAM on startup, and it was interrupted (e.g., by a bouncing power supply switch) the data is corrupted. To overcome this problem, you can exploit the fact that memory chip faults almost never result in the failure of a single byte of storage; most likely a particular bit in a whole sequence of bytes gets stuck. Consequently, you only need to test a small number (maybe just one) of the locations, which are dedicated to this use. Testing one byte per chip is reasonable. It would not be too hard to figure out suitable addresses. If you have information about the architecture of the chips, you can consider a test of a representative cell on each row and/or column, but that might be just a little "over the top."

The second problem is the classic "chicken and egg" dilemma. Since the NVRAM is supposed to survive power failure, the data should not be initialized on startup. The first time that the equipment is powered up, the NVRAM will contain random data, just like regular RAM. How can you recognize that the data has not been set up? An additional consideration is the possibility of battery backup failure or a power glitch, which corrupts the data. There are two possible corrective measures:

- Include a nonrandom byte "signature" to indicate the NVRAM is set up.
- Use a checksum to ensure the data has not been corrupted.

The signature can be used under any circumstances; the checksum is most appropriate when the data is not updated too often (e.g., setup parameters) since it must be

recalculated each time the data is updated. Care should be taken with the checksum calculation to avoid any locations that are dedicated to self-testing, as described previously.

Here is an example testing code:

```
#define SIG 0
#define CHK 4
#define NVR_SIZE 256

extern unsigned char nvram[NVR_SIZE];

int nvr_check()
{

   int i;
   unsigned char sum = 0;

   if( nvram[SIG] != 0 ||                    /* check signature */
      nvram[SIG+1] != 0xff ||
      nvram[SIG+2] != 0x55 ||
      nvram[SIG+3] != 0xaa )
         return (-1);                        /* failed */

   for (i=CHK; i<NVR_SIZE; i++)              /* check checksum */
      sum += nvram[i];

   if (sum != 0)
   {
      nvr_init();  /* failed */
         return (-1);
   }
   else
         return (0);                          /* passed */
}
```

The C function, `nvr_check()`, is designed to check the NVRAM signature (`00`, `$FF`, `$55`, `$AA`) and verify the checksum's integrity. The external array `nvram[]maps to` the NVRAM. No attempt is made to accommodate self-test cells. If the NVRAM passes the tests, 0 is returned. If it fails, the function `nvr_init()` is called and –1 is returned.

Updating the checksum on an area of NVRAM can be a chore, but C++ can make it very straightforward. All that is required is the definition of a new data type (i.e., class), which looks like a regular array to the user. By overloading the

appropriate operators, the updating of the checksum could be totally automatic. In addition, the constructor function can take care of the verification of the signature and checksum.

Persistent memory is commonly provided by use of flash. This is very cost effective and reliable but brings a slew of other programming issues, which are beyond the scope of this article. The upcoming technology is MRAM (magnetic RAM), which promises to replace both flash and SRAM, having the nonvolatility of one, with the speed of the other.

### Shared Memory

In an embedded system that includes a number of microprocessors, a common form of interface between them is an area of shared memory, often called "dual port RAM." This area provides a high-speed communications medium with a high degree of flexibility. Although simple in concept, at least three problems must be addressed:

- Since both microprocessors are running simultaneously, some data may be read by one before it has been written in its entirety by the other.
- The data representation (e.g., byte ordering and floating point format) may not be the same when two different processors are involved.
- The address of the memory may not be the same from the viewpoint of both processors. For example, a 4 K shared memory area may extend from $1000 to $1FFF on one side and from $23000 to $23FFF on the other.

All three problems can be readily overcome. The first one can be addressed by defining a message-passing scheme. Perhaps a single byte is set nonzero when a valid message has been completely loaded elsewhere. The receiving chip clears the byte when all the data has been read, thus completing the handshake. Some devices used to implement dual port RAM provide the alternative of using an interrupt-driven handshake.

To solve the second problem, it is necessary to agree which representation will be used in the shared memory; one method is for the "losing" chip's software to convert all transferred data. I encountered such a problem many years ago when I was using a PDP11 mini-computer linked by shared memory to a TMS9900 microprocessor. Those two CPUs have different byte ordering (i.e., the address of the most significant byte of a word may be lower or higher than that of the least significant) and differing floating-point formats. I settled on the 9900 format for data in the shared RAM even though it could be argued that it was the stranger one. (In the manual for that chip, the bits were numbered 0 for the most significant and 15 for the least; I guess the author hadn't heard of powers of two!) Byte-ordering differences are still common today; just try connecting an 8086 and a 68000. Most RISC chips even have the ability to run in two different modes, to facilitate compatibility with other devices. The modern term for memory architecture is "edianity" (or "endianness")—a device is either "big-endian" or "little-endian."

The different addresses mentioned in the third problem is simply a linking issue and is really both unsurprising and a source of only a little confusion.

## Conclusions

Handling specialized, exotic types of memory need not be a problem when the relevant issues are correctly addressed in the software. As always, the choice of development tools may be critical. At the simplest level, being able to locate appropriate data structures at the correct addresses is necessary. At a more complex level, the use of C++ classes enables special memory to be treated quite transparently.

## 3.2 Self-Testing in Embedded Systems

*This article is based upon one that I wrote for* NewBits *in 1993. It is fair to say that failure in an embedded system is at least as likely now as it was then—maybe more so. So the issues discussed in the original article are still important. (CW)*

Ever since the early microprocessors were first used in the design of embedded systems, code has been incorporated to ensure hardware integrity. Logically, if a complex board has some intelligence, it should be able to perform self-testing. With careful design, diagnostic software can detect most of the likely failures in an embedded system. Only the unlikely failure of the CPU or ROMs storing the program precludes any testing being performed at all.

### Memory Testing

A clear candidate for testing is the system memory; its integrity is critical to system operation. Since memory is readily accessible to the CPU, implementing tests is quite straightforward. Some thought is required to design appropriate and safe tests, which may be applied in each of a variety of situations.

#### *RAM—Startup*

RAM chips are like light bulbs (and most electrical and electronic components): they most commonly fail on power-up. The fact that the RAM does not contain any useful data on startup indicates an ideal opportunity to perform a thorough test. The most common form of test used at this time is a "moving ones" (or "moving zeros") test. Specifically:

1. Clear all bits to zero.
2. Set a single bit to one and check all the other bits to ensure they are still set to zero.
3. Clear the bit and check all bits to ensure that they are still set to zero.
4. Repeat the test for each bit.

The following code implements this algorithm:

```
#define RAM_SIZE 0x1000
extern char RAM[RAM_SIZE];

register int bit, ramloc, count;
register char mask;

for (count=0; count<RAM_SIZE; count++)
   RAM[count] = 0;
```

*Continued*

```
for (ramloc=0; ramloc<RAM_SIZE; ramloc++)
{
    mask = 1;
    for (bit=0; bit<8; bit++)
    {
        RAM[ramloc] = mask;
        if (RAM[ramloc] != mask)
            fail();
        for (count=0; count<ramloc; count++)
            if (RAM[count] != 0)
                fail();
        for (count=ramloc+1; count<RAM_SIZE; count++)
            if (RAM[count] != 0)
                fail();
        RAM[ramloc] = 0;
        for (count=0; count<RAM_SIZE; count++)
            if (RAM[count] != 0)
                fail();
        mask <<= 1;
    }
}
```

It is assumed that the array `RAM` has somehow been located over the RAM space and that the constant `RAM_SIZE` correctly reflects the size of the memory. The function `fail()` is called if the test fails.

However, these tests have two problems:

- The testing code must itself not use any RAM. All variables must be held in machine registers, and all functions called by the memory test must be inlined to avoid using the stack. Both these requirements may be met by use of a modern optimizing compiler, which will make good use of machine registers and perform inline optimization of small functions.
- Testing takes a long time. Even a high-performance microprocessor can take minutes or hours to complete such a test. Time can be saved by testing each RAM chip separately. Failures that cause a bit to affect other bits in the same chip are not likely to affect bits in other chips. Only memory bus problems cause these failures, which prevent the program from running anyway.

### *RAM—Testing on the Fly*

Once the application code is running, it is clearly impossible to perform any RAM tests that would corrupt the data held there. The only feasible kind of testing is to check each byte, word, or long word in turn, having saved its data in a machine register first so that it may be restored. This test detects two types of failure:

- Sticky bits, which stay set to zero or one, whatever is written to them
- Adjacent bit crosstalk, which can occur with certain memory architectures

The following code performs such a test:

```
#define RAM_SIZE 0x1000
extern char RAM[RAM_SIZE];

const char tests[] = { 0, 0xff, 0x55, 0xaa };

int ramtest()
{
    register int testnum, ramloc;
    register char save;

    for (ramloc=0; ramloc<RAM_SIZE; ramloc++)
    {
        save = RAM[ramloc];
        for (testnum=0; testnum<sizeof(tests); testnum++)
        {
            RAM[ramloc] = tests[testnum];
            if (RAM[ramloc] != tests[testnum])
                return (1); /* test failure */
        }
        RAM[ramloc] = save;
    }
    return (0); /* test success */
}
```

Test values of `$00` and `$FF` are used to check for sticky bits; `$55` and `$AA` are used to check for crosstalk. Once again, all variables must be placed in machine registers, and no function calls are made during the test. This test can be run from time to time or may be initiated by an operator action. In a multitasking environment, it can be run continuously as a background task; see the reentrancy considerations discussed in the section on multitasking below.

### *PROM—Checksum*

In-service failure of program memory (ROMs, PROMs, or flash memory devices) is unusual, but they may be subject to human error. If there is a set of memory chips containing application code and constant data—and there usually is more than one—the possibility exists that one or more devices will be of the wrong revision, be incorrectly programmed, or be inserted the wrong way around or in the wrong sites. If the device containing the startup code is correct, a power-up self-test can perform a checksum verification on the memory and detect many problems.

The checksum algorithm must be chosen carefully. Most would not detect devices in the wrong sites, because their contents would still add up. A cyclic redundancy check (CRC) or carefully designed diagonal parity algorithm are possible solutions.

The insertion of the checksum into the memory image has to be done after the linker has generated the absolute file. A utility can be written that reads the S-records (or Intel hex or whatever), calculates the checksum, and generates an additional record to locate the value at a known memory address.

#### *Special Memory Types*

Special kinds of memory require special care in the design of test algorithms. With non-volatile RAM, conventional RAM tests cannot be performed because the system may be powered down while the memory is corrupt. The slow access time of flash RAM can present other problems.

In addition, shared RAM cannot be tested conventionally because its corruption would most likely interfere with the interprocessor protocol. The solution is to incorporate a self-test sequence into the protocol, letting each CPU write and read from the shared memory. In addition, such a sequence enables each CPU to verify the integrity of the other—two tests for the price of one.

### Input/Output Devices

Since all I/O devices are unique, it is hard to make any useful general comments. The testing of an I/O device generally requires a loop back facility to be included in its design. Loop back permits the value sent to an output port to be read back in and verified or for test input to be sent to input ports.

### Multitasking Issues

In a multitasking environment, where a real-time kernel, RTOS, or interrupt service routines are used, additional care must be exercised in implementing self-tests. With an RTOS, additional tests may be possible.

#### *RAM Testing Reentrancy*

The ongoing testing of RAM in multitasking environments is a problem when an interrupt occurs while a location under test is corrupt. The only solution is to disable interrupts for the short period of time that the location does not hold valid data.

#### *Stack Limits*

When using an RTOS, each task typically has its own stack. The size of each stack must be specified but is hard to determine. If you can calculate how much stack space the deepest nested function call requires (assuming there is no recursion) and add to that the maximum stack requirement of any interrupt service routines (which might "steal" some stack space), the allocation is likely to be too great. The usual practice is to make educated guesses and increase the allocations after a crash.

A simple self-testing technique that helps prevent crashes is the use of guard words at the top of the stack for each task. These words are set to a particular value and monitored at regular intervals. When these words become corrupted, an error is declared: that stack is on the verge of overflow. The most suitable value for guard words is zero, as mostly nonzero addresses are pushed onto the stack.

### Watchdogs

A common way to monitor CPU and software integrity is to incorporate a watchdog timer circuit into the hardware design. These can come in two forms:

- A reactive watchdog asserts a specific interrupt and expects a specific response, which arms it for another period. This type of watchdog tests whether the CPU is functioning but does nothing to monitor the status of the software. The mainline code could be in an unintended loop, interrupted only to service the watchdog.
- A proactive watchdog requires the program to access a particular address at a minimum frequency. Watchdog assertions are placed at suitable strategic positions in the main software.

### Self-Test Failures

What action should self-test routines take when errors are located? What should a watchdog do if it times out?

The answers to these questions will vary from one embedded system to another. A common practice is to perform a reset—return the code to the power-up state. While this practice may be effective in correcting a random RAM glitch caused by a power spike, for example, it hides the details of a number of other faults. An indication of the nature of the failure should be reported or displayed. If this is not possible, at least an error code should be stored somewhere, so that an engineer using an infield debugger can inspect the value and identify the fault.

### Final Points

To write self-testing code, it is critical that the C code is translated into very efficient machine code and that special requirements (such as register usage, function inlines, and memory locating) are properly accommodated by your development tools.

## 3.3 A Command-Line Interpreter

*For some years before joining Microtec, I had been interested in the Forth programming language. One of the first embedded systems that I worked on used this language, and I remained fascinated by its possibilities. In this article, which is based upon one I wrote for* NewBits *in 1996, I applied some of the key ideas of a threaded interpretive language to a user interface. (CW)*

Two terms are commonly confused in the world of embedded systems: "debugging" and "diagnostics." The reason for the confusion is quite subtle, and we will come to that shortly. First of all, let's define the terms. *Debugging* is the process of verifying that code performs the function for which it was designed, and if not, fixing it accordingly. *Diagnostics* are components of an application program that enable the user to confirm or ensure the well-being of the software and/or hardware after deployment.

The two terms should not be confused because debugging tools should be present only during the development phase. After deployment, they should leave no trace; just the application code (including any diagnostics) should be taking up space on the target. The reason for confusion is that some development tools do not conform to this ideal and consume valuable target resources long after they have ceased to be useful. It is in the interests of the vendors of such tools to propagate this confusion.

In this article, we are going to look at the implementation of diagnostics on an embedded system. It must be emphasized that this process has very little to do with the debugging phase; it is a means of implementing a post-deployment testing facility.

### Embedded Systems Diagnostics

With a simple embedded system, a malfunction is easy enough to identify—the system no longer works. The cause of the fault may or may not always be apparent, but the fact that a fault exists is quite clear.

More complex systems present a greater challenge. With a high level of functionality, a system can develop a potentially catastrophic fault, without showing any immediate symptoms. It is essential that such systems incorporate protection against such a mishap.

The subject of self-testing was covered in the previous article in this chapter, so we will not dwell on it here. Suffice it to say that self-testing takes three forms:

1. Tests executed on reset
2. Tests executed continuously or periodically during the operation of the system
3. Tests initiated by the operator—that is, diagnostics

Diagnostics concern us here.

Communication with the operator is an obvious requirement. If the system has a screen/keyboard, then these components may be used. Otherwise, a connection to an external terminal (or computer) is required. Maybe a spare serial line is available, or one could be included in the design specifically for this purpose. Another possibility is that a

serial line, which is normally employed by the application, can be used when the system is in "diagnostic mode."

### An Embedded System Comes Alive

A difficulty (psychological, at least) with developing an embedded system is that an incomplete application, even thought it may well be executing correctly, does not appear to be doing anything. A much greater feeling of satisfaction is achieved when the software "says its first word," which is a major argument for implementing diagnostics early in the development cycle. Apart from this, working diagnostics can also help with debugging and integration.

### A Command-Line Interpreter—Requirements

There are broadly two ways that a keyboard/screen user-interface to diagnostics may be implemented:

- **Single key action:** Commands are implemented as single keys, which are actioned immediately. Other data that is required is obtained by prompting the operator. This approach may be good for very simple requirements, because a "state machine" approach to the implementation may be taken. The downside is that the structure is not intrinsically extensible.
- **Mnemonic commands:** Complete command lines are entered by the operator and only actioned when the Return key is pressed. This approach has greater flexibility because it may be extended easily. It has the additional advantage that the programmer is not burdened with line editing, because the library functions can take care of these matters. A possible downside is that interpretation of the commands may require complex programming.

Although the first approach is valid, we will address the issues involved in the second one in this article.

As with many common software requirements, the inclusion of a command-line interpreter (CLI) presents the designer with two options: buy it or build it. It is almost always better to buy a software component, if such a product exists. For larger embedded systems, a good approach is to include a UNIX-style shell as the CLI—these are available for popular RTOSs. For a smaller system, the overhead of such a sophisticated solution may not be bearable. The only option is to implement a simple, custom CLI.

### Designing a Command-Line Interpreter

The operation of a CLI occurs in three phases:

1. Recognize the command.
2. Collect and process any required parameters.
3. Execute the command.

Phase 1 is quite straightforward. All that is required is a table of valid commands, against which a sequence of string comparisons is made. A little care with case sensitivity is the only real precaution that is necessary. Phase 3 is also quite easy. A table of function addresses (i.e., pointers to functions) is constructed, with entries corresponding to each command. This table is indexed as a result of the string comparisons performed in phase 1. The only challenge to the programmer is to untangle the intractable C syntax!

Most problems occur in phase 2. Here are some of the issues:

- How many parameters (if any) are required?
- What kind of data is required for each parameter?
- Are any parameters optional?
- How much flexibility in entry format can be provided to the operator?
- How are parameters passed to the function that implements the command?

The remainder of this article will propose a design for a CLI that addresses these issues, resulting in the complete code for a real implementation.

### A CLI Implementation

The CLI described here exhibits the following behavior:

- When ready to accept a command line, it prints a `$` prompt.
- It accepts text "blindly" until the Enter key is pressed.
- An empty command line (no characters or just spaces) is ignored and a new prompt issued. This means that repeatedly pressing Enter just causes more prompts. This is useful when checking the communications line.
- A command line is considered to be a sequence of tokens, separated by spaces. (Incorporating support for other or additional separators is straightforward.)
- When a line has been received, the tokens are processed, in turn, from left to right. (Although, if preferred, you could readily adapt this approach to parse right to left.)
- Each token is checked to find out if it is a valid command. The first three characters and the length of the command are considered significant. So `PRINT` and `PRIME` are considered the same, whereas `PRINTER` is considered different. The comparison is not case sensitive (all processing is performed using lowercase), and any nonspace ASCII characters may be used. If the command is valid, its corresponding function is executed.
- If a command is invalid, an attempt is made to process it as a number. If this attempt is successful, the value is pushed onto the "parameter stack," where it may be retrieved by subsequent commands. If the token does not represent a valid number, an error message is displayed, and the remaining tokens on the line are not processed.

This behavior yields a very flexible operator interface. Here are some possible command lines, with the processing sequence:

`6 LED-FLASH`: The `6` results in this value being pushed onto the parameter stack. The function corresponding to the command `LED-FLASH` is executed and presumably expects a value on the stack.

`5000 5015 DUMP`: The `5000` and `5015` each place a value on the stack. The function corresponding to `DUMP` is executed and expects two parameters on the stack: at the top it expects the last address and below it the first address or a range to be dumped in hexadecimal format.

`MEM DUP 15 + DUMP`: The `MEM` causes a function to be executed that places an address on the stack. `DUP` causes a function to be executed that makes a copy of the top stack item. `15` pushes the value on the stack. + adds together the top two stack items. `DUMP` functions as before.

The user interface may be used in a simplistic manner by an operator with limited training. A more sophisticated user can take advantage of the flexibility.

Readers with knowledge of the Forth programming language may find this approach familiar. Although this CLI is far from being a full-blown Forth interpreter, it could form the basis for the design of such a tool.

### CLI Prototype Code

All the code for a prototype CLI is included in this section. The program is written in ANSI C and should build with any compliant compiler. In tests, using an efficient 68000 compiler, the entire program resulted in less than 1 K of code. This represents a very small overhead to gain a powerful facility.

Of course, there are library functions too, but they may well be present in the application anyway. The most significant one is `printf()`, but its use may be circumvented if the memory hit is too great.

Here is the command table, which may easily be extended:

```
struct dtable                          /* command dictionary entry */
{
    union
    {
        struct
        {
            unsigned char len;
            char word[3];
        } bytes;
        unsigned long bits;
```

*Continued*

```
    } id;
    void (*func)();
};

#define ENTRIES 10                    /* command dictionary definition */

void dot(void);                       /* command functions */
void store(void);
void at(void);
void query(void);
void plus(void);
void dump(void);
void quit(void);
void v1(void);
void v2(void);
void v3(void);

struct dtable dictionary[ENTRIES] =
{
    {1, '.', 0, 0, dot},       /* . command */
    {1, '!', 0, 0, store},     /* ! command */
    {1, '@', 0, 0, at},        /* @ command */
    {1, '?', 0, 0, query},     /* ? command */
    {1, '+', 0, 0, plus},      /* + command */
    {4, 'd', 'u', 'm', dump},  /* dump command */
    {4, 'q', 'u', 'i', quit},  /* quit command */
    {2, 'v', '1', 0, v1},      /* v1 command */
    {2, 'v', '2', 0, v2},      /* v2 command */
    {2, 'v', '3', 0, v3}       /* v3 command */
};
```

Each entry includes the first three characters and the length of the command, coded into an `unsigned long` (32 bits) by means of a `union`. This technique is portable because it makes no assumptions about the layout of structures.

Here is the `main()` function:

```
void main()
{

    char str[100];
    char *tok;
    unsigned long ident;
    unsigned match, error;
```

*Continued*

```
    unsigned i, j;
    struct dtable scratch;

    while (1)
    {

       printf("\n$ ");  /* prompt */
       gets(str);       /* get line of input */

       tok = strtok(str, " ");
       if (tok)
          do            /* compute 32 bit identifier: */
          {
             scratch.id.bytes.len =
                (unsigned char)strlen(tok);
             for (i=0; i<3; i++)
                if (i<strlen(tok))
                   scratch.id.bytes.word[i] =
                      (char)tolower(tok[i]);

                else
                   scratch.id.bytes.word[i] = 0;
             ident = scratch.id.bits;

             match = 0;             /* scan dictionary: */
             for (i=0; i<ENTRIES && !match; i++)
             {
                if (ident == dictionary[i].id.bits)
                {
                   /* match - execute function */
                   match = 1;
                   (*dictionary[i].func)();
                }
             }

             error = 0;             /* no match -try a number */
             if (!match)
             {
                if (!trypush(tok))
                   error = 1;
             }
          } while ((tok = strtok(NULL," ")) && !error);
    }
}
```

An outer loop displays a prompt and accepts command lines. The inner loop breaks down the line into tokens and decodes them into commands and parameters. The parameter stack definition, which is simply an `int` array, is shown here, along with the associated utility functions.

```
#define STACKSIZE 100          /* local parameter stack */
int stack[STACKSIZE];
int sp = 0;                    /* stack pointer */

/*** Utility Functions ***/

/* trypush() - try to convert token to number a push on stack
returns true on success */

int trypush(char *str)
{
    unsigned i;
    int val = 0;

    for (i=0; i<strlen(str); i++)
    {
        if (isdigit(str[i]))
            val = val * 10 + str[i] - '0';
        else
        {
            printf("%s???\n", str);
            return (0);
        }
    }
    push(val);
    return (1);
}

/* push() - push a value onto the stack */

void push(int n)
{
    if (sp == STACKSIZE)
    {
        printf("Stack full!!!\n");
        exit(-1);
    }
    stack[sp++] = n;
}
```

```
/* pop() - pops a value off of the stack [else returns 0] */

int pop(void)
{
   if (sp == 0)
   {
      printf(" Stack empty!!! ");
      return (0);
   }
   return (stack[--sp]);
}
```

Here is the code for commands implemented in this example:

```
/*** Command functions ***/

/* . command - pop value off the stack and display */

void dot(void)
{
      printf(" %d ", pop());
}

/* @ command - pop address off the stack and push the data at which it
points */

void at(void)
{
      push(*(int *)pop());
}

/* ! command - pop an address and some data off the stack and store the
data at the address */

void store(void)
{
      int addr;

      addr = pop();
      *(int *)addr = pop();
}

/* ? command - effect the @ and . commands in sequence */
```

*Continued*

```
void query()
{
    at();
    dot();
}

/* + command - sum top two stack items and push the result */

void plus(void)
{
    push(pop() + pop());
}

/* dump command - pop last and first address and display that range of
memory addresses in hex */

void dump(void)
{
    int last, i, j;

    last = pop();
    for (i=pop(); i<last; i+=8)
    {
        printf("%04X ", i);
        for (j=i; j<i+8; j++)
            printf(" %02X", *(char *)j);
        printf("\n");
    }
}

/* quit command - pop code and exit if correct */

void quit(void)
{
    if (pop() == 999)
          exit(0);
}

/* v1 command - push the address of the first user variable */

void v1(void)
{
    push((int)&vars[0]);
}
```

```
/* v2 command - push the address of the second user variable */

void v2(void)
{
    push((int)&vars[1]);
}

/* v3 command - push the address of the third user variable */

void v3(void)
{
    push((int)&vars[2]);
}
```

It should be noted that these commands, which allow very free access to the application's memory space, would be very unlikely candidates for inclusion (or at least documentation) in a real system. The actual commands would be much more application-specific. These commands are simply presented as examples:

- `.` (dot): Removes the top value from the parameter stack and displays its value (in decimal).
- `@` (at): Removes the top value from the parameter stack and, treating it as an address, pushes the data found at the location to which it points.
- `!` (bang): Removes the top value from the parameter stack and, treating it as an address, removes the next value off the stack and stores it at that address.
- `?`: Combines the effect of the `@` and `.` commands: it removes the top item from the stack and, treating it as an address, obtains the data to which it points and displays its value (in decimal).
- `+`: Removes the top two items from the stack and pushes their sum.
- `dup`: Makes a copy of the top item on the stack and pushes it.
- `dump`: Removes the top value from the parameter stack and treats it as the last address of a range. The next item is removed and treated as the first address of a range. The range of memory addresses is then displayed in hex.
- `quit`: Removes the top value from the stack and, if it has the value 999, terminates the CLI (otherwise, it does nothing).
- `v1`, `v2`, and `v3`: Push the address of the first, second, and third user variables, respectively.

## Conclusions

The implementation of a flexible command-line interpreter facilitates easy-to-use post-deployment diagnostics. The code may readily be written in ANSI C, which by use of a modern optimizing compiler, results in a very small memory overhead.

## 3.4 Traffic Lights: An Embedded Software Application

*When I wrote the original article on this topic for* NewBits *in 1996, I was responding to a common request: "Can you show me an embedded application that does not require reams of code and does something that I understand." I figured that we all cross the road sometimes. (CW)*

The objectives of this article are ambitious: to address a number of key techniques in the development of a simple embedded system. Issues to be covered include:

- Basic application design
- Building and debugging the program
- Programming and debugging interrupts

### The Application

The example program is a simplified traffic-light-controlled pedestrian crossing system. These systems work in various ways in different countries. This example is based upon the system used in the United Kingdom:

1. The walker presses a button.
2. The traffic lights cycle through yellow to red, and the crossing light changes from "Stop" to "Go".
3. After a delay (while the walker crosses the road), the yellow light and "Go" are set flashing.
4. After a further delay, the traffic light is restored to green and "Stop" is illuminated.
5. Then the button (the one the walker pushed in step 1) is inoperative for a period of time (to avoid too frequent interruption of traffic).

In this example, some details are omitted for simplicity: traffic sensing, pedestrian detection, sound generation, and so on.

### Hardware Configuration

The controller is based upon a microcontroller, which has two memory-mapped I/O ports. At `$1000` is the output port that controls the (five) lights. The allocation of bits is illustrated in Figure 3.1; setting a bit illuminates the bulb. The most significant three bits should not be set.

At `$2000` is the input port that corresponds to the press button; bit 0 is set (and latched) when the button is pressed, and it may be cleared by writing 0 to the port. A 250-mS timer interrupt is set up on a vector at address `$100`.


*Figure 3.1: Output port at $1000*

### Program Implementation

The logic of the program consists of two parts: a main loop, which awaits the button pressing and sequences the lights, and an interrupt service routine, which deals with time delays and lamp flashing.

### Main Loop

The structure of the main loop closely maps to the previous behavioral description; the pseudo-code is shown is as follows:

```
initialize lights to Green/Stop;
initialize button;
repeat
   wait for button to be pressed;
   clear button register;
   set lights to Yellow/Stop;
   delay 2 seconds;
   set lights to Red/Stop;
   delay 2 seconds;
   set lights to Red/Go;
   delay 5 seconds;
   set lights to flashing Yellow/Go;
   delay 3 seconds;
   set lights to Green/Stop;
   delay 1 minute;
end
```

Here is the actual C code:

```
/* Bit patterns for lights */

#define GREEN 1             /* traffic */
#define YELLOW 2
#define RED 4
#define FLASH_YELLOW 0x200

#define GO 8                /* pedestrian */
#define STOP 0x10
#define FLASH_GO 0x800

/* Control word for lights */
volatile int lights;
```

*Continued*

```
/* Button read/clear definitions - bit 0 of $2000 */
#define READ_BUTTON ((*(int *)0x2000) & 1)
#define WRITE_BUTTON (*(int *)0x2000)

/* Timing facilities */
volatile int countdown;
#define WAIT(s) countdown=s*4; while(countdown) ;

main()
{
    /* Initialisation */
    lights = GREEN   | STOP;          /* set lights */
    WRITE_BUTTON = 0;                 /* init button */
    asm(" MOVE #$2000,SR\n");         /* enable interrupts */

    /*** Main loop ***/

    while (1) /* go round for ever */
    {

        /*** Wait for button ***/
        while (!READ_BUTTON)
               ;                      /* just hang */
        WRITE_BUTTON = 0;

        /*** Change lights to allow crossing ***/
        lights = YELLOW  | STOP;
        WAIT(2);
        lights = RED STOP;
        WAIT(2);
        lights = RED | GO;
        WAIT(5);                      /* crossing now */

        /*** Change lights back ***/
        lights = FLASH_YELLOW   | FLASH_GO;
        WAIT(3);
        lights = GREEN   | STOP;
        /*** Let traffic flow for a while ***/
        WAIT(60);
    }
}
```

This code is quite simple. There are a couple of points to note:

- Access to the button is by means of two "memory variables": `READ_BUTTON` and `WRITE_BUTTON`. These macros handle the syntactical untidiness of accessing a memory location (I/O port) directly in C.
- A significant amount of work is done by the timer interrupt (which is discussed in more detail in the section that follows). The main loop communicates with the interrupt service routine (ISR) by means of two global variables: `lights` and `countdown`. The variable `countdown` is in turn is accessed by the macro `WAIT()`.

### Interrupts

The only interrupt in this system is a 250-mS timer. The ISR supports two facilities:

- Countdown timer for performing time delays in the main loop
- Output to the lamp-driving hardware, taking care of flashing

Here is the code for the ISR:

```
/* Lights port - $1000 */
#define LIGHTS (*(int *)0x1000)

interrupt void timer()
{
    int flash_bits;

    if (countdown != 0)              /* effect timer */
         countdown--;

    flash_bits = lights >> 8;        /* update lights */
    lights ^= flash_bits;
    LIGHTS = lights & 0xff;
}

/* Interrupt vector */
#pragma asm
    ORG $100
    DC.L _timer
#pragma endasm
```

Note that the function is declared as an ISR by using the keyword `interrupt`, which is a common extension to ANSI C. This ensures that context-saving code is generated and the function ends with an `RTE` instead of `RTS`. The function is necessarily declared to be of type `void`, with no parameters.

Also shown is the definition of the interrupt vector, using an assembly language insert. There are several other methods by which this may be achieved. For example, using an array of pointers to functions, which is located at link time. The method chosen has the advantage of brevity and simplicity. Anything to do with interrupt vectors and interrupt control is necessarily very processor-specific.

### Time Delays

The time-delay facility is implemented very simplistically. A global variable, `countdown`, is monitored by the ISR. If it is nonzero, it is decremented by one on each tick. To effect a delay, the code in the main loop simply loads this variable with the number of 250-mS intervals that it wished to delay for and waits for it to become zero. The `WAIT()` macro takes care of the details. Note that countdown is declared `volatile`; this ensures that the compiler generates code to actually access the variable's memory location each time it is referenced (instead of optimizing it into a register).

This timing mechanism yields a mean timing precision of +/– 125 mS, which is quite acceptable for this application.

### Lights

The ISR has exclusive access to the lamp-driving hardware. The main loop uses the global variable `lights` (again declared `volatile`) to indicate its requirements. The least significant byte of this variable maps directly onto the output register (as shown in Figure 3.1). The most significant byte is in the same format but indicates lamps that are to be flashed on each timer tick. The ISR code updates the lamp-driving hardware every tick, having toggled the bits of the flashing lamps accordingly.

### Using Global Variables

The technique of using global variables as a means of communication between tasks or between mainline code and ISRs is widely considered to be unreliable. To use it safely, certain conditions must be met:

- The application must be simple—this one is.
- The variables must be declared volatile—they are.
- The protocol for their use (i.e., which code writes and reads them and when it can do it) must be clearly defined—it is.

## 3.5 PowerPC Assembler

*Modern embedded developers write very little assembly language code—unlike in the early days, when almost everything was coded that way. It is, however, useful to have an understanding of a processor's architecture, which is reflected in its assembly language. It may only be used to verify what the compiler is actually doing, but that is a valuable skill. Fluency in writing assembler may not be a requirement, but some familiarity is useful. I draw an analogy with the French language—understanding enough to read a menu is well worthwhile, but it is a long way from being able to make everyday conversation. When Pravat Lall and Paul Carroll (who were both software engineers at Microtec) wrote an article about the PowerPC toolkit for* NewBits *in 1996, this family of devices was new and unfamiliar. Now, it is very much mainstream. So there is benefit to be gained from insight into the PowerPC assembler. The article is also a general guide to assembly language concepts, many of which are not specific to the PowerPC. Since the original piece was written, Motorola Semiconductors became Freescale, but I have left the nomenclature in this article unchanged. (CW)*

The assembler and linker are part of most, if not all, embedded applications development. Sometimes the assembler simply processes the output of a higher-level tool, such as a C or C++ compiler. In most embedded systems, however, low-level access to peripherals or processor resources requires the direct use of assembly instructions. An assembler toolkit is likely to include the assembler itself, a linker (or "linker/locator"), and a librarian and will support processing of both compiler output and handwritten assembler. This article describes some aspects of the PowerPC processor family and the functionality of a typical PowerPC assembler toolkit.

### The PowerPC Processor

The PowerPC processor family was developed jointly by IBM and Motorola. The PowerPC RISC processor was designed as replacement technology for the 68xxx family, which could not easily be adapted to work at the high clock speeds used by today's processors.

#### *The PowerPC Family*

The PowerPC family consists of a number of subfamilies, resulting in a vast number of different devices being in use. The devices are targeted at different application areas—both desktop and embedded—and cover both conventional CPUs and highly integrated microcontrollers. PowerPC cores have even been integrated into programmable logic (FPGA) devices.

All of the devices have a common architectural heritage and benefit from a portfolio of standards, which address many aspects of software development. By adhering to these standards, development tools facilitate the maximum level of interoperability. These standards also make it easier to solve various problems that might exist in an embedded system, such as endianity of data or efficiency of memory references.

### The PowerPC EABI

The PowerPC EABI (Embedded Application Binary Interface) establishes a standard interface for interoperability between software components and maintains mixed-vendor compatibility. The PowerPC EABI standard is built upon the PowerPC Application

Binary Interface (ABI) supplement to the System V Release 4 standard (SVR4), which describes the operability of a dynamic, nonembedded PowerPC environment. The PowerPC EABI removes several features from the ABI standard that are not appropriate for embedded systems and adds new functionality in their place.

In broad terms, any compliant set of tools may be used in conjunction with any other compliant producer or consumer tools.

### Big- and Little-Endian

Unlike the Motorola 68xxx family and many other older architectures, PowerPC processors can operate in either big- or little-endian modes, with the default being big-endian. Toolkits tend to support both modes. Little-endian support presumes that the processor will switch to this mode immediately upon startup, and no further switching between modes will occur. Note that mixed endianity modes are not supported by the Executable and Linking Format (ELF).

### Small Data Areas

Since PowerPC processors have a RISC architecture and all instructions are 4 bytes in length, it is not possible to form a complete 32-bit value to address a specific memory location with a single assembly instruction. For most memory accesses, two instructions containing the high and low portions of the address are required. Frequent accesses to the same memory location can be expensive if they are not controlled appropriately.

The PowerPC EABI defines some special sections, called Small Data Areas (SDA) to help solve this problem. There are three different SDAs for writable and read-only data. The purpose of an SDA is to reduce the resulting code size and improve data access time. This is accomplished by assigning a register to hold a 32-bit address that maps into the middle of a 64-K region. A signed 16-bit offset can then be used in a single instruction to access memory in that region, thus reducing the number of instructions needed to access that memory by half.

Each SDA consists of a pair of special sections. The SDA for writable data is comprised of the special sections `.sdata` and `.sbss`. The special sections `.sdata2` and `.sbss2` can hold read-only as well as writable data. The sections `.PPC.EMB.sdata0` and `.PPC.EMB.sbss0` constitute an SDA for writable data around the address `$00000000`. Each of these SDAs has a maximum size of 64 K. Data in the SDAs is accessed through a base symbol unique to each SDA that is combined with the 16-bit signed offset to reference memory locations in the SDA. The value of the base symbol is stored in designated registers for use by later memory references. The value of the base symbol for the sections `.PPC.EMB.sdata0` and `.PPC.EMB.sbss0` is always zero and does not need to be stored. The values for the other two base symbols, namely `_SDA_BASE_` and `_SDA2_BASE_`, either can be supplied by the application developer or can be computed automatically by the linker.

The computation of an appropriate base value for the SDAs is a critical task for the PowerPC linker. The linker places the paired SDA sections as close together as possible so that an appropriate base value can be computed. If all the data for an SDA does not fit within the 64-K limit or if the contents of the sections of an SDA are placed more than 64 K apart, a linker error will result.

SDAs are not necessarily supported by C and C++ compilers, because their functionality does not map well onto those languages. They are supported in standard assemblers and linkers.

#### *ELF File Format*

The file format designated for use in PowerPC applications is ELF (Executable and Linking Format). This file format was originally documented in the SVR4 ABI. Further refinements have been described in the PowerPC processor supplement to the SVR4 ABI and in the PowerPC EABI document itself.

ELF is similar to the COFF file format, except that almost everything in an object file is contained within a section: a file header, section table, and program header table. Other than these three structures, all other information is stored inside sections: the symbol table, string table (used to hold section and symbol names), relocation entries, and debug information. This partitioning greatly simplifies the ELF file format without overly restricting its use.

#### *DWARF Debug Format*

The PowerPC EABI also defines a standard for debug information in object files. The standard that has been chosen is the DWARF debug format. The EABI standard currently allows either DWARF version 1.1 or 2.0 to be used to represent debug information. The correct handling of this data by the assembler toolkit enables clear integration of any EABI-compliant compiler with any compliant debugger.

### PowerPC Assembler Toolkits

Assembler toolkits for PowerPCs are available from a variety of vendors. Most are EABI compliant, so good interoperability may be anticipated. The exact facilities, options, and mode of operation will vary from product to product, so only a broad indication may be presented here.

### The Assembler

An assembler performs the task of converting assembly language source files into ELF relocatable object modules. The assembler allows access to all of the instructions and registers of the supported PowerPC processors. Assembler directives make it possible to structure code and data so as to create a compact and efficient application.

#### *PowerPC Instructions*

Like most assembly languages, PowerPC assembly instructions are written as an instruction mnemonic and a series of comma-separated operands that are either registers or

expressions. There are both Motorola and IBM variants on these mnemonics. Some instructions have pseudo-instructions that support the more commonly used operands. Registers can generally be addressed either symbolically (such as `r13`) or numerically (such as `13`). Likewise, relocatable and external addresses can be addressed symbolically to simplify programming. Special expression operators exist to obtain the high or low 16 bits of a 32-bit address, take the size of a section or a section's start address, and get the offset of a symbol from an SDA's base.

#### *PowerPC Assembler Directives*

Various types of directives control the operation of the assembler. The types of directives available include:

- Section creation and use
- Conditional assembly
- Macro definition and use
- Repeat blocks
- Data declaration for all of the standard types (byte, half-word, word, float, double, and strings)
- Listing and object file control
- Global, external, and common symbol creation

Another useful feature is the `.USING`/`.DROP` directive pair. These directives associate a given relocatable expression with a register. They then create index-relative references based on that register for memory references that are near the location represented by the base register. This can be useful for accessing structures or for just managing memory more efficiently.

### The Linker

A linker performs the task of merging multiple object and library files together into a cohesive executable file. In doing so, the linker also assigns memory addresses to sections, resolves values for external symbols, and creates ROMable copies of initialized data.

The linker takes one or more ELF object or library archive files as input. Libraries are commonly in the SVR4 PORTAR format. The output from the linker can then be either an (absolute) ELF executable object file or an incrementally built (relocatable) ELF object file. Usually, linkers can be driven from a command line, but they tend to have more complex requirements that are best conveyed in a command file.

#### *Linker Commands*

A linker command file consists of a series of commands that provide different means of controlling the linking process. These commands are processed in order and are in effect throughout the linking process.

The command file is used mainly to control the placement of sections in memory. If no linker command file is used or no `ORDER` commands are present, the linker will assign addresses to sections in the order encountered from the input files and by section type (code, ROMable data, writable data, and uninitialized data). It also places sections in memory, starting at address `$00000000`. In most embedded systems, however, memory layout is more restrictive. The `ORDER` command makes it possible to designate the order and locations in memory for the placement of sections. This placement can be controlled simply by specifying a section type or the portion of a section from a particular object file.

### Conclusions

Even though a PowerPC is most likely to be programmed in C or C++, an understanding of underlying architecture and the capabilities of the assembler toolkit is invaluable.

# 4

# C Language

*C is still the most commonly used programming language for embedded software applications. Although a small language, it continues to challenge software engineers. Many of these challenges arise from the origins of the language, which were quite a different context from that in which it is now applied. The articles in this chapter address the topic from various perspectives.*

## 4.1 C Common

*This article is based upon one written for* NewBits *way back in 1990 by Ken Greenberg. It addresses an issue that was—and still is—a common area of confusion among C developers. Over the years, embedded tools have become more like those used for host native development in both their capability and the standards to which they adhere, but some issues, such as the handling of weak externals, persist. (CW)*

In this article, we will discuss the concept of C common, sometimes referred to as *weak externals*. A C common data object is a global variable that has not been explicitly declared to be either static or external and has not been initialized in any module. Thus, a declaration of the form:

```
int currentTask;
```

may be in C common. You can't tell by looking at only one module whether a given data object will or will not end up in C common since it may have been initialized elsewhere.

It is now common—maybe even standard practice—for software engineers to prototype their applications (or at least as much of the applications as practical) on their development platform (host). Ideally, the program should work the same way in the host-prototyping environment as in the target. The use of C common makes an embedded compiler work more like a host compiler and enables the programmer to transfer code from native to cross-compilers without making changes.

The concept of C common is closely tied to the idea of a defining point. In C, every data object is to have only one defining point. In practice (as defined by C compilers on UNIX), it is also possible to have no defining point at all, with the linker taking care of those objects not defined elsewhere. A defining point is most commonly determined by an explicit initialization. If you coded:

```
int currentTask=0;
```

then you create the defining point for `currentTask`, and no other module is allowed to initialize it. If two modules initialize the same variable, a link-time error (multiple definitions) results. Initializing a variable tells the linker that this object is an "external

definition"; it also has the side effect of placing it in a separate program section from the uninitialized variables.

Suppose you had a declaration like:

```
extern int currentTask;
```

then you would, in effect, tell the linker that this is an "external reference" to a data object defined elsewhere. If all of your modules had identical declarations, this would be a link-time error of a different sort. If all modules tell the linker that this variable is somewhere else, the variable ends up being undefined.

Variables declared as `static` do not export their names to the linker, so they are treated differently. Since they can be accessed only from within the module in which they are declared, they don't really need a defining point.

C common data objects have no defining point at all. They are not initialized in any modules. They may include the storage class `extern` in some modules (but this is essentially superfluous; the `extern` keyword is, in this context, arguably obsolete). The behavior expected by most (host) C programmers is that declarations will refer to the same variable. If all modules simply declare `currentTask` to be a variable of type `int`, then it is up to the linker to establish that they refer to the same `int`. Since the variable `currentTask` is not initialized in any modules, then the linker is responsible for allocating space for it in the uninitialized variables section. Further, if the variable is declared as:

```
long currentTask;
```

in one module, then it still refers to the same object. The linker must allocate space based upon the longest declaration found. It should be emphasized that this is *not* good programming practice!

For embedded systems developers, it has been traditional to take a somewhat different approach. In the past most C compilers have not supported C common. Any variable declared as external is a reference; any variable not declared as external is a definition. The burden is on you to make sure that all declarations are external except one.

However, the techniques used in embedded development have been changing over the last few years. Embedded tools increasingly conform to the standards of host tools, and support for C common is an example of that conformance.

In summary, Figure 4.1 illustrates the valid code constructs for the traditional and the C common approaches; Figure 4.2 illustrates the possible error conditions.

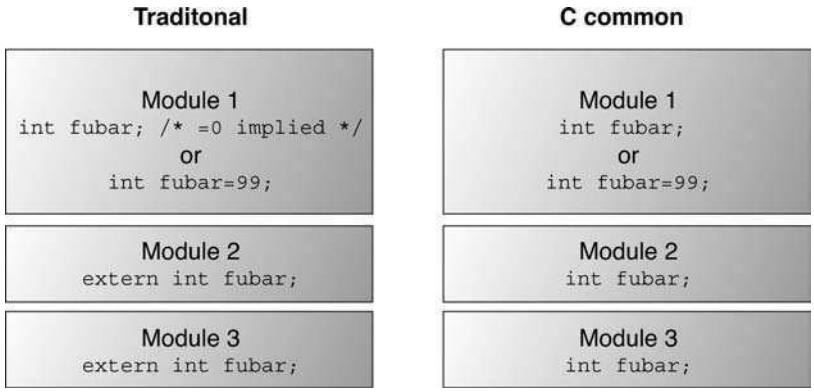



*Figure 4.1: Valid syntax*

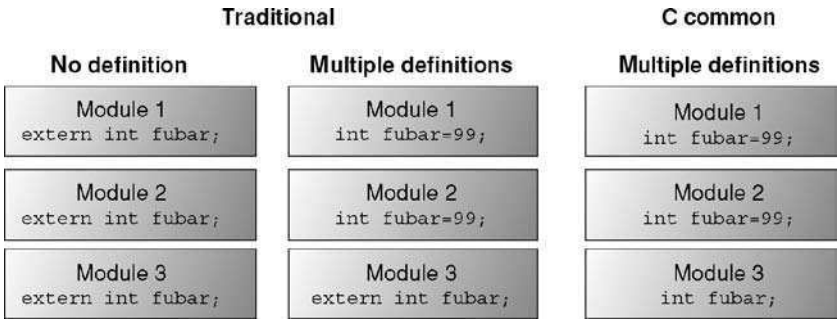



*Figure 4.2: Error conditions*

## 4.2 Using C Function Prototypes

*This article is based upon one that appeared in* NewBits *in 1990 under the title "Using ANSI C Function Prototypes"—there is no record of the identity of the original author. At that time, the ANSI standard for the C language was new, and compiler support was just beginning to appear. (CW)*

When the ANSI C language standard was first published (after a long wait), it incorporated much more than a consolidation of the language features in use at that time. A number of entirely new capabilities were added, among which were function prototypes. This feature allows programmers to specify more information about functions for improved error detection during compilation.

In discussing function prototypes, it is important to understand a three-way distinction among function declaration, definition, and calling.

- **Declaring** a function establishes a form or *prototype* for subsequent use of the function. Function declaration is optional in C (unlike C++ where it is mandatory). For example:

```
int my_function(int, char);
```

- **Defining** a function is the task of writing the program code that makes up the function, for example:

```
int my_function(int x, char c)
{
    ...
    if (c=='y')
        return (x*47);
        ...
}
```

- **Calling** a function involves its actual use as a program element, either from other functions or, in the case of recursion, from the function itself. For example:

```
z = my_function(22, 'y');
```

### Before Prototypes

Function prototypes are an extension to the optional declaration of C functions. In the original K&R definition of C, only the return type of functions could be declared before actually defining or using the function, as follows:

```
return_type my_function();
```

where `return_type` is the data type returned by the function. In the absence of `return_type`, function type defaults to `int`. This seems odd, because `void` is a more logical choice. However, `void` was not included in the language standard until ANSI, so there were backwards compatibility issues.

Note that such a declaration causes the compiler to check the return assignment of a function but makes no effort to check the types of parameters passed in a call to that function. In the absence of such parameter "prototypes," the compiler generates code assuming that all parameters are correct, leading to errors that are difficult to diagnose at runtime.

### Applying Prototypes

Function prototypes extend function declaration to include the formal parameters. For example, if the `my_function()` function, discussed in the previous section, requires two integer parameters, the declaration could be expressed as follows:

```
return_type my_function(int x, y);
```

where `int x, y` indicates that the function requires two parameters, both of which are integers. Note that the variable names `x` and `y` are optional, formal parameters. The actual parameter names in the function definition and subsequent calls need not be the same; only the types must match. Using meaningful formal parameter names is helpful in documenting the functionality.

Because the declaration includes parameters, the compiler can check whether calls to this function are made properly. If, for example, the following call is made:

```
a = my_function("hello world");
```

the compiler generates an error informing you that `my_function` was called with an incorrect number of parameters (one instead of two) and an incorrect first parameter (i.e., an integer was expected but a string was received instead).

Note that C automatically performs some forms of type conversions to ensure that actual parameters match the declared type of a function's formal parameters. Character variables (`char`), for example, are automatically promoted to integer (`int`) for parameter passing and arithmetic.

### Prototypes in Use

Using function prototypes in include (header) files can be a convenient method of preparing a specification of your code module and its external interface for others to read. Users of your code will be less reliant on browsing your source code to determine the parameters that your functions require.

The following are several examples of function prototype declarations. Note that the inclusion of parameter names is optional:

```
int uart_init(int baud, char bits, char parity);

void make_buffer(int size);

void kill_list(struct list *head);

char *look_ma(int, float, char); /* no parameter names */
```

In summary, C provides software developers with a standard language syntax to express functions' declarations in more detail. Function prototypes increase the readability of include files and allow compilers to support improved compile-time error checking.

## 4.3 Interrupt Functions and ANSI Keywords

*This article is closely based upon a piece by Ken Greenberg in* NewBits *in 1990. ANSI C was new then, but the points and principles discussed in this article still hold today. (CW)*

### Interrupt Functions

I would like to start this article with a discussion of interrupt functions. As the term implies, interrupt functions are used primarily as interrupt handlers. Traditionally, an embedded system designer would write a short routine for each interrupt in assembly language. The addresses of these routines are then inserted into the interrupt vector table. These routines save registers as necessary, call a C function to do any processing required, restore registers on return from the C function call, and return with the special "return from exception" instruction required by the specific processor. The code for each routine, and all functions that it calls, is collectively known as the Interrupt Service Routine (ISR).

Why are these short assembly language routines necessary? There are two reasons. First, every compiler has a set of registers that it is free to use as scratch registers and other registers that it is obligated to save through function calls. I like to call the first set the *scratch* set and the second the *preserve* (or *save*) set. For example, the preserve set for a particular 80x86 compiler includes the registers CS, DS, SS, SP, BP, SI, and DI; the scratch set includes ES, BX, CX, DX, and the return value register AX. If you wrote your ISR entirely in C, it is safe to assume that the preserve set registers would be as they were at the time an interrupt occurred when you were ready to resume execution of the interrupted code. However, it is just as safe to assume that the scratch set would be completely destroyed by the ISR. For this reason, these short routines will push the contents of the scratch set registers before calling a C function and will pop them before returning.

The second vital function performed by these short assembly routines is the generation of an appropriate return from the ISR. When an interrupt occurs, many processors push state information on the stack along with the return address. For example, the 80x86 family processors push a "flags" word, the contents of the CS register, and the contents of the IP register on the stack. Clearly, a normal return instruction would not clean up the stack properly, so the `IRET` instruction must be used instead.

Ideally, C programmers like to write everything in C, including their ISRs. With modern compilers, it becomes simple to get rid of the short routines for interrupt handlers that you used to write in assembly language. All you need to do is use the `interrupt` keyword when declaring a function. The compiler will then generate a special function prologue and epilogue. The function prologue will save any registers from the scratch set that will be used, along with the preserve set registers; the epilogue will restore the contents of all registers (both preserve set and scratch set) and generate an appropriate "return from exception" instruction. The actual behavior of the prologue is somewhat compiler-dependent. On the Motorola 68000, which has a "save multiple registers under

mask" instruction, only those registers that are actually used are likely to be saved; on the Intel 80x86 family, it is probable that all the scratch registers will be saved.

What does such a C interrupt function look like? Here's an example of a keyboard interrupt handler:

```
void interrupt far kbdintsvc()
{
    register int c;
    register DEVICE d;

    c = inport(0);
    ainbuf[ainptr++] = c;
    if (ainptr>=IBUFSIZ)
        ainptr = 0;
    d = deviceTable[D_CONIN];
    if (d->interruptQueue)
        SignalTask(&d->interruptQueue);
    else
        ++(d->status.dcbsema);
}
```

Don't worry too much about the work done by the handler; the form of the C interrupt service routine is what you need to understand at this point. Note that all interrupt functions must be of type `void` and have no arguments. Since the address of this function will be placed in the processor's interrupt vector table, no mechanism exists for passing any arguments to the function. In other words, this function is never actually called. Since it has no caller, there is nobody to return a value to, so `void` is an appropriate type. Also, this should be a `far` function for the 80x86 family. Leave the `far` keyword out for processors that do not support it (e.g., the 68000).

It should also be noted that in the example, the `kbdintsvc()` function makes two function calls: `inport()` and `SignalTask()`. These types of function calls indicate an important concept: interrupt functions may call "normal" (non-interrupt) functions with no interesting side effects. Of course, any assembly language functions must adhere to the C calling conventions for preserving registers. Since a copy of the scratch registers has been preserved on the stack, it isn't necessary to keep saving them. In fact, you may not call another interrupt function, since interrupt functions are not designed to be called. Thus, it is illegal to explicitly call any function declared with the `interrupt` keyword.

### ANSI C const Keyword

As long as we're discussing declaration modifiers like `interrupt`, I should say a few things about the `const` keyword for ANSI C compilers. The basic concept is easy to

understand: do not allow modification of any data object declared as `const`. The syntax can be a little tricky, however. Just remember that C is a left-associative language in most cases. That is, "`int errno`" says that `errno` is of type `int`. Each token refers to the token to its left. (Unfortunately, the `far` and `near` keywords don't follow this rule, since they were modeled after Microsoft's keyword usage.)

The `const` keyword also modifies the token to its left. Thus, "`char const * pcc`" indicates that `pcc` is a pointer to a constant `char`. The pointer itself is not constant, but the character at which it points is. Alternatively, we could have a declaration like "`char * const cpc`", which states that `cpc` is a constant pointer to a `char`. The pointer itself is not allowed to vary, but the character that it points to can be modified. A few examples of how `const` is used may be helpful.

```
static char * const days[] = {"Sun", "Mon" "Tue" "Wed" "Thu" "Fri"
"Sat"};
```

In this example, we declare `days` to be an array of constant pointers to `char` with `static` scope. There really is no reason for these pointers to be anything else but constant, since the order of days in a week is unlikely to be modified at runtime. The benefit from declaring the pointers to be `const` is that they will be placed in a separate section by the compiler, and that section may then be located in memory as desired. Since the pointers will never change, they can be placed in ROM, for example. The characters themselves are allocated to the strings program section by the compiler and may also be placed in ROM. Thus, we have the advantage of being able to use pointer types (which are quite efficient) without wasting any RAM space for them. We also won't need to copy them from ROM to RAM at program startup, which is an advantage if your system is ROMable.

I'd like to give one more example of how `const` can be used in an embedded system. Suppose you have an input routine that obtains values in decimal or hexadecimal from the system operator. You will need to convert these values from their ASCII representation to a form your program can use. The following code converts characters representing numbers in any base up to 16 into binary form:

```
while (c = *s++)
{
    for (index=0; index<base; index++)
        if (c == hextable[index])
            break;
    value = value * base + index;
}
```

The table that is used to look up the individual characters may as well be a `const`, since the table can be placed in ROM with other data that doesn't change at runtime. We can achieve this by declaring it as:

```
static char const hextable[] = "0123456789abcdef";
```

Note that this is an array of constant characters, where the previous example used constant pointers.

### ANSI C volatile Keyword

ANSI C programmers tend to think of `const` and `volatile` together, since there are similarities in how they modify storage. Syntactically, they are identical; semantically, they are nearly opposites. Whereas a `const` data object can never change, a `volatile` one is assumed to have changed since its last usage. This means that declaring something `volatile` will have an effect upon optimization. The keyword tells the compiler to refrain from optimizing accesses to a given variable, since external events can alter the variable's contents.

In the embedded systems world, this typically applies to memory-mapped I/O devices. It may also apply to data objects in regions of memory that are subject to direct memory access (DMA) transfers, or are used for data shared by multiple tasks in a multitasking environment or between mainline code and ISRs. Volatile variables never end up as register variables.

The `volatile` keyword is typically applied to pointer variables. For example, a memory-mapped I/O device may be declared as:

```
char volatile * const receivedData = 0xfeed;
```

This means that `receivedData` is a constant pointer to a volatile character. We will now use the pointer in a loop:

```
while (*receivedData != 3) /* wait for CTRL-C */

   ;
```

The code generated for this loop will perform a fetch through the pointer on each pass through the loop. If we do not declare `receivedData` as a pointer to a `volatile char`, then the optimizer is free to fetch through the pointer once and repeatedly compare the result to the constant 3. This is meaningless but perfectly legal optimization. The addition of the `volatile` keyword to the C language provides a means to communicate your intentions to the compiler. If a variable is not safe to optimize because it is really a memory-mapped I/O port, you can convey this information in a standard, portable way.

### Using const for Data Protection

A variable with the type modifier `const` cannot be modified for the scope of its declaration, and, if the variable is static or global, will likely be placed in ROM. The ANSI C keyword `const` can also serve another function: data protection. Suppose that your program includes a data structure initialized and modified by one module—the "implementer" module—that must be accessible as read-only to some other "consumer" modules, and for the purposes of this discussion, a procedural interface is considered inefficient. You can declare the data structure as `const` in the consumer modules, while retaining the ability to modify it (non-`const`) in the implementer module. As long as the definition of the data structure is in the implementer module (non-`const` declaration), the data will not end up in the `const` section, yet the consumer modules will not be permitted to modify the data.

- Module a—the implementer:

```
int sacred_data = 0;
void implementer(int x)
{
   sacred_data = x + 2;
}
```

- Module b—the consumer:

```
extern const int sacred_data;
char consumer()
{
   display(sacred_data);
}
```

## 4.4 Optimization for RISC Architectures

*This article is based upon a piece by Donald Grimes and Fu-Hwa Wang in* NewBits *in 1990. Both authors were members of the Microtec compiler development team at that time. At the beginning of the 1990s, CISC (Complex Instruction Set Computer) processors, such as the 68 K family were prevalent. But RISC (Reduced Instruction Set Computer) devices were beginning to appear: Motorola had the 88 K (the successor to 68K), Intel had the i960, AMD had the 29K, and Sun had SPARC. This trend persisted, and the common high-end devices today (e.g., ARM and PowerPC) are RISC or RISC-oriented. Hence, the ideas in this article are just as applicable now. The article also serves to illustrate the old joke: "RISC" stands for "Relegate the Important Stuff to the Compiler." (CW)*

### Review of RISC Architectures

In the early days of computers, most programs were written in assembly language. As the art of hardware design advanced, designers implemented increasingly complex instructions and addressing modes for use by assembly language programmers eager to extract the maximum performance from their hardware. The typical modern CISC is actually a computer within a computer, the former executing microcode to implement the instructions of the latter.

Today, however, most programs are written in high-level languages; the complex instruction set is no longer visible to the programmer. Examining the code generated by compilers reveals that the vast majority of the instructions are rather simple: loads, stores, additions, and so on. Compilers make little or no use of the complex instructions and addressing modes that were provided for assembly language programmers.

The development of RISC architectures was inspired by the realization that providing complex machine instructions, which constitute a small fraction of the typical mix of instructions executed, slows the execution of all instructions, including the much more common simple instructions, since all must be interpreted by the microcode. The goal, therefore, is to provide only those "common simple instructions" that can be executed directly, rather than interpreted by microcode, and run quickly—ideally, each instruction should execute in a single clock cycle. The resulting gain in speed outweighs the increase in instruction count that results from synthesizing complex operations from the minimal instruction set. Although RISC designs have proliferated in recent years, the ideas are not new; in many respects, the Control Data 6600, designed by Seymour Cray in the early 1960s, was a RISC.

### Characteristics

RISC designs differ, but they tend to have certain features in common since they are driven by the ambitious goal of executing one instruction per cycle. One way to do more operations per unit of time is to make the individual operations faster; the simplified instruction set and addressing modes accomplish this task. Another way is to do more operations in parallel. A typical design has multiple execution units, capable of operating simultaneously and driven by an instruction dispatcher that attempts to launch one instruction per cycle. An instruction prefetch scheme or a cache may be included to keep the dispatcher supplied.

More impediments to executing one instruction per cycle and typical solutions follow:

- Some instructions inherently require more than one cycle.

  Multiplication, division, and floating-point operations all require more than one cycle. A common solution is pipelining: the execution of the instruction is subdivided into stages that can be executed in a single cycle. The execution unit is designed so that an instruction being executed progresses from one stage to the next at every cycle, and a new instruction can be initiated at every cycle. The result of an instruction appears in the result register some cycles after the instruction was initiated; depending upon the architecture, enforcing the necessary interlock (i.e., not using the result until it is ready) may be done by hardware or software. In either case, the compiler's challenge is to exploit this parallelism effectively.

- Access to memory is relatively slow.

  To address this problem, RISC processors have a large complement of high-speed registers and a load/store architecture—that is, only load and store instructions access memory; all other instructions operate on values in registers and return results to registers. Frequently, references to memory are handled by a separate execution unit: a store is launched in one cycle and then proceeds independently; a load is launched in one cycle, and the value appears in a register some number of cycles later. While the load is being processed, other instructions can be executed. The execution unit may also be pipelined so that a number of memory references can be in progress simultaneously. The compiler's challenge is first to use the large register set effectively so that references to memory are minimized and, second, when reference to memory is unavoidable, to exploit parallelism.

- Branches interrupt the instruction stream.

  Conditional branches, in particular, complicate the task of keeping the instruction dispatcher supplied: which instruction is next? Some architectures use a branch prediction bit in the branch instruction—the processor proceeds on the assumption that the prediction is correct, just as for an unconditional branch; delay is incurred only when the prediction is wrong. In other architectures, the instruction following the branch instruction can be executed in parallel with the branch, either unconditionally or depending on a bit in the branch instruction; this is called *delayed* or *deferred branching*.

  The Motorola MC88000 and Intel *i*80960 are examples of different RISC architectures. The MC88000 is a fairly pure RISC design: no microcode, most instructions execute in a single cycle, load/store architecture, 32 general-purpose registers, four independent pipelined execution units, and delayed branches. The *i*80960 uses a number of RISC concepts: many instructions execute in a single cycle, load/store architecture, 16 global and 16 local registers, four independent pipeline execution units, and branch prediction bits. The *i*80960, unlike a pure RISC, also provides complex instructions and addressing modes implemented in microcode. The

*i*80960CA is a superscalar machine. At each cycle, its instruction dispatcher examines the next three or four instructions and dispatches as many as three in a single cycle. The maximum sustainable execution rate is two instructions per cycle. This rate can be achieved only if the instructions appear in the correct order, the execution units are available, and no register conflicts occur.

### Optimization Goals

To a certain extent, optimizing for RISC has the same objective as optimizing for any other architecture:

- Eliminate redundant computation by such techniques as common subexpression elimination, constant propagation, and copy propagation.
- Make the best use of the processor's complement of fast registers by various techniques such as register coloring (i.e., shared register allocation based on an analysis of variables' lifetimes).

Realizing the one-instruction-per-cycle potential of RISC architectures requires special attention to these issues:

- Avoiding loads and stores, particularly within loops.
- Keeping all the execution units busy and the pipelines full.

The next section describes techniques for achieving these goals.

### Instruction-Scheduling Optimizations

Instruction scheduling refers to the process of rearranging a given instruction stream to make better use of the hardware's capabilities. This section discusses important examples of this process.

#### *Advance Long Operations*

If a multicycle instruction (a load, an integer multiply or divide, or a floating-point operation, for example) is immediately followed by an instruction that uses its result, the processor must delay while the first instruction executes. If, on the other hand, the multicycle instruction can be moved to an earlier position in the instruction stream, it may be possible, in the absence of conflicts over execution units and registers, to do other useful work in parallel with it.

#### *Compaction*

Compaction is the process of moving useful instructions into execution slots that would otherwise be idle. Compaction is important in efficiently utilizing the CPU's capability of executing one instruction per cycle. An execution slot may be idle because a multicycle instruction has been moved to an earlier position in the instruction stream or, on an architecture like the MC88000 or SPARC, it may be the instruction following a branch. If no useful instruction can be found to fill the slot, the CPU delays while the branch is executed.

### Avoiding Branch Delays

For several reasons, branch instructions, particularly conditional branches, make it difficult to achieve the one-instruction-per-cycle execution rate. Typically, a conditional branch tests the condition code, so it potentially must wait until a previous instruction has finished setting it.

The process of resetting the program counter and reloading the instruction cache may also introduce a delay. Different architectures use different strategies to deal with this delay. The *i*80960 chooses one execution path, based on a branch prediction bit in the instruction. It incurs a delay only when it guesses wrong. On the MC88000, branches that are taken always require two cycles, but the architecture offers the opportunity to execute the instruction following the branch while the branch is taking place.

Instruction scheduling can attack branch delays on several fronts. On the *i*80960, it can attempt to set branch prediction bits correctly, either statistically, based on the assumption that looping branches are usually taken, or dynamically, based on actual execution statistics. It may be able to move the instruction that sets the condition code well ahead of the branch instruction that tests it, so that the branch need not wait. Otherwise, on the *i*80960, it can combine the compare and branch into a single special-purpose instruction called `COBR` (compare and branch), thereby saving space in the instruction cache.

### Order for Concurrency

In architectures with multiple independent execution units, the compiler's goal is to keep them all busy. Techniques for achieving this goal are necessarily dependent upon the architecture. The *i*80960CA can sustain its maximum execution rate of two instructions per cycle only when the first instruction is executed by the execution unit (REG side) and the second by the address-generation unit (MEM side). For this reason, instructions are rearranged to appear in this order whenever possible. Sometimes an arithmetic calculation can be moved from the REG side to the MEM side by expressing it as a form of load-address instruction, thereby enabling great concurrency. This optimization is called *instruction migration*.

The MC88000 is capable of issuing one instruction per cycle, so maintaining an effective execution rate of one per cycle is a matter of keeping the pipelines full. To achieve this execution rate, one needs to move multicycle instructions well ahead of instructions that use their results and avoid unnecessary register dependencies.

### Register Cycling

The usual register assignment technique, register coloring, is designed to use the minimum number of registers. It tends to reuse a register as soon as it becomes available. Instruction scheduling, however, tries to advance multicycle instructions in the instruction stream. This strategy introduces more opportunities for parallelism if the register containing the result is not reused immediately. Register cycling attempts to use all the scratch registers, avoiding reuse until absolutely necessary.

## Loop Optimizations

Some classical loop optimizations are particularly important for RISC architectures. These loop optimizations are described in the sections that follow.

### *Induction Expression Elimination*

Array index computations generally involve multiplication, usually a slow operation on a RISC processor. Induction expression elimination replaces the index computation inside the loop with a simple addition.

For example, the code fragment:

```
int a[BOUND][BOUND];
int i, j;
for (i=0; i<BOUND; i++)
    for (j=0; j<BOUND; j++)
        a[i][j] = 1;
```

is compiled roughly as if it had been written:

```
int a[BOUND][BOUND];
int i, j;
register int *p1;
for (i=0, p1=a; i<BOUND; i++, p1+=BOUND)
{
    register int *p2;
    for (j=0, p2=p1; j<BOUND; j++, p2++)
        *p2 = 1;
}
```

In the former case, the following expression:

```
address(a) + BOUND * i + j
```

that is, the address of `a[i][j]`, is recomputed on each iteration of the loop; in the latter, it is computed once at the beginning and then maintained by simply incrementing a pointer on each iteration.

### *Register Caching*

Loads and stores are among the more time-consuming operations on a RISC processor. Register caching assigns a variable to a fast register for the duration of the loop. The value is loaded from memory at the beginning of the loop and stored back at the end. References to the variable within the loop are then fast register accesses instead of slow loads and stores. For example, a loop such as:

```
sum = 0;
for (i=0; i<n; i++)
   sum += a[i];
```

is compiled as if it were:

```
register int temp;
temp = 0;
for (i=0; i<n; i++)
   temp += a[i];
sum = temp;
```

thereby eliminating (n-1) loads and stores of `sum`.

### *Software Pipelining*

In some cases, rearranging the instructions within a loop and moving part of the first iteration out of the loop enables more concurrency in the body of the loop and allows the entire loop to run faster. This technique is called *software pipelining*.

Consider, for example, the simple loop:

```
for (i=0; i<BOUND; i++)
   z[i] = x[i] * y[i];
```

Naïve code generation will produce (in pseudo-assembler):

```
   i := 0
loop:
   load x[i]
   load y[i]
   multiply
   store z[i]
   if (++i<BOUND) goto loop
```

Execution of this loop is essentially serial. Specifically, the multiply must wait for the load to complete, and the store must wait for the multiply. Worse yet, load, multiply, and store are all multicycle instructions, so the processor spends a number of cycles waiting. The processor is therefore unable to maintain an instruction issue rate of one per cycle. A slight rearrangement offers several benefits:

```
    i := 0
    load x[0]
    load y[0]
loop:
    multiply
    load x[i+1]
    load y[i+1]
    store z[i]
    if (++i<BOUND) goto loop
```

Now, on every iteration but the first, the multiply has a shorter wait (or no wait at all) for the results of the load instructions. Furthermore, opportunities for parallel execution have greatly increased: the loads inside the loop can execute in parallel with the multiply; the store can execute in parallel with the loads; and the multiply in the next iteration can execute in parallel with the store from the previous iteration.

### Effects of Optimization

This section presents some results of some benchmarks on the *i*80960CA that show the effectiveness of these optimization techniques.

#### *Puzzle*

Puzzle is a compute-bound program that solves a combinatorial puzzle by backtracking. Loop optimizations streamline the inner loop as follows:

```
int fit(int i, int j)
{
    register int k;
    for (k=0; k<pieceMax[i]; k++)
        if (p[i][k])
            if (puzzle[j+k])
                return (false);
    return (true);
}
```

Since `i` doesn't vary within the loop, the value of `pieceMax[i]` is fetched once at the beginning of the loop and cached in a register. Induction expression elimination reduces the address calculations for `p[i][k]` and `puzzle[j+k]` to incrementing and dereferencing of pointers also cached in registers. The result is a 42% speed improvement over the unoptimized version.

### *Bitblt*

Bitblt is a memory-bound benchmark. It simulates a graphic display application by copying blocks of bit patterns around in memory. Largely because of the instruction scheduling of its inner loop, the optimized version runs 23% faster.

### *Bezier*

Bezier is a recursive, curve-splitting algorithm for cubic Bezier curves. The program contains no loops; nevertheless, register allocation and instruction scheduling yield an improvement of 72% over the unoptimized version.

## Conclusion

RISC architectures require sophisticated compiler technology to realize their performance potential. In a sense, RISC designs move complexity from hardware into software. Effective optimization is crucial to achieve the promise of RISC designs.

## 4.5 Bit by Bit

*This piece is based upon one that I wrote for* NewBits *in 1992, when I was with Microtec UK. Even today, C programmers find bit fields challenging. (CW)*

In this article I plan to get down to basics and take a look at the issues and challenges associated with memory in embedded systems. At the lowest level, memory is just lots of bits: numerous logic states represented by 1s and 0s. The grouping of these bits into bytes and words is really quite arbitrary; sometimes it becomes necessary to deal with memory bit by bit or in small groups if bits. I will look at how this is accomplished from the C programmer's perspective.

### Bitwise Operators

C is generally considered a good language for embedded systems programming for a number of reasons, one of which is its ability to perform bitwise arithmetic. All the primary bit operations are available: inversion (using ~) to change 1s to 0s and vice versa; AND (using `&`) to test bits; OR (using |) to set bits; exclusive OR (using ^) to all toggle specific bits; shift right and left (using >> and <<) by one or more bits.

Using these operators, the programmer can perform any required bit manipulation. However, the notation does not yield very easily readable code. This situation is exacerbated by the need for the expression of bit patterns in hexadecimal (or octal). C does not have a means of including binary constants. For example, it is not intuitive that the following code:

```
bytevar |= 0xc0;
```

results in the setting of the top two bits of the 8-bit variable `bytevar`.

### Binary Constants

There is a way to provide what appear to be binary constants in standard C. All that is necessary is a series of macro definitions in a header file (`bits.h` perhaps), like this:

```
#define b00000000 ((unsigned char) 0x00)
#define b00000001 ((unsigned char) 0x01)
#define b00000010 ((unsigned char) 0x02)
...
#define b11111110 ((unsigned char) 0xfe)
#define b11111111 ((unsigned char) 0xff)
```

### Bit Fields in Structures

Since working with single bits or small groups of bits is useful, the C language was designed to include integers of any bit size in structure definitions. In ANSI C, these

integers may be signed or unsigned and may be as large as the widest available base type (typically 32 bits).

The mechanism for allocating memory to bit fields is governed by the rules and definitions of the C language and the design of the specific compiler. An understanding of the latter is important, since, without care, it is easy to write compiler-specific code. As Kernighan and Ritchie wrote in the second edition of *The C Programming Language*, "Almost everything about (bit) fields is implementation-dependent."

Bit fields may be allocated from the top of the first word of memory or, less commonly, from the bottom. Unnamed fields may be used for padding. Bit fields are allocated through the first word until one of a number of criteria cause the next word to be utilized:

- Insufficient bits remain in the current word, and word boundary straddling is not supported by the chip/compiler.
- An unnamed field with width 0 is encountered.
- A new base type is encountered.

The last criterion has interesting effects. Compare the following two structures, `alpha` and `beta`:

```
struct
{
   char a : 2;
   char b : 3;
} alpha;

struct
{
   char a : 2;
   int b : 3;
} beta;
```

These two apparently similar structures use different amounts of memory. The structure `alpha` requires a single byte of memory, whereas `beta` requires eight bytes.

Remember that the C bit fields are just a convenient notation for the programmer. The compiler must still generate the instructions to perform the necessary ANDs and ORs. However, beyond their notational convenience, C bit fields also give the well-designed compiler an opportunity to generate particularly efficient code.

### Microprocessor Bit-Field Instructions

Recognizing the importance of the C language and the usefulness of the bit-field notation, a number of microprocessors include instructions to operate on bit fields of

arbitrary width. Typical chips with this capability are the Motorola 68020/30/40 and the Intel x86 family in native 32-bit mode. Here is some compiler output that shows 68020 bit fields in use:

```
;struct
;{
; unsigned a : 3;
; unsigned b : 4;
;} blob;
;
;main()
;{
    XDEF _main
_main:
; blob.b = 1;
    moveq #1,d0
    bfins d0,_blob{3:4}
;}
    rts
```

In the same way that the use of bit fields in C gives the compiler a chance to produce optimal instructions, the use of a microprocessor's bit-field instructions enables the chip to perform well. Although the same memory read/write cycles are required, a significant speed increase is possible.

### I/O Devices and Bit Fields

A commonly cited example of the application of bit fields is for accessing input/output device control registers. It is very common for I/O devices to have control registers that are segmented into a number of fields of various bit lengths. An obvious concept is to map a C structure onto such an I/O device control register. Each of the individual fields can then be read and/or written as required. There are potential problems with this technique, if the ports are write-only; I have addressed this issue from the C++ standpoint in another article—"Write-Only Ports in C++" in Chapter 5.

It is generally difficult to find examples of code used typically in an embedded system that you, the reader, can experiment with. This is simply because such examples make assumptions about the available target hardware, assumptions that may not be fulfilled. However, in one suitable example, the target is a standard PC and the I/O device the program addresses is the screen. The screen of a PC, in text mode, appears to the program to be an area of memory, where each word is divided into two (eight-bit) bytes, the first of which contains the ASCII code of the character to be displayed; the second is divided into a number of bit fields. Specifically, the latter is divided into a

one-bit, a three-bit, and a four-bit field, which specify flashing, background color, and foreground color, respectively. Here is the definition of a structure that reflects this format:

```
struct cell
{
    unsigned sym : 8;
    unsigned fg : 4;
    unsigned bg : 3;
    unsigned flash : 1;
};
```

In the following code, the variable `p` is a pointer to type `struct cell` and is initialized to point to the screen memory. The code of `main()` simply writes a pattern on the screen using the bit-field references.

```
struct cell *p = (struct cell *) 0xb8000000;

void main()
{
    int row, col=0, inx;

    for (row=0; row<25; row++, col+=3)
    {
        inx = row*80 + col;
        p[inx].sym = 1;
        p[inx].fg = 1;
        p[inx].bg = 0;
        p[inx].flash = 0;
    }
}
```

This code does, of course, make certain assumptions about the compiler—that is, the allocation of fields across a word is compiler-dependent.

### Conclusions

For embedded systems, the manipulation of bits of memory is often necessary, and the C language bit-field notation is useful for this application. If you are using a high-performance target chip that supports bit-field manipulation directly, you need a compiler that can take full advantage of this facility. Since the use of bit fields, in general, introduces compiler dependencies, the documentation should carefully identify these dependencies.

## 4.6 Programming Floating-Point Applications

*This article uses a test case from a piece in* NewBits *in 1992 by Sarah Joseph-Bigazzi, who was a software engineer with Microtec. The original article was very product-centric, but the example code has much wider application. (CW)*

Floating-point values are used widely in mathematical and number-crunching applications. Capable of representing extraordinarily large and small numbers, floating-point numbers offer a numerical range that is essential to many embedded systems designs. Yet, despite the importance of these numbers, few developers are fully aware of the intricacies and pitfalls in their use. In this brief article, we will look at an example of some floating-point code that looks fine on the surface, but contains a subtle, but important, error.

### A Test Case

Here we have a program using floating-point operations:

```
void compare(double x, double y, int i)
{
    if (x == y)
        printf("Compare #%d worked\n", i);
    else
        printf("Compare #%d failed\n", i);
}

main()
{
    const double PI=ASM(double, "fldpi");
    double d_array[5], d_value=PI;
    float f_array[5], f_value=PI;
    register int i;

    for (i=0; i<5; i++)
    {
        d_array[i] = d_value / (i+1);
        f_array[i] = f_value / (i+1);
        compare(d_array[i], f_array[i], i+1);
    }
}
```

This program simply fills `d_array` (a `double` array) and `f_array` (a `float` array) with the floating-point values π, π/2, π/3, π/4, and π/5 (the `ASM` instruction `fldpi` extracts the value π from the 8087 coprocessor—we are assuming an x86 processor, but the same principle can be applied for other devices). Each element of `d_array` and each element of `f_array` will be passed to `compare()` as double-precision

parameters. The parameters x and y are then compared. If the values are equal, the program will display a message in the form:

```
Compare #n worked
```

for values of n from 1 to 5.

If the values are not equal, the program will display:

```
Compare #n failed
```

### Running the Test Case

Since both arrays are loaded with equal values, you should expect all the comparisons to succeed and the following output to appear:

```
Compare #1 worked
Compare #2 worked
Compare #3 worked
Compare #4 worked
Compare #5 worked
```

But that is not what happens. (Try it!) What you actually see is:

```
Compare #1 failed
Compare #2 failed
Compare #3 failed
Compare #4 failed
Compare #5 failed
```

So what is going on here?

### Troubleshooting

If you have a debugger available, step through the code and have a look at the two parameters, x and y, of compare(). You will observe that they are, indeed, slightly different. To find out why they are not equal, we need to go back to the code in main() where the values were derived.

Upon examination of the declaration of arrays d_array and f_array, we find the source of the problem: the first is declared double and the second is float. Thus, the values π, π/2, π/3, π/4, and π/5 were calculated in d_array and f_array using different precisions. Since the values in f_array were calculated using a single-precision floating-point format (float), f_array remained less precise than d_array even after the values in f_array were promoted to double-precision prior to comparison. Because of this difference in precision, in every call to compare(), x and y were

inherently unequal. Using the == operator reflects a classic programming error, namely, an attempt to test for equality between two values of different precision.

The comparison `(x == y)` should have been expressed alternatively as: `(fabs(x-y) < delta)` where `fabs()` is a C library function that returns the absolute value of its real argument, and `delta` denotes the maximum difference acceptable for the two values to be considered equal. In this case, a value of `delta` of 0.0005 is appropriate.

### Lessons Learned

Although this was a purely fabricated example, two lessons may be learned and applied to real embedded floating-point applications.

First, it is very easy to "play fast and loose" with data types in C—in other words, allowing automatic conversions to occur and not worrying about the consequences. This practice can cause enough difficulties with integral data types (sign extension, etc.), but with floating point, the bugs can be very subtle and hard to locate.

The second lesson is to dismiss all thoughts of "equality" between any two floating-point values. There is always the possibility of very slight rounding errors, even if the data type issues are taken care of. It is normally not even safe to look for a precise zero value—even though there is an exact representation for zero. You should never code `(x == 0.0)`; it should always be something like: `(fabs(x) < TINY)`, where `TINY` is defined to be a value small enough to represent a rounding error in the floating-point format for your processor.

## 4.7 Looking at C—A Different Perspective

*This article was written as a result of my experience training engineers in C programming for embedded applications. It appeared in a more expanded form in* NewBits *in 1993, including some topics that are covered in articles elsewhere in this book. I have also added some new examples. (CW)*

Even back in the mid-80s, when many other languages were significant for microprocessor software development (assembler, Pascal, even FORTRAN), C was emerging as the clear winner and the choice for the 1990s.

Nowadays, it's no contest. Everyone uses C (or its derivative C++) for embedded systems work. It's easy to forget that C was not designed for this purpose. Of course, when Kernighan and Ritchie first defined the language, microprocessors had not been invented, and electronic control systems were either hard-wired or controlled by minicomputers.

The experience of training is always a two-way flow of information; the teacher generally has something to learn from the pupil. In C language training, that something is a different perspective on aspects of the language that we all take for granted.

### Static Things

The keyword `static` is often a source of confusion for those learning C for the first time. The reasons for the confusion are that the keyword really performs two separate functions (three in C++), and some variables may be stored statically but are not declared as `static`.

The word *static* means "unmoving or stationary." Static variables are ones that are allocated space at compile time rather than being dynamically allocated machine registers or stack space at runtime. All variables outside of C functions are stored statically. Inside a function, local variables may be declared as static (using the `static` keyword as a qualifier on the declaration of the variable), if it is required to preserve a value from one call to the next. This capability is useful, but results in the function not being reentrant.

Variables outside of a function need not be declared static, but if they are, they are rendered local to the module (file) in which they appear, instead of being global, as is otherwise the case. This is a useful facility because it provides a means of communication between a group of related functions. However, it is really a different use of the `static` keyword. The same idea can be applied to functions. Functions are normally global. By declaring one as `static`, it is made available only in the module in which it is defined.

So the `static` keyword has two distinct meanings: it can specify memory allocation, and it affects the scope of a variable or function. I have seen the suggestion that you could `#define` a new "keyword" `local`, which would just be `static` in disguise.

### All Those Semicolons

When someone first learns C programming, he or she usually has problems with using the semicolon. After all, many of the semicolons do seem unnecessary. As a result, lots of compiler diagnostics about missing semicolons are generated. Ironically, the effect of

missing a semicolon can cause errors several lines down in the program, which is confusing for the beginner.

The usual result is a big swing the other way. The C novice starts putting in more semicolons left, right, and center. This solves the problem of the missing ones but introduces some more subtle difficulties at runtime. The following loop, for example, gets executed only once (for a value of `i` of 5):

```
for (i=0; i<5; i++); printf("%d", i);
```

instead of the five iterations that the programmer intended.

This string-character-counting loop never finishes (or never actually gets started):

```
while (*str); str++, len++;
```

Under some circumstances, you want a loop to be empty. This code, for example, is waiting for a device to return a nonzero value:

```
while (*devptr == 0)
   ;  /* wait */
```

It is a good practice to put the lone semicolon (the empty statement) on a line by itself and include a comment, as shown in the example. And let's hope that `devptr` is declared as a pointer to a `volatile int`, or we will be in real trouble.

### Pointers and Pointer Arithmetic

For many, pointers are thought of as strange and mysterious entities and operations on them akin to a black art. This topic is far-ranging enough to deserve its own article, "Pointers and Arrays in C and C++," later in this chapter.

### When Being Clever Is Not Being Smart

A certain kind of programmer likes to demonstrate how clever they are at every opportunity. I guess that we have all met similar people in many walks of life. Such programmers specialize in arcane code that *might just* be a tiny bit more efficient. A little thought about what "efficient" might mean can often give some useful insight.

Here is an example of some "clever" code:

```
void bin(int x)
{
   if ((x/2) != 0)
      bin(x/2);
   printf("%d", x%2);
}
```

What does it do?

To save you having to spend ages trying to analyze the code, I will tell you: it prints a number in binary format.

What is wrong with the code? Two things. First, it is hard to understand, and since a large proportion of programming time is spent on maintenance, such lack of clarity is short-sighted. Second, the code is recursive, and while it is valid in C for an embedded application, recursion should be used with great caution and only when absolutely necessary. It is easy for the code to consume more resources (i.e., stack space) than may have been anticipated.

Consider this alternative code (which does almost exactly the same thing):

```
void bin(int x)
{
    unsigned mask=0x80000000;
    int i;

    for (i=0; i<32; i++, mask >>= 1)
        if ((x & mask) != 0)
            printf("1");
        else
            printf("0");
}
```

This implementation may seem simplistic, but it is certainly much easier to understand.

So, did the "clever" code have any virtues or advantages? It may have been faster, but, for a user I/O function like this, speed is not an issue. What about size? We compiled both pieces of code for a 68 K target using "optimize for size." I expected the results to be similar and to argue that a few extra bytes are a small price to pay for extra clarity. I did not expect the result that I got: both produce exactly the same amount of code—60 bytes.

### Conclusions

Although it is the most used language for embedded systems work and is well suited for this purpose, care is still required when writing in C. Another precaution to take is the selection of software development tools that are specifically designed for embedded systems development.

## 4.8 Reducing Function Call Overhead

*Embedded engineers are always concerned about resources (usually time and memory) and want to be sure that they are working efficiently. This quest for efficiency can often be at the expense of good programming practice. In this article, based upon one that I wrote for* NewBits *in 1994, I look at how modern compilers address some of the efficiency issues without conflict. (CW)*

It sometimes seems that the life of a programmer is cursed. There never seem to be any good breaks. If code is small, it probably runs slowly. Fast code is generally too big. Worst of all, code that is easy to read (i.e., structured code) is probably large and slow. Why is this the worst case? Because, although it is commonly believed that programming is all about making a microprocessor do something, it is really a task performed to communicate your ideas to another programmer (or yourself at a later date). Since at least 90% of programming time is spent on maintenance, it is financial common sense to write clear, easy-to-read code.

### Compilers and Structured Code

Recognizing these needs, modern compilers are designed to process structured code well. Broadly, this goal involves translating the intentions of the programmer into machine code rather than simply making the executable map directly onto the original source code. The following loop provides a simple example:

```
char v[4];
for (i=0; i<4; i++)
  v[i] = 0;
```

which an efficient compiler may compile into a single 32-bit move of `0` into memory.

This presents a minor debugging challenge but not really a difficulty for a debugger designed to cope with fully optimized code. The improvement in both code size and speed is well worth it.

The biggest challenge in efficiently processing structured code is function calls. An easy-to-read program consists of a modular collection of smaller functions that call one another as required. This approach to program design achieves the goal of more maintainable and flexible code. However, the cost in runtime performance may be severe, because every call/return sequence carries overhead.

### Inline Functions

Before considering what a function call overhead really means, it is worth considering how function calls may be avoided. One solution might be the C++ `inline` keyword. This language extension is supported by a number of modern C compilers. The `inline` keyword offers advice to the compiler that communicates that the indicated (small) function could be usefully inlined. For each call to the function, the compiler inserts the actual code of the function instead of the call. Instead of the `inline` keyword, some

compilers have command-line switches that the programmer can use to indicate which functions to inline.

Both of these techniques have a drawback: the programmer must determine which functions to inline. The best approach is for the compiler to be "smart" enough to make the decision itself, as long as the programmer has indicated that execution time is a priority.

### Function Calls

The process of executing a function call can be broken into four discrete stages:

1. Evaluating the parameters (if any) and preparing them for transfer into the called function.
2. Constructing the stack frame; space is likely to be required for local variables (which have not been allocated machine registers) and for internal workings.
3. Executing the function code.
4. Returning a value from the function.

Each of these stages (except stage 3) represents a real or potential overhead beyond the processing necessary to perform the job required of the function.

### Parameter Passing

In C, the way parameters are handled is clearly defined: they are evaluated from right to left and pushed onto the stack in turn. In principle, the parameters could be passed by other means (e.g., via machine registers), but studies tend to indicate that such other means do not yield the expected performance benefits.

Why are the parameters dealt with from right to left? (Were Kernighan and Ritchie left-handed?) This question is commonly asked in C language training courses. It seems a reasonable query, because a number of other languages deal with parameters the more "logical" way—left to right. The reason is simple: C permits a given function to accept a variable number of parameters. The only way such a function can determine just how many parameters have actually been passed is from information in a mandatory (typically the first) parameter. For this to work, that parameter must be at a predictable location, such as the top of the stack. An example of this technique is the standard `printf()` function. The initial (mandatory) parameter is a string `(char*)` containing format descriptors that indicate how many parameters follow. Of course, programmer error may result in the failure of this mechanism, but a disadvantage of such great flexibility is some lack of robustness.

Can the knowledge of parameter passing be exploited by the programmer? If the ability to handle a variable number of parameters is used, it is significant that the first parameter is on top of the stack. By taking its address, indexing off of it can provide access to all the others. But, of course, this practice will probably result in nonportable code.

### Local Storage

Once a function has been called, the parameter values are pushed onto the stack with the return address on top. The next concern is the provision of local work space for the function. This includes memory allocated to local (automatic) variables and storage space for intermediate values during expression evaluation.

The need for intermediate storage is a misconception that often results in less-readable code. For example, the complex statement:

```
v[i] = w[p + q->t];
```

may be used out of fear that using a variable to store an intermediate value would result in unwanted overhead. In fact, recoding the statement as a series of simple steps:

```
j = p + q->t;
v[i] = w[j];
```

is unlikely to have any effect at all upon the resulting machine code.

In a small function, both local variables and intermediate storage may be accommodated by machine registers, but this depends upon the target microprocessor architecture and the degree of compiler optimization.

### Stack Frame Generation

In most functions, local memory storage is a requirement, and the stack is the obvious place to accommodate it, which is logical since indexing off of the stack pointer is efficient for most 16/32 bit processors. Also, functions are inherently reentrant, since a new area can be allocated on each call, and the code may be shared between tasks (or between main line and interrupt code). This chunk of memory on the stack is known as the *stack frame.*

The generation of a stack frame is quite a simple process. The compiler determines how much local storage will be required, and the stack pointer is incremented, or more often decremented, by the appropriate amount. At the end of the function, the stack pointer's previous value is restored so that the return address is on top again.

Because high-end microprocessors are invariably programmed in a high-level language, support for stack frame generation is incorporated into the instruction sets of such devices. The instructions that perform the stack frame construction and destruction are generally called `LINK` and `UNLK` (i.e., unlink), respectively. The idea is quite simple. A suitable (address) register is nominated to be the *frame pointer* `(FP)`. For the 68000 family, the frame pointer is generally `A6` because `A7` is used as the stack pointer `(SP)`. The `FP` points to a linked list of stack frames to facilitate their easy deallocation by the `UNLK` instruction. The `LINK` instruction builds the frame in three simple steps:

1. The current FP is pushed onto the stack.
2. The FP is set to point to the pushed (old) value.
3. The stack pointer is advanced by an appropriate amount to allocate the required space.

The structure of the stack at the beginning of the "real" function code is shown in Figure 4.3.

The UNLK instruction undoes the frame structure in two stages:

1. The SP is set to the value of the FP.
2. The old FP is popped off of the stack.

Once the stack frame is established, the function parameters may be accessed at (known) positive displacements (for most processors) from the FP. Local variables and work space are at negative displacements.

In principle, the LINK/UNLK sequence has the ability to dynamically adjust the size of the local work space during the execution of the function while still being able to locate parameters easily. However, the C language does not really exploit this capability.

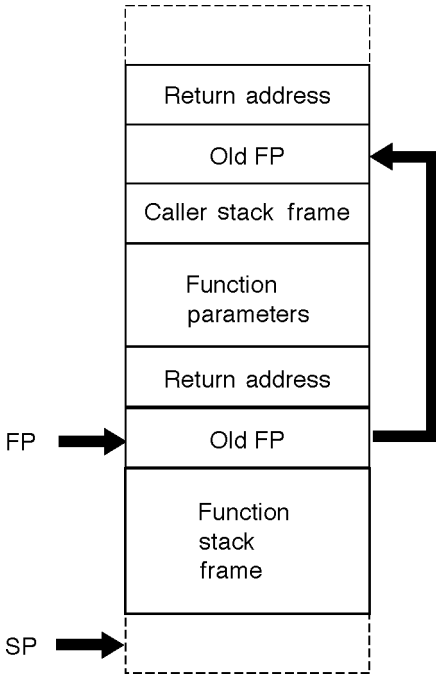



*Figure 4.3: Stack structure after a function call*

### Return Values

In C, any non-void function is assumed to return a value. For most functions, this is an integral type (int, char, short, etc.), a floating-point value, or a pointer (address). Almost without exception, a machine register (or pair of registers) of the appropriate type is employed. There is little scope for improved efficiency here.

### Conclusions

Programmers who are particularly concerned with the runtime performance of their programs can apply a knowledge of function call overheads to their program design and use modern compilers that combine efficient local optimization with automatic inlining capabilities.

## 4.9 Structure Layout—Become an Expert

*This article has a long and complex history. It is based upon a piece in the Spring 2004 issue of* NewBits *by Antonio Bigazzi, the leader of the compiler team at Accelerated Technology, with valued assistance from Meador Inge, which, in turn, was an updated version of an article he had done at Microtec, also for* NewBits, *back in 1993. It illustrates a fine example of something that seems quite simple and can be taken for granted but, in the embedded world, must be understood in more detail. (CW)*

Modern embedded C/C++ compilers give fine-grained control and a wealth of options for determining how C structures (or C++ classes) are laid out. The result is that any arbitrary layout can be attained. This article describes the simple and consistent rules that many compilers use in laying out structures, showing you how to tailor these rules to your needs. You will need to check whether the rules and the options that control them are implemented in a specific compiler, and we have endeavored to indicate where options may be unusual. These rules are described with reference to Freescale 68 K, ColdFire, and PowerPC, but the same ideas apply to all target architectures. We will discuss these rules for C, with the understanding that the same rules apply to C++.

The article will help you to:

- Determine the memory layout of any structure, whether packed or unpacked; determine the offset and alignment of members.
- Force any desired layout; for instance, force cpu32 (e.g., 68332—a bus16) and cpu32p (e.g., 68340—a bus32) structures to have identical layouts.
- Achieve maximum space or time efficiency for accessing the structures in your program.
- Learn a number of useful techniques and test yourself; a test is included at the end of this article.

### Key Concepts

To understand structure layout fully, you first must be familiar with the concepts of data bus width and natural boundaries. The compiler uses these concepts in the basic algorithms that compute offsets and total size when allocating structures.

#### *Data Bus Width*

The 68-K family includes many variants, which are divided into two groups

- Those that utilize `BUS16`: 68000/08/10, 68302, 68330/1/2/3, 68340, 68ec000, 68hc000/1, cpu32
- Those that utilize `BUS32`: 68020/30/40/60, 68ec020/30/40/60, cpu32p

All ColdFire devices are `BUS32` (the 5204, now obsolete, was a `BUS16`).

PowerPCs are either `BUS32` or `BUS64`. For example, PPC403GA is `BUS32`, while PPC740 is `BUS64`.

BUS16s have a 16-bit data bus width, meaning that a memory cycle can access a maximum of 16 bits. Multibyte quantities can be properly accessed only if they reside at an address that is a multiple of two. For instance, fetching a short integer (which takes up 2 bytes), located at an odd address, causes an address exception. Since it is not, in general, possible or practical to recover from such an event, the compiler must avoid this kind of situation by properly accessing misaligned data in a byte-wise, piecemeal fashion.

BUS32s have a 32-bit data bus width, meaning that a memory cycle can access a maximum of 32 bits. Multibyte quantities can be properly accessed at any address, but if they are not properly aligned, performance will be degraded because the hardware must supply extra memory cycles. In other words, the piecemeal access is provided by the hardware itself—at a cost. BUS64s have a 64-bit data bus width, meaning that a memory cycle can access up to 64 bits. Efficiency considerations are similar to BUS32. Eight-byte quantities (C doubles) should be aligned to a multiple of 8 for maximum efficiency.

### *Natural Boundaries*

Each C type has a natural boundary—that is, a number N such that a variable of that type is accessed best if aligned to an address multiple of N. As a general rule for scalar types, the natural boundary is determined by:

N = min (bus width, data size)

The following table specifies the sizes and natural boundaries for all C types:

| C Type | Size | BUS16 | BUS32/64 |
|---|---|---|---|
| char | 1 | 1 | |
| short | 2 | 2 | |
| int | 4 | 2 | 4 |
| long | 4 | 2 | 4 |
| pointer | 4 | 2 | 4 |
| enum | 4 | same as int | |
| packed enum | 1 | same as char | |
| packed enum | 2 | same as short | |
| packed enum | 4 | same as int | |
| integral type : bit(m) | - | same as integral type | |
| float | 4 | 2 | 4 |
| double | 8 | 2 | 4/8 |
| struct | - | same as most demanding member | |
| union | - | same as most demanding member | |
| array | - | same as component | |
| packed struct | - | 1 | |

### Unpacked Structures

Memory layout for unpacked structures is governed by Golden Rules 1 and 2:

- **Golden Rule 1:** The natural boundary for an unpacked structure is the natural boundary of its most demanding member. Padding bytes may be needed between members to guarantee proper aligning.
- **Golden Rule 2:** The size of an unpacked structure must be a multiple of its natural boundary. Trailing bytes may be needed at the end of the structure to satisfy this rule.

These two simple rules guarantee that, should a structure become a substructure or a component of an array, the natural alignment of its members is preserved.

The table below shows the notation used in the code examples that follow.

| Notation | Description |
|---|---|
| A...Z | Uppercase letters denote a nonbit structure member. The letter is repeated to show how many bytes the member occupies (one letter per byte). For example, `AAAA` represents a 4-byte structure member. |
| { } | Curly braces surround an unpacked structure. For example, {`AAAABBBB`} represents an unpacked structure containing 4-byte members A and B. |
| ( ) | Parentheses surround a packed structure. For example, `(BCCD)` represents a packed structure containing a char-shot-char. |
| ∅ | This character represents a padding byte or trailing byte. For example, {`A∅BBC∅`} shows a padding byte between `A` and `B` and a trailing byte after `C`. |
| a..z | Lowercase letters denote a bit field. The letter is repeated to show the number of bits the member occupies (one letter per bit). For example, `aaa` represents a 3-bit field. |
| \| | A bar emphasizes byte boundaries when bit fields are represented. For example, `aaaaaaab\|cccccccc` indicates 2 bytes. |
| Δ | This character represents a trailing bit. For example, {`abΔΔΔΔΔΔ`} means 6 trailing bits follow bit-field member b. |

Consider the following declarations:

```
struct simple
{
   char A;
   int B;
   char C;
};
struct simple vect[2];
struct outer
{
   char X;
   struct simple Y;
};
struct point
{
   char C;
   double X, Y;
};
```

Memory layout on BUS16:

```
struct simple {AøBBBBCø}
array vect [{AøBBBBCø}{AøBBBBCø}]
struct outer {Xø{AøBBBBCø}}
struct point {CøXXXXXXXXYYYYYYYY}
```

Memory layout on BUS32:

```
struct simple {AøøøBBBBCøøø}
array vect [{AøøøBBBBCøøø}{AøøøBBBBCøøø}]
struct outer {Xøøø{AøøøBBBBCøøø}}
struct point {CøøøXXXXXXXXYYYYYYYY}
```

Memory layout on BUS64:

```
struct simple {AøøøBBBBCøøø}
array vect [{AøøøBBBBCøøø}{AøøøBBBBCøøø}]
struct outer {Xøøø{AøøøBBBBCøøø}}
struct point {CøøøøøøøXXXXXXXXYYYYYYYY}
```

On BUS32, the natural boundary for simple is 4, which means its size must be 12. The most demanding member in simple is the integer B, whose natural boundary is 4

on BUS32. By following the Golden Rules, the compiler guarantees that B is properly aligned in every possible case, as shown by the memory layouts for structure simple, array vect, and structure outer. In all instances of structure simple, B retains an offset that is a multiple of 4. For BUS16, the same code fragment would yield a natural boundary of 2 and a size of 8 for simple. In all cases, B retains an offset that is a multiple of 2.

Removing any of the padding or trailing bytes would make B improperly aligned in some cases (try to lay out vect if simple has no trailing bytes). It is worth noting that the sizeof operator must always yield the same value when applied to a type. Adding padding or trailing bytes only when simple is used as a component of a structure or of an array would make its size ambiguous. Let's state this as:

**Golden Rule 3:** The size of a structure cannot depend on its context.

### *Packed Structures*

With packed structures, the compiler ignores the natural boundaries of C types. There are never padding or trailing bytes between members of packed structures. See the example that follows.

Memory layout on any bus:

```
packed struct simple {ABBBBC}
array of packed struct simple [{ABBBBC}{ABBBBC}]
packed struct outer {X{ABBBBC}}
packed struct point {CXXXXXXXXYYYYYYYY}
```

### *Unpacked Inside Packed*

Unpacked structures are never packed, even when enveloped by packed structures. If unpacked structures were packed when nested inside packed structures, we would reach the following paradox:

```
struct alpha
{
   char A;
   short B;
   char C;
} solo;
```

```
{AøBBCø} sizeof(alpha) == 6
```

```
packed struct omega
{
   char X;
   struct alpha Y;
} pair;
```

```
(X{ABBC}) sizeof(alpha) == 4 ?!
```

### Bit Fields

There is no `bit` keyword in C. Bit fields are declared through `int` or `unsigned int`. In many compilers, however, you may use any integral type to declare bit fields: `char`, `short`, `long`, or even a packed enumerated type. The integral type determines the byte offset of the first bit, whether the bits are signed or unsigned and whether they come in groups of 8, 16, or 32. Notice that the keyword `packed` affects the alignment of the bit-field containers only. Packing of bits always happens.

Examples of memory layout for bit fields (assuming big endianity):

```
struct ua
{
   unsigned char a : 4, b : 4;
};
```

{aaaabbbb}

```
struct ub
{
   char A;
   short b : 3, c : 3, d : 3;
   short : 0;
};
```

{A|ø|bbbcccdd|dΔΔΔΔΔΔΔ}

```
struct uc { enum color a : 12, b : 12; };
```

{aaaaaaaa|aaaabbbb|bbbbbbbb|ΔΔΔΔΔΔΔΔ}

```
packed enum UBYTE {x1=0u,y1=0xffu};
packed enum UWORD {x1=0u,y1=0xffffu};
packed enum ULONG {x1=0u,y1=0xffffffffu};
packed struct pa
{
   char A;
   UBYTE b : 3;
   UBYTE c : 4;
};
```

{A|bbbccccΔ}

```
packed struct pb
{
   char A;
   UWORD b : 4, c : 4, d : 4;
};
```

{A|bbbbcccc|ddddΔΔΔΔ}

```
packed struct pc
{
    char A;
    ULONG b : 4, c : 4, d : 4;
    ULONG e : 20
};
```

```
{A|bbbbcccc|ddddeeee|eeeeeeee|eeeeeeee}
```

## Tips and Techniques

### *Quick Detection of Size*

A simple trick to detect the size of any type is to "make a mistake" and pass the type as a parameter to any function. Your compiler may reveal the size. For example:

```
1 #pragma options -p68000
2 struct tag {
3      char A;
4      int B; };
5 int foo(int x) { foo(struct tag); }
                                  ^

>> (W) type used as argument; replaced with its sizeof: 6
```

### *Detection of Offsets and Alignments*

A more general approach to discover sizes, offsets, and alignments is to use an inspector program as follows:

```
#include <stddef.h> /* get offsetof macro */
#include <stdio.h>
#define alignof(type) offsetof(struct{char any;type x;},x)
#define print_offset(tag,member)            \
   printf("\noffset of %s.%s\t%d",          \
   #tag,#member,offsetof(tag,member))
#define print_alignment(type)               \
   printf("\nalignment of %s\t%d",#type,alignof(type))
struct x
{
       char a;
       long b;
};
struct y
{
       char a;
       struct x b;
```

*Continued*

```
        long c;
        char d;
};

void main(void)
{
        printf("size of x: %d\n", sizeof(struct x));
        print_offset(struct x, a);
        print_offset(struct x, b);
        print_offset(struct y, a);
        print_offset(struct y, b);
        print_offset(struct y, c);
        print_offset(struct y, d);
        print_alignment(char);
        print_alignment(int);
        print_alignment(struct x);
        print_alignment(struct y);
}
```

### *Tight Packing*

Structures come in three flavors: explicitly unpacked (rare), explicitly packed, and plain. For example:

```
unpacked struct unpk {...
packed struct pk {...
struct plain {...
```

Plain structures are, by default, unpacked. They can all be made packed via the `-KP` compiler option. If you have fairly large arrays of structures, this option may save some space, but it trades off memory with runtime efficiency. In these examples, the additional keywords `packed` and `unpacked` are illustrated; many compilers achieve the same result with `#pragma` directives.

### *Inflating Structures*

Two minor options may be available and can be used to affect the size of structures—namely `-Zn<num>` for unpacked and `-Zm<num>` for packed. Their defaults are `-Zn1` and `-Zm1`; that is, make the size of unpacked and packed structures a multiple of 1 (trivially true). In some cases, you may want to adjust sizes to be a multiple of a given value. For instance, to confirm a memory override during debugging, you may want to specify larger values of `-Zm` and `-Zn`. For instance, `-Zm8` will make the size of all your packed structures a multiple of 8.

### Managing Alignment

Perhaps you have an array of structures, and efficiency reasons dictate they should be 8 bytes long. For instance:

```
typedef packed struct {char head; long load;} MSG;
MSG arry[1000];
```

To make sure that array also starts at a multiple of 8, you can inject an alignment directive just before the array declaration, as follows:

```
#pragma asm
.align 8 # PPC assembler
#pragma endasm
MSG arry[1000];
```

An alternative is to give the array a "roommate" in an unpacked union, as follows (assuming `BUS64`):

```
unpacked union
{
   double alignMeToMultipleOf8;
   MSG arry[1000];
} u;
```

As the alignment requirement of the union is that of its most demanding element (the double), the whole union will be aligned to a multiple of 8. Recent compilers are likely to provide an option (`-Zd<num>`) to the same effect:

```
#pragma options -Zd8
MSG arry[1000]; /* aligned to 8*n */
```

### Forcing Identical Layout Across `BUS16` and `BUS32`

If you are writing an application that is meant to be portable across all variants of the 68 K family, you may want the layout for all structures in your program to be identical across processors.

For unpacked structures, your compiler may support the options `-Za2` and `-Za4` to alter the natural boundaries. The option `-Za2` forces all types with a natural boundary of 4 to "lower" their demands to 2. The option `-Za4` "raises" the demands of the types with a natural boundary of 2 (and a size greater than 2) to 4. Substantially, `-Za2` and `-Za4` allow you to define the bus width.

An example:

```
struct
{
    char A;
    int B;
    float C;
};
```

```
68000 {AøBBBBCCCC}
68020 {AøøøBBBBCCCC}
```

Specify `-Za4` to make the 68000 version identical to the one for the 68020. You could also use `-Za2` to make the 68020 structure comply with the 68000 one, with a savings of 2 bytes at the expense of runtime efficiency for the 68020.

```
-Za2 {AøBBBBCCCC}
-Za4 {AøøøBBBBCCCC}
```

### *Forcing Identical Layout Across* `BUS32` *and* `BUS64`

If there are no doubles, the layout is identical. If there are doubles, it should be pretty obvious that "doubling" the `-Zas` will obtain the desired result:

```
struct
{
    char A;
    double B;
};
```

```
ppc403ga {AøøøBBBBBBBB}
ppc740 {AøøøøøøøBBBBBBBB}
-Za4 {AøøøBBBBBBBB}
-Za8 {AøøøøøøøBBBBBBBB}
```

Be aware that many PowerPC compilers use an alignment of 8 for doubles regardless of the bus width. Ideally, doubles are aligned to 4 for `BUS32` variants and to 8 for `BUS64` variants.

### Test Your Knowledge (Difficult)

Test your knowledge of structure alignment. Assign yourself a point for each correct answer. Check your answers with the solutions at the end of this article and compute your score and expertise as follows:

| Score | You Are |
|---|---|
| 0–7 | Well . . . |
| 8–15 | Structure Apprentice |
| 16–22 | Structure Cadet |
| 23–29 | Structure Master |
| 30 | The Compiler |

1. Is the layout of the following structure the same for `BUS16` and `BUS32` and for `BUS32` and `BUS64`?

```
struct
{
       char A;
       float B;
};
```

2. What is the offset of `X` in the following structure for `BUS16`, `BUS32`, and `BUS64`?

```
struct
{
       char A;
       int X;
};
```

3. Same question for

```
struct
{
       char A;
       double X;
};
```

4. How many trailing bytes can be found in the following `BUS32` structure?

```
struct
{
       char A, B;
       int C;
       char D;
};
```

5. How can you avoid trailing bytes in the previous structure? What size would result?

6. How many trailing bits can be found in the following structure? Would it make a difference to remove the keyword `packed`?

```
packed struct { char a : 7; };
```

7. How many trailing bits can be found in the following structure? Would it make a difference to add the keyword `packed`?

```
struct { short a : 7; };
```

8. How many padding bytes can be found between `X` and `Y`? Are there any trailing bytes?

```
struct { char X; short Y; };
```

9. How many padding bytes can be found between `X` and `Y`? Are there any trailing bytes?

```
#pragma options -pcpu32 -Za4 -Zm4
packed struct
{
        char X;
        short Y;
};
```

In questions 10–16, consider the following structure:

```
packed struct mix
{
        char A;
        struct
        {
                char B;
                double C;
        } T;
};
```

10. Are there padding bytes between `A` and `T` in the structure `mix`?

11. Are there padding bytes between `B` and `C` in the structure `mix.T`?

12. What is the size of the structure `mix` for `BUS16` and `-Zm4`?

13. What is the size of the structure `mix` for `BUS32`?

14. What is the size of the structure `mix` for `BUS64` and `-Zm10`?

15. What is the minimum set of options to force the structure mix to have an identical layout for `BUS16` and `BUS32`?

16. What is the minimum set of options that can make the structure `mix` have an identical layout across `BUS16` and `BUS32` and also make its size 16 bytes?

17. What is the size in bytes of the following structure?

```
struct
{
       short a : 7;
       short b : 8;
};
```

18. Would declaring the preceding structure to be `packed` change its size in bytes? Or the bit offset of `b`?

19. What is the size in bytes of the following array? Are the bit fields signed?

```
typedef packed enum {F, T} BOOLEAN;
struct
{
       BOOLEAN a:1, b:1, c:1, d:1,e:1, f:1, g:1;
       BOOLEAN:0;
} arry[100];
```

20. What is the size in bytes of the preceding array with the following definition of `BOOLEAN`? Are the bit fields signed?

```
typedef unpacked enum {F=0u, T} BOOLEAN;
```

21. Write a C declaration for a tightly packed `BUSany` structure containing only 8 (1-bit) bit fields followed by a `short`.

22. Can unpacked substructures exist inside packed structures?

23. What option would make sure the offset of `B` would be 4 in the following `BUS16` structure?

```
struct
{
       char A;
       int B;
};
```

24. Write a C declaration for a `BUSany` structure that contains a `double` and has a size of exactly 9 bytes.

25. Write a C declaration for an array of 100 single unsigned bits and size = 100.

26. Write a union of size 1 containing 8 bits that can be accessed together as a byte or individually as a bit.

27. What is the size of the following structure if `BUS64` and `-Zm2`?

```
struct
{
       char A;
       struct inner
       {
              char B;
              short C;
       } S;
};
```

28. What offset has `sub` in the following `BUS32` declaration?

```
struct
{
       char A;
       struct inner
       {
              char B,C,D;
              long E;
       } sub;
};
```

29. Determine the size of the following structure if `BUS16/32/64` and if packed or unpacked.

```
struct
{
       char A;
       double B;
}
```

30. What option will give the following C structure a size of 128 bytes?

```
packed struct {char A;};
```

Now check your answers. Add up your points to find your score. If you are dissatisfied with your results—which is not unlikely because the test is difficult—experiment a bit with the compiler and try the test again. After all, what really matters is how much you know in the end.

### Answers

1. No.

   ```
   BUS16: {AøBBBB}
   BUS32: {AøøøBBBB}
   ```

   `-Zn2` or `-Zn4` would force the same.

   Yes, `BUS32` and `BUS64` are identical but for doubles.

2. <2, 4, 4>.

3. <2, 4, 8>.

4. Three, to make the size a multiple of C's alignment:

   ```
   {ABøøCCCCDøøø}
   ```

   Golden Rule number 2.

5. Declare the structure packed. Size 7.

6. One: `{aaaaaaaΔ}`.

   No.

7. Nine: `{aaaaaaaΔ|ΔΔΔΔΔΔΔΔ}`.

   No.

8. One: `{XøYY}` with any bus. No trailing bytes.

9. None: `(XYYø)`. One trailing byte. `-Za4` has no effect on packed structures or on a `short` type, since its size is 2.

10. Never. `A` and `T` are fields of a packed structure.

11. Always. `B` and `C` are fields of an unpacked structure, and the byte after `B` has an odd offset.

12. 12 bytes: `(A{BøCCCCCCCC}ø)`

13. 13 bytes: `(A{BøøøCCCCCCCC})`

14. 20 bytes: `(A{BøøøøøøøCCCCCCCC}øøø)`

15. Three solutions:

```
-Za2 (A{BøCCCCCCCC})
-Za4 (A{BøøøCCCCCCCC})
-KP (A(BCCCCCCCC))
```

16. `-Za2 -Zm8: (A{BøCCCCCCCC}øøøøø)`
    `-Za4 -Zm16: (A{BøøøCCCCCCCC}øøø)`

17. 2 bytes: `{aaaaaaab|bbbbbbbΔ}`

18. No. No.

19. 100: `[{abcdefgΔ}{abcdefgΔ}...]`. Yes, since `BOOLEAN` is a `signed char`.

20. 400:
    `[{abcdefgΔ|ΔΔΔΔΔΔΔ|ΔΔΔΔΔΔΔ|ΔΔΔΔΔΔΔ}...]`

    Unpacked enums are integers—unsigned integers in this case.
    No, they are unsigned, as is their container.

21.
```
packed struct
{
      char a:1, b:1, c:1, d:1, e:1, f:1, g:1; h:1;
      short sh;
};
```

22. Yes. And vice versa.

23. `-Za4: {AøøøBBBB}`

24.
```
packed struct
{
      char A;
      double B;
};
```

25. `struct {unsigned char x : 1;} a[100];`

26. `typedef struct {char a:1, b:1, ..., h:1;} BITS;`
    `union { BITS bits; char allBits; };`
    Use a `typedef` to increase readability. Don't give yourself a point if your answer included an unnecessary packed.

27. Six: `{Aø{BøCC}}`. For any bus, `-Zm2` is immaterial since the structures are not packed.

28. Four bytes. According to Golden Rule 1, `sub` has 4 as its natural boundary—that is, the natural boundary of its most demanding field, the `long E`.
    `{Aøøø{BCDøEEEE}}`

29. Unpacked `BUS16`: 10 `{AøBBBBBBBB}`
    Unpacked `BUS32`: 12 `{AøøøBBBBBBBB}`
    Unpacked `BUS64`: 16 `{AøøøøøøøBBBBBBBB}`
    Packed `BUSany`: 9 `{ABBBBBBBB}`

30. `-Zm128`

## 4.10 Memory and Programming in C

*In the mid-1990s I wrote a regular column in* New Electronics *magazine for a while. The brief was to "educate" hardware engineers in some of the concepts of embedded software. Of course, it didn't last. One of the other embedded software vendors was aggrieved that I was writing this column and complained, even though I had been careful not to write a "sales pitch." This article is closely based upon my first "Byte Site" column. (CW)*

In this article, we focus on various facets of embedded microprocessor software development and address some of the challenges faced by engineers new to this programming environment. Nowadays, the most commonly used programming language for embedded systems applications is C (or C++); assembler being reserved for particularly tricky situations. To an engineer who is familiar with the hardware details of the embedded system, C may appear to be rather abstract and far removed from the "real world." Keeping that in mind, we will take a look at memory usage by programs written in C.

### Memory

Typically, an embedded system uses a number of different types of memory, including read-only memory (ROM), read/write memory (RAM), and CPU registers. In turn, the different types of memory may be used in different ways: ROM contains both code and constant data; RAM contains variables and stack space; registers may be used for different variables at different times. Add to this the complication that input/output devices may be mapped to memory addresses. Memory usage is clearly visible to the assembly language programmer, but the C programmer needs to understand how the development tools accommodate memory usage.

### Sections

The most powerful facility that enables the programmer to control memory usage is support for multiple *program sections*. A *section* is a logical area of memory that is given an arbitrary name and may be located at specific addresses at link time. The simplest possible case might contain two sections, one called "CODE" and another "DATA", each assigned to the addresses of ROM and RAM, respectively.

A cross-compiler (one designed specifically for embedded systems development) will automatically segment the code and data into a number of sections. A typical compiler might, for example, generate up to seven sections for different categories of information. These sections will most likely have default names that may be overridden by the programmer, as required.

As each module (file) is compiled, the resulting relocatable object module contains parts of each section that will be contained in the final executable. The task of the linker is to assemble all the parts of each section and correctly locate the sections in memory.

Fundamentally, this approach works because the programmer does not (ever) need to know the precise address of a variable or a piece of code. It is sufficient to be sure that the code and data have been placed in the right type of memory and that the debugging tools can locate them symbolically, when required.

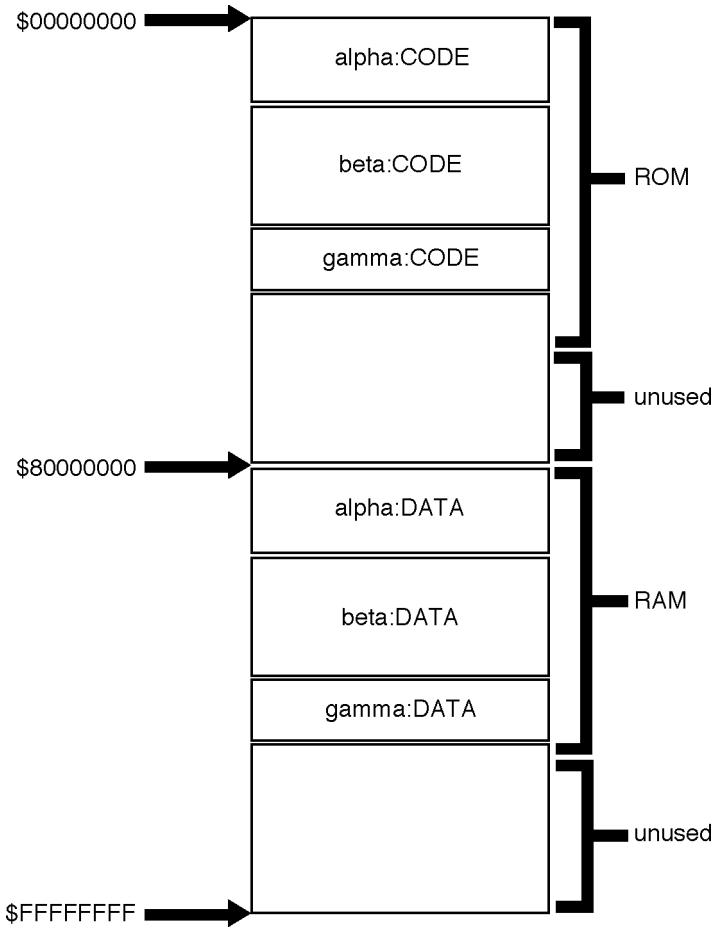



*Figure 4.4: Typical memory map*

Figure 4.4 illustrates this concept, showing the memory map for an application. In this case, ROM is located at address $00000000 and RAM at $80000000. Three modules *alpha*, *beta*, and *gamma* are included, each contributing fragments of the CODE and DATA sections.

## Conclusions

Clearly, for a software developer to easily map code and data to the right type of memory and then to debug the result, tools must be designed specifically for embedded systems software development and include facilities to enable flexible handling of program sections. Only in this context will the programmer find all the control required to complete the task.

## 4.11 Pointers and Arrays in C and C++

*This article is based on another of my "Byte Site" columns in* New Electronics *in 1996. (CW)*

### Pointers and Pointer Arithmetic

For many, pointers are thought of as strange and mysterious entities, and operations on them are considered akin to a black art. It is surprising that even programmers with years of assembler experience, where manipulating addresses is second nature, can come unstuck with pointer arithmetic.

This last point is really the key to the secret. A common question from C language students is "What is the difference between a C pointer and an address?" The answer is that, although the value of a pointer may indeed be an address, it is itself a distinct data type, the characteristics of which are influenced, in turn, by the type of data to which it points.

When the C language was first defined, pointers had "target-specific" types for two reasons. The first was to permit type-specific arithmetic, which we will consider again shortly. The second was the notion that the addresses of different kinds of data might indeed be different in nature. From one perspective, this may seem to have been an unnecessary precaution, because modern computers tend to have simple, single address spaces (if you ignore `near`/`far` in 80x86, I/O space addressing, and the multiple address spaces of 680x0 and SPARC). However, other chips used for embedded systems designs do have distinctly different ways of accessing different kinds of data. The most graphic example is the 8051 series, with its five address spaces.

In C, as with most high-level programming languages, all the arithmetic operations are defined in type-specific ways. For example, the process of adding two integers is quite different than that applied to floating-point numbers, but the programmer need not normally be concerned with this underlying complexity. With an embedded system, manipulation of addresses may be necessary, so an appreciation of how the C language does arithmetic on pointers is likely be useful.

Here is an example of some pointer arithmetic:

```
int *ptr;
...
ptr = ptr + 1;
```

Apart from the passing thought that this is rather Pascal-like in its style (a C programmer would normally use ++ or +=), the code is quite simple. However, a program line just like this resulted in a support call a few years ago, "When I add one to the pointer, why is it that the code generated by my compiler adds four?" On the surface this query seems reasonable. The reason for the addition of four is that integer variables occupy four bytes of memory (with a compiler for a 32-bit processor anyway). The idea is that

when a pointer points at a value in memory, incrementing the pointer should result in it pointing to the next variable of that type. In other words, C automatically scales the arithmetic according to the size of the data type that is pointed to. This even works with user-defined data types (`struct` and `typedef`).

If you need a way to deal with addresses directly in C, the best approach is to use pointers to single-byte data types. A good choice would be a pointer to `unsigned char`. Although this approach will make the arithmetic "easy," accessing nonbyte data will require a cast.

The idea of a generalized pointer, capable of containing the address of anything, was introduced in ANSI C: the `void` pointer. No operations may be performed on a `void` pointer until it has been cast into a "normal" pointer type.

### Arrays and Pointers

Another common query in C courses is, "Just what is the relationship between arrays and pointers—they seem kind of similar, but different?" The fact is that the relationship is very close. An array is just an area of memory with a name (and data type). The name is simply a pointer to the memory area and can be treated just like any other pointer. The only difference is that it is a constant, so it cannot be incremented, assigned to, or modified in any way. The usual way that array elements are accessed is by means of an index in brackets after the array name. What is not obvious is that these brackets are simply an operator (look at the table of operator precedence in Kernighan and Ritchie).

To place the value 99 in the third element of an array `z`, you would probably write:

```
z[2] = 99;
```

However, by treating `z` as a pointer, the same code may be written:

```
*(z+2) = 99;
```

The brackets notation is just an alternative way to show array indexing, which is much clearer and easier to understand. No rules designate one notation as better than the other. A good guide to writing clear code is to treat "real" pointers as pointers and arrays as arrays. For example, to get the data at the location pointed to by `ptr` into the variable `x`, you would normally write:

```
x = *ptr;
```

There is nothing to be gained from the (equally syntactically valid) form:

```
x = ptr[0];
```

The one exceptional circumstance is when you want to take offsets from a pointer that has been passed to a function. Within that function, treating the pointer like an array is reasonable.

### Conclusions

Although the use of pointers causes 70% of the bugs in C/C++ programs, pointers are a necessity for most applications. With care, their use may not be hazardous, and the intelligent use of arrays may simplify matters further.

# 5

# C++

*C is still the default language for embedded software development, but most modern engineers have some knowledge of C++, even if they have yet to apply it in this context. In this chapter, articles cover the broad topic of moving from C to C++, some specific language features are looked at in detail, and particular applications of C++ are outlined.*

## 5.1 C++ in Embedded Systems—A Management Perspective

*Engineers are often attracted by new technology. They are naturally enthusiastic about being on the leading edge—using the latest programming techniques and languages, for example. Even though embedded developers are, on the whole, somewhat conservative, they ultimately have the same mind set. Managers, on the other hand, have an "if it ain't broke, don't fix it" attitude. In any embedded development environment, a healthy tension exists between these two cultures. I wrote this white paper a few years ago in answer to engineers' request: "How can I persuade my manager to pay for C++? I know it's what we need." (CW)*

The technical pros and cons of using C++ for embedded systems applications are widely discussed. This paper considers the matter from the viewpoint of the manager. A software team leader is not concerned with the finer points of a programming language; he is interested only in the benefits that its adoption might yield.

### Embedded Systems Development Teams

In the early days of embedded systems development, the "team" would be just one engineer, who would design the hardware, build the prototype, and implement the software. He would understand all aspects of the system. The software, which was probably written in assembly language, represented quite a small part of the overall design effort.

Over time, as systems became more complex, separate hardware and software development teams evolved. The program design and implementation represented a much greater proportion of the development investment. Software was increasingly written in high-level languages, typically C.

Nowadays, with very large and complex embedded systems being implemented in ever-shorter design cycles, large teams of software engineers may be involved. This results in some challenging management problems.

### Object-Oriented Programming

In recent years, there has been an increase in the popularity of object-oriented programming techniques. The idea is not new. The first object-oriented programming language was implemented in the late 1960s. However, this approach lends itself particularly well to the programming problems of today.

Broadly, object-oriented programming is a technique whereby software is wholly or partly implemented as a set of *objects*. An object is a self-contained package of code and data, with a clearly defined interface to the outside world. *An object may be utilized without the programmer having to understand its inner workings*. This is exactly the same approach that hardware designers use when making use of integrated circuits.

### Team Management and Object-Oriented Techniques

At the philosophical level, object-oriented programming facilitates the "encapsulation of expertise": the specific knowledge and experience of one engineer may be placed inside an object and made available for the use of others, without their needing to acquire that expertise.

The alternative, which must be faced when conventional, procedural programming techniques (such as those imposed by the C language) are used, is that all members of the team must have wide knowledge of the complete application, which is inefficient. Object-oriented programming techniques enable the collective expertise of the team to be *divided, distributed, and managed in an efficient manner*.

### C++ as an Object-Oriented Language

C++ is an extended variant of the C language. The extensions provide object-oriented programming facilities. Strictly speaking, C++ is not an object-oriented language; it is a procedural language with some object-oriented capabilities. This does not devalue C++ as a useful language for embedded systems development. Although there are other languages that may be described as more purely object oriented, none of them have the backwards compatibility (to C) that C++ offers. This is a benefit because it reduces the cost of training engineers and porting existing code. The use of C may continue, with C++ being introduced incrementally, as required. *No sharp learning curve limits continued productivity*.

### Overheads

At any mention of C++, many experienced embedded systems developers will make dark comments about excessive memory use and real-time overhead. They are wise to be concerned about these issues. There are always some resource limitations in an embedded system. The same concerns arose when C was first used. However, over time, three things happened:

- C compilers improved in quality and functionality (the overhead was reduced).
- Microprocessor power and available memory increased (the overhead was less important).
- Demand for increased programmer productivity increased (the overhead was a lower priority).

Exactly the same factors apply to C++.

### The Way Forward

Given that C++ appears to be an attractive option for the future, a number of steps must be taken:

1. Determine the appropriateness of C++ for your type of application.
2. Select and acquire C++ tools.
3. Develop a migration plan—for example, staff training, existing code conversion, and update development procedures.

C++ offers clear benefits to the development manager:

- Increased productivity
- Reduced time to market
- Full use of software team members' individual skills

## 5.2 Why Convert from C to C++?

*Many embedded developers still use C. For many years, there have been ongoing arguments over the benefits of moving to C++. In the early 1990s, Michael Fay (who now teaches at Santa Barbara City College) addressed this topic in a piece for* NewBits, *which formed the basis of this article. (CW)*

The well-publicized advantage of C++ over C is that it supports object-oriented programming. Reflecting more deeply on the topic, we can identify several specific advantages of C++ over C.

### Hide Implementation Details

In C, one clean way to provide a data structure is to encapsulate it in a separately compiled module in which only functions can access the underlying data. Here is a simplified example of a stack implemented that way:

```
static int stack_array[MAX];
static int stack_top = -1;
void push(int i)
    { ... }
int pop(void)
    { ... }
```

In C++, you can hide implementation details on a finer level, within each object. For example, here is a class that defines simplified stack objects:

```
class stack
{
    int stack_array[MAX];
    int stack_top;
public:
    stack()
    {
        stack_top = -1;
    }
    void push(int i)
        { ... }
    int pop(void)
        { ... }
};
```

In this example, `stack_array` and `stack_top` are private class elements, visible only to `push()` and `pop()`, whereas in C, `stack_array` and `stack_top` are visible to all subsequent functions in the module. (The element `stack()` is a constructor—that is, a function invoked when a stack is declared.)

### Reuse Class Code

If a program has more than one stack of `int`s, the single class definition in the preceding C++ example will suffice as the only implementation of `int` stack operations. You can declare many such stacks, each having its own private data:

```
stack s1;
stack s2;
```

In C, you have to expand the stack implementation to cover multiple `int` stacks.

### Reuse Generic Classes

If your only requirement for stacks is for `int` data, the example previously given will suffice. To accommodate a different data type—say `float`—you need to define another, almost identical, class. An alternative is to use a C++ class template, which enables you to outline what a class would look like and leave the compiler to generate code for specific data types as needed.

### Extend Operators

You can extend operators to apply to new, user-defined types. For example, if you define a class `complex` to handle complex numbers, you may wish to define (i.e., overload) the + operator to apply this operation to complex numbers in a natural way. Trying to use an undefined operator on a new data type would, of course, result in compile-time errors.

This capability may be used (or abused) creatively. There is no specific reason why the functionality of a redefined operator needs to mirror that of its standard counterpart—although this can lead to endless confusion. An interesting example is the extension of the << operator in the `stream.h` library. By default, << has the same meaning as in C: left shift. However, `stream.h` defines a class called `ostream`, and if the left operand of << is an `ostream`, << means to output the right argument to the `ostream` and return an `ostream` as the result. Since <<, like most operators, is left-associative, you can output a series of variables and constants by connecting them with <<:

```
cout << "Total = " << i << '\n';
```

### Derive Classes from Base Classes

Suppose you need to define an object (e.g., a `filename`) that is a special case of a simpler object (e.g., a `string`). A `filename` could have a slightly different concatenation operation (+) and a new `open()` function. In C++, you can define `filename` in terms of `string` by declaring it to be a "derived class." A derived class inherits the properties of its base class. In the following example, adapted from Wybolt (1990), class `filename` inherits the properties of class `string`:

```
class filename : public string
{
public:
    friend filename & operator+(filename &, filename &);
    FILE *open();
};
```

With this definition, a `filename` allows all the operations that can be performed on any `string` but has its own + operator and `open()` function. (`string` may have these operations; if it does, they will be superseded in `filename` objects.)

The preceding example illustrates other features of C++ beyond inheritance. To avoid copying large strings, the `&` operators denote call and return by *reference* rather than by *value*. The `friend` keyword indicates that a function has access to the nonpublic members of objects in the class.

### Avoid Errors Through Function Prototyping

C++ requires you to give the types of all parameters and to declare all functions before they are used:

```
extern void func(long l);
...
func(1, 2); // error!
```

This requirement is necessary so that the C++ compiler can apply various type conversions. It also prevents you from passing too many parameters or parameters with incompatible types.

Although prototyping is optionally available in C, it is mandatory in C++.

### Add Parameters Without Changing Function Calls

In C++, you can specify the default values of parameters in case the trailing parameters are omitted at the calling point:

```
extern void f2(int j, int i=0);
...
f2(3); // j=3; i=0
```

Thus, by giving default values in the function declaration, you can expand the number of parameters of an existing function without changing all the calls. This makes it easier to extend the application of a function.

### Using Safer, Simpler I/O

By using the `stream.h` standard library, you don't have to match `printf()` format strings with parameters, which is a common source of C errors.

C:

```
#include <stdio.h>
printf("%d\n", floatvar); /* error: incorrect format */
```

C++:

```
#include <stream.h>
cout << floatvar << "\n"; // format implicity matches type
```

If you still need `printf()` functionality, a series of functions are available to do simple formatting (`oct()`, `dec()`, `hex()`, etc.). Another useful function is `form()`, which operates rather like `sprintf()`.

### Improve Performance with Fast Inline Functions

Within the module in which a function is defined, you can declare it to be `inline`, giving a hint to the compiler that the function code should be substituted rather than called. Inline functions are faster than ordinary functions because there is no call or return overhead, and the inlined code is subject to optimization. Inline functions are safer than macros because they can impose type promotion and do not evaluate their arguments more than once. The following example is from the standard include file `defs.h`:

```
inline int max(int a, int b)
{
    return a > b ? a : b;
}
```

When you define class member functions, you have the option of defining the function within the class body or simply declaring it for definition elsewhere. The former method has an implied `inline` declaration and is typically used for simple, one- or two-line function definitions.

### Overload Function Names

C++ lets you redeclare (i.e., overload) the same function name as long as the parameters' types are not all the same. Overloading is convenient because you do not have to remember different names for functions that perform essentially the same task. For example, the aforementioned `defs.h` provides two inline `max()` functions for other types as well as `int`:

```
inline float max(float a, float b)
{
   return a > b ? a : b;
}

inline double max(double a, double b)
{
   return a > b ? a : b;
}
```

The C++ compiler will invoke the function whose parameters match the actual parameters' types. If no match is found, an error will result. This particular example could have been more efficiently and flexibly implemented by using a C++ function template, but it illustrates the point of overloading function names. Overloaded functions still serve a useful purpose when the code within each variant of the function is quite different.

### Embedded System Support

Moving from C to C++ for an embedded project may be fine, so long as the C++ compiler offers the same specialized functionality that you need:

- Inline assembly code
- `interrupt` functions
- Correct implementation of `const` and `volatile`
- `char` and `short` bit fields
- Packed structures and classes
- Efficient implementation of C++ constructs (which may require an optimized linker)

### Change Involves Effort

Newcomers to C++ might hope that C++ is simply a superset of C and, therefore, by simply taking the existing C code, they can run it through a C++ compiler, and everything still works.

Unfortunately, that is not quite the case. It is very likely that some changes will be necessary. The language must be studied more carefully, and a migration strategy should be developed.

### Massage C Code into C++

To cushion the blow, you may convert some C modules to C++, leaving others in C. In this case, each C++ module must have special declarations for all C `extern` functions, as well as for all its own C++ functions that are directly called from C. The special declarations are enclosed in the following construct:

```
extern "C" { ... }
```

This construct may enclose types and variables—even entire header files. For example:

```
extern "C"
{
    #include "externs.h"
    int forward_func(int i);
}
...
int forward_func(int i)
{
    ...
}
```

Note that functions with `extern "C"` linkage cannot be overloaded. This is not a problem if you are linking C code into a C++ application but could cause you problems when you're working the other way around.

### The Hard Part: Designing Objects

Now that you are ready to take advantage of the features of C++, you should spend some time designing objects that will be useful to you and to other members of your organization. When designing objects, you should ask what fundamental operations define each object. It is important to exclude nonessential operations. For example, as shown in Dewhurst and Stark (1989), the fundamental operations on complex numbers are converting from `float`, adding, multiplying, and so on. Extracting the components is not a fundamental operation:

```
class complex
{
    double re, im;
public:
    complex(double r=0.0, double i=0.0)
        { re = r; im = i; };
    friend complex operator+(complex, complex);
    friend complex operator*(complex, complex);
    ...
    // do not do this:
    double first()
        { return re; }
    double second()
        { return im; }
};
```

The reason that `first()` and `second()` should not be included is that they allow users of complex objects to write programs that depend on the underlying representation of complex numbers to be Cartesian coordinates. This dependency can exist even though the users have no direct access to `re` and `im`. If the provider of `complex` later changes the underlying representation to the polar coordinates `rho` and `theta`, and `first()` and `second()` return `rho` and `theta`, respectively, programs that use `complex` would be invalidated.

### If It Ain't Broke, Don't Fix It

Some C programs should just be left in C. If your program is fairly stable, and you don't expect to make significant changes to it, converting it to C++ may be an interesting exercise, but it is not justified from an engineering standpoint. However, if you are continually changing your program, you can make a gradual transition to C++ using the techniques described in this article.

### References

1. Nicholas Wybolt, "Experiences with C++ and Object-Oriented Software Development," *ACM Software Engineering Notes*, 15, 2 (April 1990).
2. Stephen C. Dewhurst and Kathy T. Stark, *Programming in C++*, Prentice Hall, 1989.

## 5.3 Clearing the Path to C++

*Over many years, engineers have considered the transition from C to C++ for embedded applications. I have given many seminars on the topic and, in collaboration with Lily Chang in 1993, wrote a piece for* NewBits, *upon which this article is based. (CW)*

### A Strategy for Transition

Object-oriented programming techniques and the C++ language have rapidly gained popularity among software developers in all spheres of computing. For embedded systems development, especially of larger, more complex systems, the object-oriented programming approach is particularly apposite, since it lets you hide intricate parts of your program.

Compared to other object-oriented programming languages, C++ has its pros and cons, but that is not the subject of this article. C++ is successful because the C language is well established and represents (almost) a complete subset of C++. As a result, the learning process and the application of the C++ language may be approached incrementally. We will explore this approach to using the language in this article.

### Evolutionary Steps

To make the move from programming purely in ANSI C to full-scale application of C++ requires a series of evolutionary steps. There are three "stepping stones":

1. Applying reusability (i.e., linking C and C++ modules): New modules written in C++ may be linked with older C code, or old C language modules may be reused in a new C++ project.
2. Writing Clean C: C language code should be written with additional care (and older code overhauled) to ensure its acceptability as a C++ subset. This permits older C code to be treated as if it were C++—that is, processed by the C++ language build and debug tools.
3. Introducing C++ :C++ language features can be gradually introduced as they are learned, and their suitability established for the application in hand.

### Applying Reusability

The first stage is to simply link together C and C++ modules to add C++ to an existing C project, or reuse C modules in a new C++ project. However, this procedure is not totally straightforward. One of the very attractive features of C++ is *type-safe linkage*. This feature ensures that the correct correspondence between a function's definition and its use (in terms of parameter number and types and function return types) is monitored. Type-safe linkage is realized by the compiler. It is achieved by "mangling" the function name to incorporate the types of its parameters. In addition, the return type information is included for the name mangling of pointers to functions. Applying the same mangling to a definition and call should yield the same modified function name

and, hence, no link errors. Having the compiler mangle function names has the advantage of the independence of the binary formats in use and provides built-in support for overloaded functions (which was really the original motivation for the technique).

Of course, a C function will not have had its identifier mangled if it is processed by the C compiler. However, a call to the function from C++ code will reference a mangled form of its name, resulting in link errors. To overcome this problem, you can define in C++ the external linkage of a function as of some "foreign" form. By declaring a function as `extern "C"`, its name is not mangled by the compiler.

Less known is the technique whereby conventional `extern` declarations can be enclosed by braces and preceded by `extern "C"` to encompass all of them in its effect. For example:

```
extern "C"
{
   extern void alpha(int a);
   extern int beta(char *p, int b);
}
```

This is particularly useful if conditional compilation (`#if . . . #endif`) is used to activate the facility during the transitional phase from C to C++. This code is portable between ANSI C and C++ compilers:

```
#ifdef _cplusplus
extern "C"
{
#endif
   extern void alpha(int a);
   extern int beta(char *p, int b);
#ifdef _cplusplus
}
#endif
```

Ironically, this problem did not occur with early definitions of the C++ language (prior to V2.0) since function name mangling was originally used just to differentiate between overloaded functions. The first (and, maybe, only) instance of a function did not need to be mangled to have a unique name; only the second and subsequent instances needed new names. When it was realized that mangling all the names gave type-safe linkage and more secure function overloading, this method was incorporated into the language, and the `overload` keyword rendered redundant. With current versions of C++, this necessary complication is a major incentive to progress to the next step in the move to C++.

## Writing Clean C

Since ANSI C is essentially a subset of C++, it would seem reasonable to use the C++ build and debug tools on the old C code and simply write all the new parts in C++. However, there are numerous minor exceptions that make the conversion process less than 100% straightforward. Even if you could not circumvent these exceptions, the fact that C++ is a superset of C is a major asset when learning and understanding the new language.

The solution to these incompatibilities is to write C (or modify it) to take them into account. We call the resulting dialect "Clean C," which is ANSI C written in such a way that it is acceptable as a C++ subset. The following sections describe the major issues to address when writing Clean C.

### *Type Checking*

C++ enforces a higher degree of type checking on function calls; function prototypes are mandatory. In Clean C, make sure that all functions have valid prototypes, even when the only calls to a function are elsewhere in the same file.

It is easy to code an erroneous function call in C:

```
void q()
{
    char *p;
    f(p);
}
f(int i)
{
    ...
```

In Clean C an error is flagged:

```
void f(int);

void q()
{
    char *p;
    f(p); // compile-time error
}

f(int i)
{
    ...
```

### Casting

C++ is more "careful" about loss of data from implicit casts of types from a higher precision to a lower one. In Clean C, always perform such casts explicitly. For example, in Clean C:

```
long big; char little;
little = (char) big;
```

### Enumerated Types

ANSI C does not take `enum` variables very "seriously"; they are viewed as being a convenient alternate `#define` notation. A function parameter, for example, may be declared as an enumerated type, but a call is still permitted where an integer value is provided.

C++ is stricter in this respect. For example:

```
enum greek { ALPHA, BETA, GAMMA, DELTA };
void func(enum greek);
. . .
func(GAMMA); // not func(2)
```

A call in the form `func(4)` would result in an error from the C++ compiler.

### Character Strings

In ANSI C a `char` array that is statically initialized to be a character string need only be large enough to hold the characters of the string; that is, it is not essential to accommodate the null terminator. This is not the case in C++ and, hence, in Clean C.

In C it is possible to allocate minimal space:

```
char str[3] = "xyz";
```

In Clean C the `NULL` should be accommodated:

```
char str[4] = "xyz"; // room for NULL
```

or better, allow the compiler to allocate the space:

```
char str[] = "xyz";
```

### *Scope of struct and enum*

In C++, the scope of `struct` and `enum` data types has limitations that are not present in ANSI C. In C, if you define a `struct` within a `struct`, you can use the inner structure outside of the outer one. Because you can't do this in C++, define a nested structure in Clean C only if its use is suitably confined.

Inner-nested structures can be used in C:

```
struct out
{
    struct in
    {
        int i;
    } m;
    int j;
};
struct in inner;
struct out outer;
```

In Clean C the scope of structures must be clarified:

```
struct in
{
    int i;
};
struct out
{
    struct in m;
    int j;
};
struct in inner;
struct out outer;
```

### *Multiple Declarations*

In ANSI C, it is permissible to include multiple declarations of a (single) variable in a file, even though the second and subsequent declarations are redundant and the practice can lead to confusion. This technique is not legal in C++, and thus should not be done in Clean C.

A variable may be declared several times in C:

```
int i;
f()
{
   ...
}
int i;
...
```

In Clean C only one true definition can be included:

```
int i;
f()
{
   ...
}
extern int i;
...
```

### Extra Keywords

C++ has additional keywords that are not reserved in ANSI C. These include:

```
asm       friend     private      try
catch     inline     protected    this
class     new        public       virtual
delete    operator   template     throw
```

In Clean C, these keywords must be avoided. Some keywords that can be particularly troublesome are: `class`, `new`, `template`, and `delete`. You can, however, apply a naming convention that avoids all clashes with keywords, such as initial letter capitalization.

These extra keywords are not an issue in C:

```
struct class
{
   int private;
};
void copy (char *old, char *new);
```

In Clean C capitalization solves the problem:

```
struct Class
{
   int Private;
};
void Copy (char *Old, char *New);
```

### *Compiler Assistance*

Some of the requirements of Clean C are easier to apply than others. A number (mandatory prototyping, for example) may be available as options with an ANSI C compiler. Such facilities permit the transition to Clean C to be commenced at an early opportunity.

## C+—Nearly C++

Having programmed in Clean C and having passed the code through the C++ build tools, you can now take advantage of C++ features. As you learn about and discover needs for these features, incorporate them into your code incrementally. The use of C++ features will increase at a gradual rate until the program is transformed from a C/C++ dialect that we call "C+" to code that is entirely written in C++.

Let's first look at some of the simpler, syntax-enhancing C++ features:

### *Comment Notation*

C++ has an alternate end-of-line comment form, specifically the // notation. This notation is actually being reintroduced; it was lost many years ago in the transition from BCPL to C. This is particularly useful for commenting out code and adding annotation to variable declarations.

In C, comments must be terminated:

```
int *vp; /* pointer to store */
```

In C+, a comment can finish at the end of a line:

```
int *vp; // pointer to store
```

### *Reference Parameters*

Reference parameters are useful for writing clearer code to hide the necessary use of pointers.

In C, using pointers is the only option:

```
void swap(int *to, int *from)
{
    int temp;
    temp = *from;
    *from = *to;
    *to = temp;
}
```

Reference parameters make the code clearer in C+:

```
void swap(int &to, int &from)
{
    int temp;
    temp = from;
    from = to;
    to = temp;
}
```

References also avoid null pointers dereferenced at a call site and avoid infinite recursive calls.

### Placement of Variable Definitions

In C++, local variable definitions can be placed almost anywhere, unlike C, where they must be placed before the executable code.

C requires variables to be defined at the top of the function:

```
w()
{
    int x, y;
    for (x=0; x<3; x++)
           f(x);
    ...
    for (y=5; y>0; y--)
           g(x,y);
}
```

In C+, variables may be defined at the point of use:

```
w()
{
    for (int x=0; x<3; x++)
        f(x);
    ...
    int y;
    for (y=5; y>0; y--)
        g(x,y);
}
```

As your C++ implementation progresses, you can incorporate more sophisticated C++ features such as the following:

### *Constructor Functions*

The constructor function capability in C++ can facilitate the initialization of `struct` member variables (as, in C++, a `struct` is just a degenerate class).

In C a structure needs to be explicitly initialized:

```
struct list
{
    int data;
    struct list *ptr;
};

void init(struct list *);

struct list links;
init(&links);
```

Initialization can be implicit in C+:

```
struct list
{
    int data;
    list *ptr;
    list(int d=0, list *p=NULL)
    {
        data = d; ptr = p;
    };

};

list links; // initialized automatically
```

### Dynamic Memory

A common error in C is the allocation of dynamic memory without the corresponding deallocation. The use of a C++ class destructor provides a means to avoid this problem.

It is easy to forget to deallocate memory in C:

```
void f()
{
    char *sp;
    sp = (char *) malloc(1024);

}

g()
{
    f();
}
```

A destructor can handle deallocation in C+:

```
extern "C" { char* malloc(int); }

struct memory
{
    char *p;
    memory(int s=1024)
    {
        p = (char *) malloc(s);
    };
    ~memory() { free(p); };
}

g()
{
    memory mob;   // allocate memory
}                 // clean up automatically
```

### Error Handling

In C, the handling of error conditions (exceptions) is not specifically accommodated in the language, so it is difficult to implement securely. A useful feature of C++ is exception handling.

The definition of an exception, in this context, should be appreciated. An exception is an unintended synchronous event, not an interrupt. For an embedded system, the controlled handling of exceptions is particularly important. Simply terminating the program with an error message is rarely an option.

The typical use of exception handling in C++ is to facilitate communication of an error condition between reused code and the new application using it; for example, between a library function and its caller.

The programming of exception handling is a two-stage process:

1. A function that has detected an error "throws" an exception, using the keyword `throw` and possibly an object of an appropriate type to convey details of the exception. This is in the hope that the exception is "caught" later by the caller or some other routine in the call chain.
2. The calling function encloses calls to the reused code in a `try` block, followed by a series of `catch` blocks (exception handlers) corresponding to one or more types of "thrown" objects. Of course, the opportunity still exists for there to be no `catch` block corresponding to a thrown object, and for an error to go unchecked. In that case, an unexpected exception is raised.

### Conclusions—The Path Ahead

With C++ established as the most appropriate language for the next generation of embedded systems, now is the time to take the first steps towards its use. There are various strategies that may be adopted, but a cautious, incremental approach has many merits.

## 5.4 C++ Templates—Benefits and Pitfalls

*In the 1990s, as C++ was beginning to gain popularity among embedded developers and certain features were getting "bad press," I wrote a piece for* NewBits, *in collaboration with Michael Eager, about the usefulness of templates; this article is based upon that work. Another article in this chapter, "Looking at Code Size and Performance with C++," expands upon some of the issues raised here. (CW)*

As C++ evolved, various new features were included in the language. Important examples include multiple inheritance, transparent function overloading, exception handling (EHS), and class function templates. In the embedded context, the first two are widely understood; the latter two, less so. In this article, we will focus on function templates from the viewpoint of the embedded systems programmer: examining what they are, how to use them, how they are implemented, and pitfalls in their use.

### What Are Templates?

In broad terms, templates are a means whereby the designer of a function or class parameterizes more of the function's or class' properties than was previously possible. The most obvious property that may be subjected to this treatment is the type of data contained within an object or processed by a function.

In this article we will concentrate on function templates, simply because they can be described with minimal preamble. All the same concepts and much of the syntax is identical for class templates.

#### *Using Function Templates*

Consider this code:

```
void swap(int& x, int& y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

This is a simple C++ function, which effects a swap of the contents of the two integer parameters `x` and `y`. It is generally unremarkable, except that it takes advantage of C++ reference variables to render the code more readable than the explicit pointer manipulation than would be necessary in a C implementation of the same function.

The problem with this `swap()` function is that it can only deal with a pair of `int`s. Of course, a similar function that operates upon, say, two `float`s, could easily be written. With C++ function overloading, it could even have the same name in order to enhance readability. However, this does not solve the problem of the function being limited to

two `int`s or two `float`s. To accommodate all eventualities, an indefinite number of overloadings would be necessary.

Another solution to this kind of problem may well be to use a `#define` macro. A macro may be able to cope with calls of this form:

```
swap(v1, v2)
```

Since the expansion of this macro would most likely make more than one reference to the parameters, the fact that the expressions `v1` and `v2` have no side effects (because they are just variables) is significant. However, the following would still cause difficulties:

```
swap(v[i++],w[i++])
```

since `i++` and `j++` would be referenced twice and, hence, incremented twice.

The solution is to use a function template, thus:

```
template <class X> void swap(X& x, X& y)
{
    X  temp;
    temp = x;
    x = y;
    y = temp;
}
```

This example defines a generalized function that operates on data of type `X`. The actual type of `X` will not be determined until the template is created (i.e., *instantiated*). Using templates allows the designer to focus on implementing an algorithm that can be used for any type of data.

As another example of using a specific algorithm for different data types, consider the binary search algorithm. Although this algorithm is described in many books, you can still make a mistake in writing the function. But once the function is written and tested for one type, you can reuse it wherever needed.

Here is a template that implements the binary search function:

```
template <class T>
int bsrch(const T data[], int num_elem, const T val)
{
    int max = num_elem, min = 0, mid;

    while (max > min)
    {
        mid = (max + min) / 2;
        if (data[mid] == val)
            return mid;
        if (data[mid] < val)
            min = mid + 1;
        else
            max = mid - 1;
    }
    return -1;
}
```

This function expects you to give it an array that is sorted in ascending order, the number of elements in the array, and a value to search for. It will return the index into the array if the value is found or `-1` if it is not.

To use this template, all you have to do is call the function with the correct parameters. The compiler will automatically create a function that will work on the data type of the array.

Care must be taken when writing a function template. For example, the `bsrch` function template will work for any type that has the operators == and < defined. But if you use it to search an array of pointers to strings, you will be surprised. The function will compare the values of the pointers and not the strings.

In this case, the best solution is to create a string class that has the == and < operators defined as member functions. These functions would call `strcmp` and return `TRUE` or `FALSE`.

### Template Instantiation

The definition of a template does not, in itself, result in the generation of any code. It is simply a specification to the compiler for a function that should be instantiated when a corresponding (otherwise unresolved) function call is encountered. Typically, function templates are placed in header files; use `#include` to make them visible to the compiler.

This code:

```
int i, j;
float a, b;
...
swap(a, b);
swap(i, j);
```

would result in two instantiations of the `swap()` function: one for two `int`s (`i` and `j`) and one for two `float`s (`a` and `b`).

### Problems with Templates

Although templates represent an elegant solution to a very real problem, their use may cause unexpected difficulties. Using a template with many different types of data can result in many copies of essentially identical code. Moreover, the implementation of templates can result in multiple copies of identical binary code, which may limit their usefulness for embedded systems development.

#### *Program Overhead*

As discussed previously, the attraction of function templates, compared with overloaded functions, is that you write the code just once. This single instance of source code is a real advantage in both productivity terms (i.e. writing the code in the first place) and in ongoing efficiency (i.e. later maintenance of the code; there is only one function to modify).

However, this efficiency may lead to a false sense of security. It must be remembered that, each time a template is applied to a new situation (i.e. instantiated for a new parameter data type), a new version of the (binary) code is generated. In other words, the use of templates maximizes programmer efficiency (which is good) but does not have any impact upon the space/time efficiency of the resulting program. If the development tools do not perform well, a very significant program overhead may unexpectedly develop.

#### *Duplicate Instantiations*

To reiterate: when the compiler "sees" a function template, it accepts it as a specification for functions that it can generate when corresponding calls are encountered. The resulting functions (template instantiations) are effectively static functions; they are local to the current module. This is necessary because the responsibilities of the compiler itself are limited to the module. The implication of this seemingly innocuous implementation detail is that identical code may be present in many modules, where identical template instantiations have occurred.

For host computers, with very large memory capacities, this redundant code duplication may be a reasonable price to pay for the programmer efficiency gained by the use of

templates. For embedded systems, where resources like memory are precious, the cost can be too high.

What is the solution? With many C++ development toolkits, the only solution is to limit or prohibit the use of function templates. A solution may be effected by implementing a "smart" linker. At link time, identical functions (which can be template instantiations or "out-of-line" inline functions) can be identified and merged, so that only one copy of the code exists in the executable image. The memory requirements of many programs built with tools employing this facility may be reduced very substantially.

### Multiple Template Parameters

There are limitations in the function templates presented so far, since only a single data type has been parameterized. This need not be the case. For example, this function template:

```
template <class Y> int bigger(Y a, Y b)
{
   return (a>b);
}
```

yields a function, `bigger()`, which returns a logical value indicating whether or not the first parameter is larger than the second. As implemented, this function will work satisfactorily for, say, two `int`s or two `float`s. However, it will not work for two parameters of differing type, even if this would be logically acceptable. This capability can be imparted by changing the first line of the template to:

```
template <class Y, class Z> int bigger(Y a, Z b)
```

The type of each parameter is now considered individually.

### Other Template Applications

Although function parameter and internal object types are the most common template parameters, the facility is not specifically limited to this application. The template parameters may, in principle, specify anything that may be determined at compile time.

A useful example is shown here:

```
template <class X, int S> void f(X n)
{
   X buff[S];
   buff[0] = n;
   ...
}
```

where an array size is parameterized. This example is particularly pertinent to embedded systems because it eliminates possible heap usage, which is a common source of runtime difficulties.

One limitation is specific to function templates (and does not apply to class templates): at least one of the template parameters must be a function parameter. This is because the C++ function overloading mechanism relies on functions being differentiated by the type of the parameters in the call. Without this difference existing between template instantiations, ambiguities would result.

## Conclusions

Templates are a useful addition to the C++ language. Their use can improve programmer productivity and code readability. However, their use must be considered carefully because the code generated may be unexpectedly large. Also, careful selection of development tools will be beneficial in avoiding redundant code overheads.

## Postscript

*Since the article was written, C++ templates have developed significantly, and much of this development was not anticipated. The C++ standard library consists of a large number of class templates, as well as a few function templates. Newer class libraries, such as Boost, extend the Standard Template Library with powerful components such as reference-counting pointers.*

*Templates, in many ways, have extended C++ to be a much richer and more robust language, although at the expense of ease of understanding and debugging. Error messages that involve templates remain less than clear and understandable, even in the best of compilers. (ME)*

## 5.5 Exception Handling in C++

*While C++ is widely used for embedded systems programming today, it was in the early stages of acceptance in 1995 when I, in collaboration with Michael Eager, wrote a piece for* NewBits, *upon which this article is based. (CW)*

This article is an introduction to a less-understood facility in the C++ language—the exception handling system (EHS).

### Error Handling in C

In C, there is no built-in facility to deal with error conditions that may occur during the execution of the code. For example, if you call a library function and it detects an error, how is the error dealt with?

The convention is to utilize a global variable, `errno`, which is set to a specific nonzero value by a library function if it detects an error condition. This convention has several problems:

- It is awkward. Writing code that checks `errno` for each library function call is impractical.
- It may be impossible. If a library call is made by code generated by the compiler, the programmer may be unaware that a check is required.
- Reentrancy can be a problem. What happens if an interrupt (or task swap) occurs between the library call and the `errno` check? This situation may result in another library call that corrupts `errno`. Some modern implementations do circumvent this problem.

Exception handling in C++ attempts to solve these difficulties by incorporating an error-handling facility into the language.

### Does Not Involve Interrupts

A common misconception associated with C++ exception handling is that it has something to do with interrupts, which are confusingly called "exceptions" on some devices. Exception handling in C++ does not involve asynchronous events; it is a method to deal with synchronous error conditions.

### C++ Exception Handling

The conceptual model for exception handling in C++ is very simple:

- A library routine that detects an error can `throw` an exception.
- An exception handler can `catch` the exception.
- Blocks of code that activate exception handling `try` to catch exceptions.

These actions, which are keywords, are described in the next section.

Exceptions are characterized by type. The type may be a built-in one (`int`, `float`, `char`, etc.) or user defined (a class). User-defined types are most common. An exception can also have a value that can be interrogated by the exception handler.

### *Keywords*

Exception handling is implemented using three keywords: `throw`, `try`, and `catch`. The syntax for them is shown in this section. Code to activate EHS should be enclosed in a `try` block, thus:

```
try
{
    // code that calls library functions
}
```

This is followed by one or more `catch` blocks:

```
catch (int n)
{
    // exception handling code
}
```

where, in this case, exceptions of type `int` are caught.

A library function that has detected an error can throw an exception with the following:

```
throw 99;
```

resulting in an exception of type `int`, with the value 99 being thrown. Figure 5.1 illustrates the modularity of the EHS.

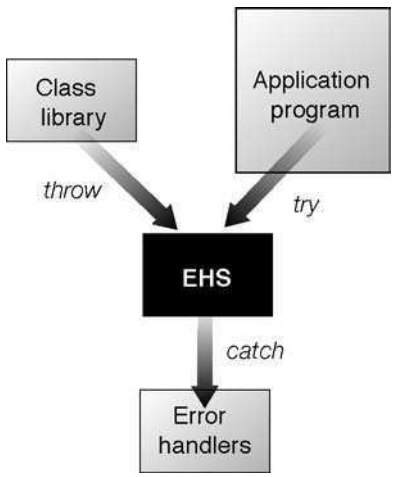



*Figure 5.1: EHS modularity*

### An Example of Exception Handling

This code illustrates a more complete example of the use of exception handling:

```
char store[100];

void scopy(char* str)
{
    if (sizeof(store)+1 < strlen(str))
        throw –1;
    strcpy(store, str);
}
void get_string()
{
    char buff[100];

    cin >> buff;
    scopy(buff);
}
main()
{
    try
    {
        ...
        get_string();
        ...
    }
    catch (int err)
    {
        cout << "String too long!";
    }
}
```

The function `scopy()` takes a string parameter and copies it to an array, `store`, using the standard `strcpy()` library function. Before attempting the copy, the function checks the size of the string and, if it is too large, throws an integer exception with the value –1.

The function `get_string()` accepts some text from the console into a buffer and passes it as a parameter to `scopy()`. Notice the C++ input notation usage.

A `try` block exists in `main()`, which includes a call to `get_string()`. Following the `try` block is a single `catch` block, which catches `int` exceptions. This block prints a useful message. The value of the exception is collected in `err` but not used.

### *Multiple Catch Blocks*

In real applications using EHS, multiple `catch` blocks accommodate exception types that may be thrown by library functions. For example:

```
catch (int n)           // catches ints, including value
{
   // handler code
}
catch (float)           // catches floats, discarding value
{
   // handler code
}
catch (complex c)       // user-defined data type
{
   // handler code
}
```

## Special Cases

Situations can occur that cause unusual exception handling requirements.

### *No Matching Catch Block*

What if an exception is thrown where no `catch` block exists that corresponds to the exception type? If this occurs, the `terminate()` library function is called. In practice, the version of this function in the library is most likely a stub and does not attempt to handle the situation—it just terminates the program. You must supply your own `terminate()` function and incorporate it during linking.

### *Catch Everything*

A series of `catch` blocks are normally provided, which includes entries for all possible exception types. Circumstances can occur that require a simpler processing model. You also may be unaware of all possible exception types that can be thrown by a library. Both situations can be addressed using a special `catch` block construct, as follows:

```
catch (...)
{
   // handler code
}
```

This block is a handler definition for all exceptions, regardless of type. It is impossible to ascertain the exception value or type in the handler.

This construct also means "catch all *other* exception types." It may be used after more specific exception handlers to catch any exceptions that were not explicitly addressed.

### Exception Objects

A common technique is to create a hierarchy of exception classes that are all derived from a single base class. This allows us to identify the nature of an exception more precisely than by just throwing an `int` or a `float`. The following code illustrates this approach:

```
class general_exception
{
public:
   general_exception() {}
};

class memory_error : public general_exception
{
public:
   memory_error() {}
};

class timeout : public general_exception
{
public:
   timeout() {}
};

class io_error : public general_exception
{
public:
   io_error() {}
};

void func1()
{
   try
   {
      func2();
   }
   catch (timeout)
   {
      cout << "Timeout\n";
   }
}
```

*Continued*

```
void func2()
{
    static int x = 0;

    if (++x % 2)
        throw timeout();
    else
        throw memory_error();
}

int main()
{
    for (int loop=0; loop<10; loop++)
    {
        try
        {
            func1();
        }
        catch (general_exception)
        {
            cout << "General Exception\n";
        }
    }
}
```

First, the base class for our hierarchy, `general_exception`, is defined, from which three further classes are derived: `memory_error`, `timeout`, and `io_error`. Note that none of these classes contain any data. We could add data members to each of them to pass details about the exception from the place where the error is encountered to the place it is caught.

Our `main()` routine, at the bottom, consists of a simple loop in which we call `func1()` 10 times. This is within a `try` block, so if `func1()`, or any function that it calls, throws a `general_exception`, we can catch it in the succeeding `catch` block.

`func1()` also has a `try` block. In this `try` block, we call `func2()`. If a `timeout` exception is thrown, we will catch it in the local `catch` block and print a message.

`func2()` will alternately throw a `timeout` exception or a `memory_error` exception each time it is called. In a real application, this might be a routine that encounters a variety of different situations and throws different exceptions.

When `func2()` throws a `timeout` exception, it is caught by the lowest level `catch` block that can accept a `timeout` exception. This is in `func1()`, which will issue a message and then return to where it was called in `main()`.

When `func2()` throws a `memory_error` exception, the EHS will search for a matching `catch` block. A `catch` block matches the thrown exception if the types match or if they are convertible. As the runtime routine searches for a matching `catch` block, it checks the `catch` specification in `func1()` and sees that it does not match. The `catch` block in `main()` specifies `general_exception`, which is the base class for `memory_error`. A derived class can be converted to a base class (losing whatever data is added in the derived class), so the `catch` block in `main()` will catch `memory_error` or any of the other exceptions derived from `general_exception`.

As this example shows, we can nest exception handlers so that the higherlevel handlers take care of general (and often more extreme) actions such as restarting a system. The lower level handlers take more specific actions when they encounter an error, such as retrying an I/O operation.

Using a class as an exception object allows us to group similar exceptions together by creating derived classes. All of these can be caught by the same exception handler. This object-oriented approach to exceptions offers great flexibility in organizing the exception structure of a program and allows us to add exception types to the program as they are needed.

### EHS and Embedded Systems

Like the C++ language, exception handling was not specifically designed for use with embedded systems. Embedded system environments demand special consideration.

#### *Overheads*

Embedded systems developers are concerned about the efficient use of limited resources. Hence, a common question is "Does EHS usage result in a runtime or memory overhead?" Of course it does. In software development, a *Law of Conservation of Effort* exists: you cannot get something for nothing. C++ exception handling provides an example in which the compiler performs operations, which would normally be the programmer's responsibility. It eliminates the need to write code to check for error conditions.

A very important factor concerning overhead is that compilers supporting EHS tend to generate additional code, regardless of EHS usage. This could result in a serious problem of unexpected overhead. Some compilers include a switch to control EHS support, with the switch "off" by default. You need to explicitly activate EHS if you are planning to use EHS (thus implicitly accepting the overhead).

### Conclusions

The exception-handling facility in C++ may be useful in an embedded systems application. It offers a clear and consistent method of handling unusual conditions within a program.

### A Case for Simple Exception Handling

Since overheads are so important in many embedded systems, care needs to be taken in the application of EHS. Keeping it simple is the best approach. To appreciate the point here, consider a nonembedded example:

You are running a Microsoft Windows program, and a runtime error occurs; maybe you have run out of memory. A dialog box pops up explaining the situation. Clicking OK results in shutting down the application. Although the result may ultimately be the same, a number of possible causes for this problem are detailed in the dialog box. C++ EHS provides a straightforward means of programming such an application, if the overhead is acceptable.

With an embedded system, the situation may be very different; let's consider another example. A microprocessor-based heart pacemaker, programmed in C++, finds a problem. What action can it take? Popping up a dialog box is not an option. Clearly the priority is to return to a stable state so that the operation of the pacemaker may continue. The best option is to simply reset the software (and maybe the hardware) so that it starts up afresh. The fact that an error has occurred possibly could be logged away for future reference.

Although the life-or-death nature of a heart pacemaker is dramatic, the need for a very simple response to runtime errors is common among typical embedded systems: telephone switches, instruments, engine management systems, and so forth.

The obvious way to handle exceptions, if the requirements are simple, is to make use of the `catch(...)` construct. It turns out that the code generated by the compiler for this construct is substantially simpler (and faster) than more selective exception handling. An alternative is to allow exceptions to be thrown but provide no catch blocks at all. Then the `terminate()` function can be called (and effect the reset or whatever).

## 5.6 Looking at Code Size and Performance with C++

*Although C is still widely used, the popularity of C++ for embedded programming has steadily increased. However, there are still frequent objections to the language on the grounds of excess overhead and inefficiency. This article is based upon one written for* NewBits *in 1995 by Nick Lethaby (now with Texas Instruments), in which he addressed these objections. (CW)*

The C++ language has gained rapidly in popularity among embedded developers in recent years and has clearly established itself as the language of choice for developers of larger systems. C is still in widespread use among embedded and real-time developers, but many of these developers are turning to C++ for new projects.

As its popularity has increased over the years, the C++ language has also undergone significant evolution, adding new features such as templates (parameterized types), runtime type identification, and structured exceptions. The language is now fairly stable, with the establishment of an ANSI standard.

### How Efficient Is C++ Compared to C?

While the benefits of C++ and object-oriented programming have been well publicized, very little has been said about the efficiency of C++. Unlike host computer systems that have abundant memory and CPU resources, embedded systems must typically minimize such resources to minimize production costs and/or power consumption. Many embedded programmers are rightly concerned about program size and performance degradation when applications are developed in C++.

C++ is essentially a superset of C and may be used exactly like C. Under these circumstances, applications developed using C++ will incur no more overhead than those written in C. However, although C++ does provide improved type checking over C, along with some other niceties, most developers are attracted to C++ for the promised benefits of object-oriented programming techniques, including improved reusability and maintainability. These benefits require using the many new language features of C++, which may have program size implications, depending upon how they are implemented by the compilation system.

Using object-oriented programming methods with C++ can lead to performance problems unless the underlying implementation of the objects used in the program is well understood. For example, it is important to understand how long it takes to construct or destruct an object. Such an understanding is critical if a programmer is to avoid arbitrarily defining objects with long creation times in performance-critical sections. Proper programming practice is by far the best defense against the adverse effect on performance of poorly designed constructors and destructors.

### How C++ Affects Application Memory Requirements

Several C++ language features, including templates, inline functions, virtual functions, and inheritance, can affect application size. All of these features are widely used by C++ programmers as well as by the ANSI C++ libraries and other commercial C++ class libraries.

### *Templates*

Templates provide an excellent mechanism for sharing source code because they allow you to write just a single implementation of a class or function to handle multiple data types. However, the C++ language definition says nothing about whether templates should share target code. This convention is left to the compilation system and is therefore highly implementation-dependent. A poorly implemented solution can result in significant increases in code size, which would be unimportant in host-based software but could be a disaster for an embedded application.

In some compilation systems, the compiler will simply instantiate a copy of the template function (or class) for each data type invoked within a particular source file (or module). In Figure 5.2, for example, the compilation of each source file will result in its own instantiations of the `(max)` function. The consequences of this approach are obvious: if an application has 500 source files that use the `(max)` function, 1,500 copies of the `(max)` template function are generated. If the linker simply links all of these into the final object, the application will be significantly larger than necessary. In reality, only three copies (actually, variants—they are all composed of different code) of the `(max)` template function are needed: one for `int`, one for `char`, and one for `float`, as shown in Figure 5.3.

### *Inline Functions*

Inline functions can also result in unwanted duplicate code, although the problem is less obvious than with templates. When a programmer defines a function with the `inline` keyword (or defines it inside a class definition, thereby making it inline by default), the expected result is for the function to be inlined everywhere the function is called. This approach improves performance by eliminating the overhead of a function call. Whether or not code size increases depends on the size of the function. If the function body is

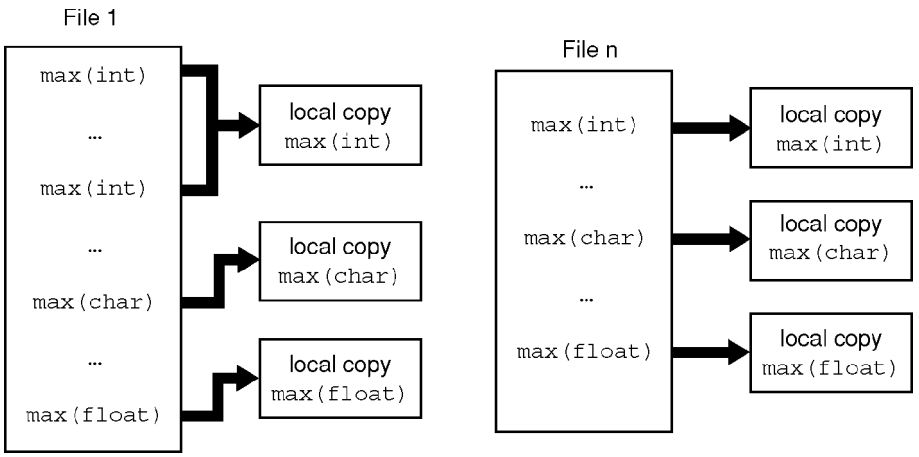



*Figure 5.2: Inefficient template implementation*

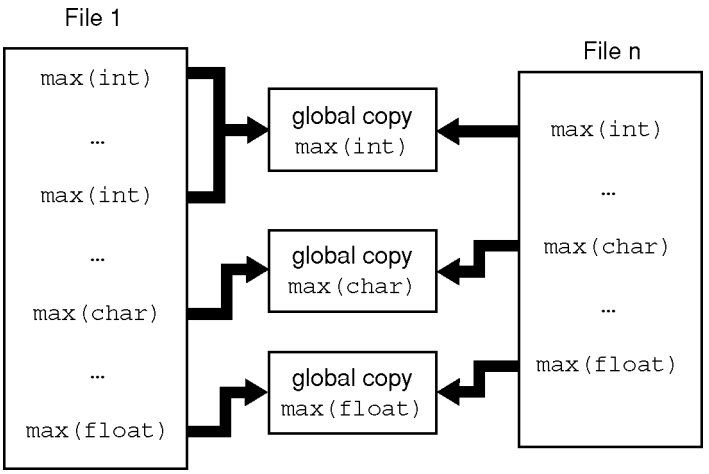



*Figure 5.3: Efficient template implementation*

smaller than the code required to set up and return from a function call, as many inline functions are, code size will decrease. If not, code size will increase.

Unfortunately, there is an awkward twist to this scenario. It is not always possible for the compiler to inline an inline function. Sometimes this is due to poor programming style. For example, a programmer may use an inline function recursively or take its address. In these cases, the compiler treats the inline function as a regular function. However, a more serious problem occurs when the compiler itself cannot inline the function. This is typically caused by a flow-of-control complexity that prevents the optimizer from inlining the function. In these cases, the compiler reverts to treating the function as a regular function.

When an inline function cannot be inlined, the function defaults to being `static`, which means that each file that invokes the function will create its own local copy. Therefore, these "non-inlined inline functions" present the worst of both worlds. No performance increase is realized because the function call is still being made, and code size is increased unnecessarily because a copy of the function exists in every source file that invokes it.

### *Virtual Functions*

The other side of the inefficiency equation is duplicate data. Just as template classes can result in duplicate data, virtual functions, or to be more precise, virtual function tables, can also result in duplicate data.

Virtual functions are called through tables of function pointers known as *virtual function tables*, which are created by the compiler to enable the C++ runtime environment to call the correct function. Inefficiencies can arise, for example, because some compilers create virtual function tables in every file where a virtual function is called, which automatically results in duplicate function tables. A better approach is for the C++ compiler to

create virtual function tables only where they are defined. Some duplication will still occur, however, if the virtual function is an inline or template function. This duplication of virtual function tables further compounds the potential for size inefficiencies.

### *Inheritance*

The layout of inherited objects also contributes to space requirements. Space can be saved by optimizing the layout of objects and virtual function tables. Some C++ toolkits are inefficient in the way that they lay out certain types of inherited objects. Consider the following code fragment for deriving a class called `MDerived` from several classes that are in turn derived from a virtual base class:

```
class BaseClass { ... };

class DerivedA : virtual public BaseClass { ... };
class DerivedB : virtual public BaseClass { ... };
class DerivedC : virtual public BaseClass { ... };
class DerivedD : virtual public BaseClass { ... };

class MDerived : DerivedA, DerivedB, DerivedC, DerivedD { ... };
```

In inefficient toolkits, if you derive a class from *n* classes that in turn are derived from the same virtual base class, the derived class will have at best *n-1* copies (and often *n* copies) of the virtual base class members, as illustrated in Figure 5.4.

An efficient implementation requires only one copy of virtual base class members. Since instances of such a class are created at runtime, this inefficiency has the potential to consume valuable RAM if an application creates thousands of objects containing duplicate virtual base class members.

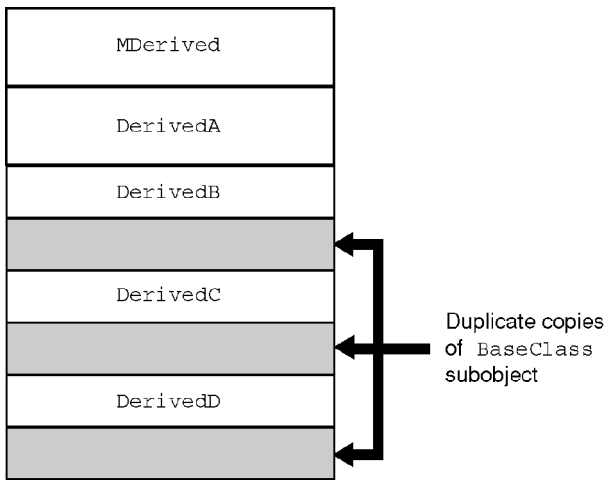



*Figure 5.4: Internal layout of object of type MDerived*

### *Duplicate Debug Information Impacts Productivity*

While minimizing the final application size is important in helping to reduce the cost of the finished product, the problem of duplicate debug information also affects productivity during the development cycle. Although this problem existed with C programming tools, the changes in programming style introduced with C++ has greatly magnified its effect in that C++ applications tend to include every header file in each source file. The complex interrelationships of classes in an application make it difficult to decide exactly which header files are required to successfully compile a certain source file.

In the past, compilers have tended to generate debug information for all the types declared or defined in the header files, regardless of whether the source file actually used them. Obviously, including every header file in each source file can result in vast amounts of duplicated debug information. This extraneous debug information can affect productivity in several ways:

- Link times are much longer because the linker must process all the debug information in each file.
- Debugger initialization times are much longer because all the information must be read by the debugger, which in turn discards any duplicate information.
- Files can become prohibitively large, which impacts disk space, network traffic, and backup procedures.

## Doing C++ Right

Although this article has highlighted a number of potential problems in using C++ for embedded applications, the good news is that there are solutions. Increasingly, these issues are being addressed by C++ toolkits on the market. It is still important to be able to identify the problems, if they persist, and to know what functionality to look for when selecting tools. We will now take a look at just those concerns.

### *Faster Compile and Link Times*

The old way of implementing C++ was to translate it to standard C first. Although this approach was instrumental in the success of the language (compared with other object-oriented languages based upon C), major performance and functional benefits are to be gained from modern, C++ to assembly, compilers.

It is important for a compiler to be discriminating about the debug information that it outputs. The compiler should output only debug information for functions and variables used in the source file rather than for every program construct declared or defined in the header files included by the source file. Typically, this optimization results in a 50% reduction in the final size of an application (absolute file) built with debugging enabled.

The main benefits of this optimization are seen downstream from the compiler. Link times are shortened dramatically because the linker is now processing smaller files. Since linkers tend to be I/O-bound applications, any reduction in size of the files processed automatically translates into shorter link times.

### *Reduced Program Size*

The problems of duplicate code and data must be addressed in the linker as well as in the compiler. This is because the linker is the most practical place to optimize across file boundaries.

Compiler optimizations may address virtual function tables and object layout. Virtual function tables should be created only in files where the function is defined. This practice significantly reduces the problems of duplicate virtual function tables. Improvements to the layout of objects can eliminate the problem of derived objects containing duplicate copies of a virtual base class (see Figure 5.5). This improvement can save significant amounts of RAM in applications with large numbers of objects derived via multiple inheritance and virtual base classes.

The linker may be enhanced to address the issue of duplicate template functions and inline functions, as well as duplicate virtual function tables generated by inline, template, or virtual functions. Solutions involving the compiler tend to lead to considerable increases in compile times.

Linker optimizations that reduce final application memory requirements are performed by special algorithms that merge duplicate copies of code and data. The compiler tags legitimate duplicate copies, such as functions defined as `static` by the user, so the linker knows not to merge them. Algorithms can also be used to merge duplicate debug information.

Linkers tend to be I/O-bound programs. As a result, performing these optimizations does not greatly increase link times because the new linker simply throws away extraneous code and data rather than putting it into a file. In contrast, a compiler-based approach would increase compile times if the compiler had to perform complex cross-file optimizations to remove duplicate code and data.

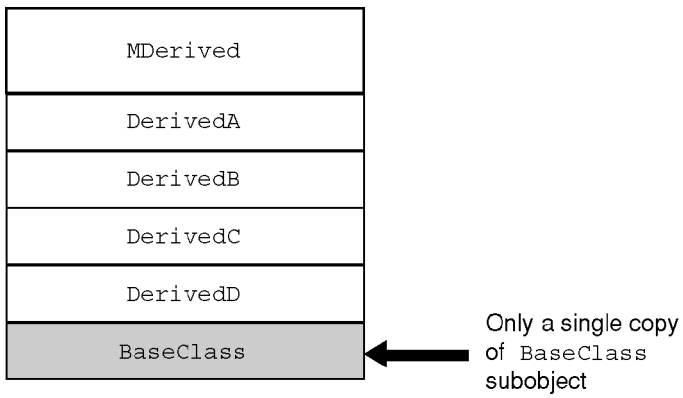



*Figure 5.5: New layout of object of type MDerived*

## Conclusions

As C++ is the standard language used for the implementation of large complex embedded systems, it is important that C++ compilation technology provide compact code and data size to reduce memory requirements and improve compile-link-debug cycle times. Because conventional C optimization technology does not address potential code and data size inefficiencies introduced by some of the C++ object-oriented programming constructs, new optimization techniques need to be implemented to eliminate the unnecessary overhead often associated with C++.

## 5.7 Write-Only Ports in C++

*The debate whether C++ really is a good language for embedded applications has continued for many years. I have given many seminar and conference presentations and conducted numerous training classes on the topic. In 1995, I captured a key example in a piece for* NewBits, *which was the basis for this article. (CW)*

C++ is widely, but not exclusively, used for embedded software programming. C also continues to be popular. Many engineers and managers seek advice upon the choice of development language. Typical questions are "Is C++ suitable for my application?" and "What real benefit does C++ provide for the embedded systems developer?" This article provides guidelines to enable you to answer the first question yourself. In addition, it will offer at least one answer to the second. To work toward this answer, we will define a specific embedded systems programming challenge, consider a solution programmed in C, and investigate how it may be solved in C++. Then we will develop the idea to provide a more general solution. First, however, we need to consider the fundamental philosophy of using C++ in this context.

### Encapsulating Expertise

If you ask any programmers what the benefits of an object-oriented approach are, their responses will most likely focus on the reusability of the code. This is a true benefit because the avoidance of "reinventing the wheel" is important in any computer programming context. However, reusability has no *specific* benefit to embedded systems programming.

The difference between "normal" programming and embedded systems work may be hard to define; it has as much to do with the composition of the development team as the techniques employed. Writing software for a large embedded system ideally requires a good knowledge of software techniques, a reasonable understanding of the hardware, and a thorough appreciation of the real application. It is rare to find an individual who can actually bring all of this expertise to a project team. In general, a team consists of various specialists: software engineers, hardware engineers, and applications experts. It is quite common, for example, for a software engineer to have no understanding at all of the hardware used to implement the system. This leads to difficulties in implementing low-level drivers and other elements.

An advantage of using object-oriented techniques is that the difficult-to-understand driver code may be contained in an object, effectively encapsulating the expertise of the hardware engineer. The software engineer can then use the object quite safely because their access to the internals of the driving code is strictly controlled.

### Defining the Problem

To illustrate this point, we need an example of an embedded system feature that may present difficulties to a software engineer—for example, write-only ports. A write-only port is an output device register (usually memory-mapped), to which data can be written,

but from which data cannot be read. An attempt to read from the device can, at best, yield useless data; at worst it might result in a processor exception. Figure 5.6 illustrates the implementation of a write-only port.

Note that the `Read Enable` signal is simply not connected to the port electronics. Write-only ports are not the work of a deranged hardware designer with a strange sense of humor or who harbors a grudge against software engineers (even though we may all be acquainted with individuals who fit this description). Write-only ports are an inevitable result of rational design; the ability to read from the device requires additional and otherwise useless electronics.

The problem for the software designer is exacerbated by a common feature of write-only ports—they are frequently collections of unrelated output bits. This means that different parts of the software may be concerned with the setting or clearing of different bits on the port. Without care, one routine will set a bit, for example, then another will unwittingly change its state.

The usual solution is to maintain a "snapshot" of the last value written to the port. This "snapshot" is commonly referred to as the *shadow*. It is then an apparently simple matter of updating the shadow and outputting it to the port each time a bit needs to be changed. However, this approach may be beset by a number of difficulties:

- Initialization
- Control
- Distribution of code and data—not encapsulated
- Reentrancy

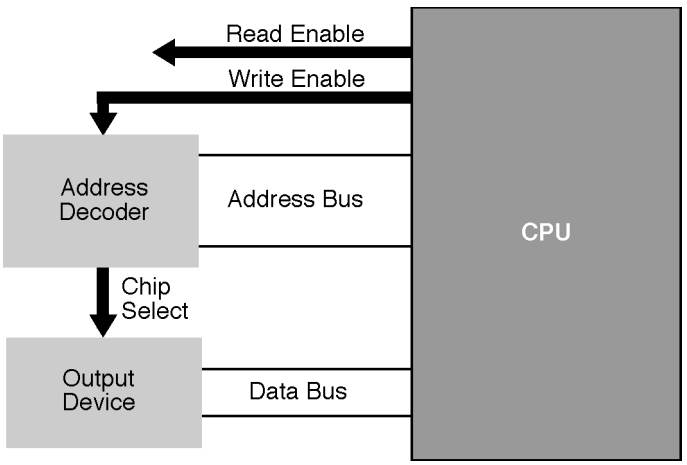



*Figure 5.6: Write-only port implementation*

### A Solution in C

To illustrate these problems, the following example shows a possible implementation of write-only ports in C:

```
extern int ports[10];
int shadows[10];

void wop_set(int port, int bit)
{
    int val;
    val = 1 << bit;
    shadows[port] |= val;
    ports[port] = shadows[port];
}

void wop_clear(int port, int bit)
{
    int val;
    val = ~(1 << bit);
    shadows[port] &= val;
    ports[port] = shadows[port];
}
```

In the program, `ports` is an array that has been mapped onto a series of output ports at consecutive addresses, and `shadows` is an array that contains shadow copies of the ports. Two functions have been provided: `wop_set()` and `wop_clear()`. This code is rather inflexible and does not address the issues of initialization or reentrancy. Along with these shortcomings, the programmer is provided with a very limited means of setting and clearing bits.

### A First Attempt in C++

The first objective in attempting to implement a write-only port object is to simply encapsulate the necessary functionality into a C++ object. The code shown in the following example is a first shot:

```
class wop
{
    int shadow;      // not accessible -
    int* address;    // to user
public:
    wop(long);       // constructor
    ~wop();          // destructor
    void or(int);    // "operator" -
    void and(int);   // functions
};
```

The class `wop` has two private member variables, `shadow` and `address`, which hold the stored port value and the address of the port, respectively. Along with the constructor and destructor, two member functions provide `OR` and `AND` functionality, which is quite sufficient to set and clear bits on the port.

The constructor:

```
wop::wop(long port)
{
    address = (int*) port;
    shadow = 0;
    *address = 0;
}
```

which is called automatically when the object comes into scope, simply initializes the `address` variable, and writes 0 (an arbitrary value) to the port and the shadow copy.

The destructor:

```
wop::~wop()
{
    *address = 0;
}
```

which is called when the object goes out of scope, just resets the port to 0 again.

The `or()` and `and()` member functions:

```
void wop::or(int val)
{
    shadow | = val;      // set bit(s) in copy
    *address = shadow;  // update port

}

void wop::and(int val)
{
    shadow &= val;       // clear bit(s) in copy
    *address = shadow;  // update port
}
```

just take the provided parameter, operate on the shadow data, and write the new value out to the port.

This `main()` function illustrates the creation and use of a `wop` object:

```
main()
{
    wop out(0x10000);

    out.or(0x30); // set bits 4 and 5
    out.and(~7);  // clear bits 0, 1 and 2
}
```

### Using Overloaded Operators

In C++ it is possible to redefine operators to enable them to work with new data types. This concept sounds confusing, but it can lead to a very natural way of using objects if the usual behavior of the operators is carefully retained. The example is modified as follows:

```
class wop
{
    int shadow;
    int* address;
public:
    wop(long);
    ~wop();
    void operator|=(int);      // overloaded
    void operator&=(int);      // operators
};
```

The member variables and the constructor and destructor functions are unchanged. The `or()` and `and()` member functions have been replaced by two operator functions that overload the `|=` and `&=` operators:

```
void wop::operator|=(int val)
{
    shadow |= val;      // set bit(s) in copy
    *address = shadow;  // update port
}

void wop::operator&=(int val)
{
    shadow &= val;      // clear bit(s) in copy
    *address = shadow;  // update port
}
```

It turns out that the code for the two functions is actually unchanged.

A modified `main()` function shows the natural use of the new operators on a `wop` object:

```
main()
{
    wop out(0x10000);

    out |= 0x30; // set bits 4 and 5
    out &= ~7;   // clear bits 0, 1 and 2
}
```

At this point, it would be entirely reasonable to hand over such an object to the applications programmers, who could use it in any way they wished, without having to worry about the write-only nature of the port.

### Enhancing the wop Class

Although the `wop` class defined in the previous section would be useful, it is quite straightforward to add other functionality. For example, the initialization of the port to the value 0 was purely arbitrary; this matter could be put under the control of the programmer. This code shows the necessary modifications:

```
class wop
{
    int shadow;
    int* address;
    int initval; // stored initial value
public:
    wop(long, int);
    ~wop();
    void   operator|=(int);
    void   operator&=(int);
};
```

A second parameter has been added to the constructor to provide the means of setting the initial port value. The parameter is optional; if the programmer omits it, the value 0 is utilized. This retains compatibility with the previous version of the class definition:

```
wop::wop(long port, int init=0)
{
    address = (int*) port;
    initval = init;
    shadow = initval;
    *address = initval;
}
```

Another private member variable, `initval`, has been added to store the initial value of the port. This value needs to be retained because it will be required by the destructor:

```
wop::~wop()
{
    *address = initval;
}
```

A small adjustment to the `main()` function illustrates the use of this enhancement:

```
main()
{
    wop out(0×10000, 0×0f);

    out |= 0×30;
    out &= ~7;
}
```

### Addressing Reentrancy

In the C++ code presented so far, most of the defined problems with the implementation of a write-only port have been addressed: initialization, control, and the encapsulation of code and data. The remaining issue is reentrancy. In many systems, reentrancy would not be a concern. It is a problem only if an interrupt may occur between the updating of the shadow data and the writing to the actual port. This, in turn, is a problem only if the interrupting code makes use of the port.

In general terms, we can make a resource reentrant by providing a locking mechanism on that resource; that is, code that uses the resource locks it, uses it, and unlocks it again. As a first step, we will modify the `wop` class to include dummy `lock()` and `unlock()` functions:

```
class wop
{
    int shadow;
    int* address;
    int initval;
    void lock() { };    // dummy
    void unlock() { };  // functions
public:
    wop(long, int);
    ~wop();
    void   operator|=(int);
    void   operator&=(int);
};
```

These are called by the operator functions to utilize the resource:

```
void wop::operator|=(int val)
{
    lock();
    shadow |= val;
    *address = shadow;
    unlock();
}

void wop::operator&=(int val)
{
    lock();
    shadow &= val;
    *address = shadow;
    unlock();
}
```

Before proceeding, we should consider how useful it would be to make the `wop` object reentrant. This would be an unnecessary overhead in a system where reentrancy is not a problem. So we really need some flexibility. The answer lies in object-oriented programming techniques. We can define a new object, `rwop`, derived from the already existing `wop` object, which imparts reentrancy. In turn, to make this workable, we need to make the dummy `lock()` and `unlock()` functions replaceable; that is, we need them to be virtual functions:

```
class wop
{
    int shadow;
    int* address;
    int initval;
    virtual void lock();       // replaceable
    virtual void unlock();     // functions
public:
    wop(long);
    ~wop();
    void   operator| = (int);
    void   operator&=(int);
};
```

The implementation of the `rwop` object is as shown:

```
class rwop : public wop
{
    int flag;
    void lock()  // replacement functions
    {
        while (flag)
            ;
        flag = 1;
    };
    void unlock()
    {
        flag = 0;
    };
public:
    rwop(long, int); // constructor
};
```

The locking mechanism needs to be initialized by the constructor:

```
rwop::rwop(long port, init init=0) : wop(port, init)
{
    unlock();
}
```

The new `lock()` and `unlock()` functions use a simple token variable, `flag`, to effect the locking. This is a simplistic example and not strictly accurate, but it serves to illustrate the point.

Once again, the applications programmer can use this object with no knowledge at all of the underlying mechanisms. Just using `rwop` instead of `wop`:

```
main()
{
    rwop out(0x10000, 0x0f);

    out |= 0x30;
    out &= ~7;
}
```

### Using an RTOS

Most commonly, reentrancy problems arise as a result of the use of a real-time operating system (RTOS). In this case, it is only logical to make use of the RTOS facilities to control access to the resource. One approach is to use a mailbox to contain a token, which imparts ownership of the resource. In the following example, a new type of object, `vwop`, is defined, which uses RTOS calls to maintain a mailbox and create a reentrant write-only port:

```
class vwop : public wop
{
    int* mbox;
    int err;
    void lock()
    {
        pend(&mbox, 0, &err);
    };
    void unlock()
    {
        post(&mbox, (char*)1, &err);
    };
public:
    vwop(long, int);
};
```

No change is necessary to the `wop` base class. In addition to the new `lock()` and `unlock()` functions, a constructor is required for the `vwop` class to initialize the mailbox:

```
vwop::vwop(long port, init init=0) : wop(port, init)
{
    mbox = (int*)0;
    unlock();
}
```

This last example is a very powerful illustration of the use of C++ for embedded systems development. It uses fairly simple object-oriented techniques, and yet the applications programmer can utilize `vwop` objects very straightforwardly, without any knowledge of write-only ports or the use of an RTOS.

If, at some future point, a different RTOS is selected, the `vwop` class will most likely need modification. However, the `wop` base class and the applications code will not need changing.

### Expertise Encapsulated

We started out with the idea of the encapsulation of expertise. In working through this example, we successfully encapsulated the expertise of the hardware specialist, by hiding the write-only functionality. Then we encapsulated the expertise of the RTOS specialist by the transparent addition of reentrancy protection. At every stage, the applications developer could concentrate on their expertise—the application.

Interestingly, this approach would have been useful, even if the hardware designers had taken a benevolent attitude to the software engineers and implemented a read/write port. Such a device might still need careful handling to avoid reentrancy problems.

### Other Possibilities

Many of the ideas discussed here can be extended further: an `XOR` operator, external access to the shadow, separate initialization and shut-down values, and so on.

Also, the same principles employed in this article could easily apply to other embedded systems programming problems. For example, many serial I/O chips have multiple internal registers that need to be selected before they can be accessed. A suitable class could be designed to hide this detail from the applications programmer and also to protect against errors and reentrancy problems.

### The Way Forward

C++ is very commonly the language of choice for embedded systems development. But many developers are still concerned about whether the benefits outweigh any possible downsides and take an “if it ain’t broke, don’t fix it” approach. Hopefully this article has given them some useful ideas.

The best guideline in the use of the language is the simple suggestion that you make conservative use of the object-oriented techniques. Do not get buried in deep inheritance swamps; concentrate on encapsulating vital expertise.

## 5.8 Using Nonvolatile RAM with C++

*I wrote a piece in* NewBits *in late 1997 to illustrate a practical and useful application of C++ and object-oriented programming for embedded applications. This article is based on that piece. (CW)*

Nonvolatile RAM (NVRAM) is memory that retains data after the power has been removed. It is increasingly common for embedded systems to require storage of data (often configuration parameters) after they have powered down. This necessitates some careful coding to accommodate integrity checking (such memory can be corrupted) and initialization. Various NVRAM technologies are available, each of which may need different handling.

To use NVRAM, a means of reading and writing data in a transparent, secure manner is required. The actual mechanism for accessing the memory and effecting the security should be hidden from the applications programmer. C++ facilitates this by permitting the definition of a class that describes the functionality of an NVRAM variable.

### Requirements of Using Nonvolatile RAM With C++

The problem of using NVRAM with C++ will be addressed from the perspective of the applications programmer. How would someone be comfortable using NVRAM?

This code illustrates the way an applications programmer might like to make use of NVRAM:

```
main()
{
    nvram x(0x10000,10,0 x100);
    nvram y(0x10000,11,0 x100);
    char   a;

    x = 10;
    a = x;
    y = a + x;
}
```

Two NVRAM variables have been defined, `x` and `y`. In both cases, the declaration includes three parameters (which are passed to the constructor): the base address of the NVRAM area, the offset into the area of this particular variable, and the total size of the area. After the definitions, the variables are treated as if they are regular `char` variables.

The preceding code example indicates what is required by the applications programmer from the class definition:

- A constructor that accepts the specified parameters.
- An implementation of the = operator to enable values to be assigned to the `nvram` variable.

- An implementation of an `int` conversion function that permits the value of the `nvram` variable to be extracted.

### NVRAM Implementation

Various NVRAM technologies are available. Some have special access requirements—for example, timing. In all cases, integrity checking (checksum, CRC, etc.) is necessary to ensure that the memory was not corrupted during the power-down period. The memory may be affected by power spikes, or some types are prone to "forget" after a time. It is also a requirement to track whether the memory has been initialized or just contains garbage values, because it will the first time that it is powered up.

In the case study, some assumptions and arbitrary decisions have been made (as illustrated in Figure 5.7):

- The NVRAM is a simple technology, not requiring any special timing.
- The last 6 bytes of the NVRAM area contain a null-terminated signature string: "NVRAM".
- A single-byte checksum is placed just before the signature. This is a simple exclusive-OR of the data bytes.

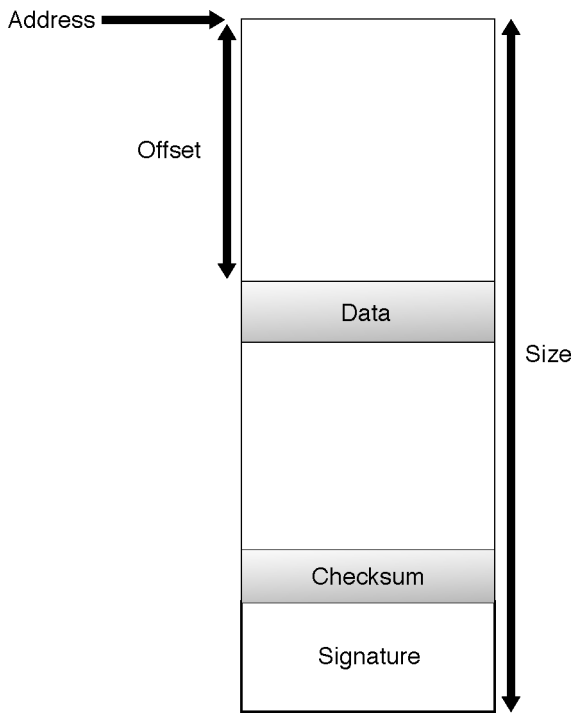



*Figure 5.7: NVRAM architecture*

### A C++ nvram Class

Here is the code for an implementation of a class to handle nonvolatile memory, `nvram`:

```
class nvram
{
    int    check_sig();
    int    eval_checksum();
    char   *address;
    int    size;
    char   *data;
public:
    nvram(long, int, int);
    void operator=(int);
    int operator int();
};
```

The `nvram` class includes two private member functions that check the integrity of the signature string (`check_sig()`) and evaluate the checksum (`eval_checksum()`). There are three private member variables: a pointer to the base of the NVRAM area (`address`), the total size of the area (`size`), and a pointer to the specific location in the NVRAM that is allocated to this object (`data`). There are three public member functions: the constructor, an overloading of the = operator, and an `int` conversion function.

Here is the code for the constructor:

```
#define CHECKSUM (address + size – 7)
#define SIGNATURE (address + size – 6)

nvram::nvram(long memaddr, int offset, int memsize)
{
    address = (char*)memaddr;
    size = memsize;
    data = address + offset;
    if (!check_sig() || *CHECKSUM != eval_checksum())
    {
        for (int i=0; i<size-6; i++)
               *(address + i) = 0;
        strcpy(SIGNATURE, "NVRAM");
    }
}
```

This code sets up the three member variables and then checks the signature and checksum integrity. If either of them indicates that the NVRAM is corrupt, it is initialized to all zeros.

Here are the two utility member functions:

```
int nvram::check_sig()
{
    return !strcmp(SIGNATURE, "NVRAM");
}

int nvram::eval_checksum()
{
    char cs = 0;

    for (int i=0; i<size-7; i++)
        cs ^= *(address + i);
    return cs;
}
```

The `check_sig()` function simply compares the last six bytes with the string "NVRAM", returning `TRUE` if there is a match. The `eval_checksum()` function calculates and returns the checksum value. This is the exclusive-OR of all the bytes of the data area of the NVRAM (i.e., all but the last 7 bytes).

Here is the code for the operator functions:

```
void nvram::operator=(int val)
{
    *data = val;
    *CHECKSUM = eval_checksum();
}

int nvram::operator int()
{
    return *data;
}
```

The = operator overloading function permits a value to be assigned into the `nvram` object. It stores the value into the location in the NVRAM allocated to the object and updates the checksum. This functionality could have been implemented by means of an alternate constructor, with a single `int` parameter. However, doing so would have given the applications programmer the opportunity to attempt to define an `nvram` object erroneously using a constructor call with a single parameter.

The `int` conversion function obtains the data from the `nvram` object and returns it as an `int`. This permits the `nvram` object to be used anywhere in an expression where an `int` would be valid. Why `int` and not `char`? When a `char` is used in an expression,

the rules of C/C++ demand that it be converted to an `int`. So conversion to the `int` type makes more sense here.

Using the previous definition of the `nvram` class, the original application code may be employed.

### Enhancing the nvram Class

Various enhancements and variations upon the `nvram` class implementation may be considered:

- Each element of an NVRAM could have a flag that indicates its validity. This would permit each object's constructor to initialize the element to a specific value, if the flag indicated that the NVRAM had been corrupted.
- A popular NVRAM implementation is flash RAM. This type of memory has specific timing and access requirements, which may be readily incorporated into a C++ class definition.
- An additional public member function could be implemented that enabled the entire NVRAM to be initialized to zeros.
- The `nvram` member functions could be made reentrant by including appropriate locking mechanisms into the operator functions. This would permit their straightforward use with an RTOS.
- With the current implementation, every object definition requires the base address and size of the total NVRAM area, even if only one such area is in existence. It would be more convenient if this information could be provided just once, the relevant data being saved in static member variables. A single object that accepted the base address and size as constructor parameters could be defined initially.

### Conclusions

In C++, objects may be created that encapsulate specific expertise. These objects may be used by application programmers to exploit that expertise without needing to understand it. Further objects may be derived from existing ones that include additional, mutually exclusive expertise. Using object-oriented techniques, the knowledge and experience of a large project team may be segmented, encapsulated, and distributed.

Classes are designed, implemented, and maintained by an "expert." The access and usage by an applications programmer may be restricted to that which is strictly necessary.

Careful design of classes permits straightforward definition and subsequent manipulation of quite complex objects. In the particular case of nonvolatile RAM, the applications programmer may be provided with an entirely intuitive interface, whereas the systems integrator has the freedom to change the underlying nonvolatile RAM technology, as required.

# 6

# *Real Time*

*In my original plan for this book, I had a single chapter allocated to all real-time related articles, whether this included real-time operating system (RTOS) topics or not. I then concluded that I should emphasize the fact that there's more to real-time than RTOSs. So these articles got a chapter to themselves.*

## 6.1 Real-Time Systems

*Back in 1992, when I wrote the original* NewBits *article, upon which this piece is based, there was an increasing feeling among embedded developers that "real-time" meant "real-time operating system" (RTOS). This was an attitude only encouraged by vendors of commercial RTOS products. My intention was to set the record straight. (CW)*

What exactly is a "Real-Time System"? There are those who would say that an embedded system is a real-time system (RTS). I have a little trouble with that statement, since a definition of the term "real-time" is rather hard to come by. That seems to be the way with computer jargon. There are many terms, which we use quite freely (and accurately), but for which we cannot offer a precise definition.

Looking up "real-time system" in a rather old computer dictionary yields: "Any system in which the processing of data input to the system to obtain a result occurs virtually simultaneously with the event generating that data." It cites airline booking systems being an example. Clearly this is not an adequate definition of what most embedded systems do. If you consider typical embedded systems, they are generally performing supervisory, control, or data acquisition functions. Such processing is considered "real-time" because it happens as and when required, not when the system "gets around to it." It also has very little to do with being fast, just being on time. A system that gathers some data once an hour is just as much "real time" as one which reads sensors every ten milliseconds.

### Implementing an RTS

Having arrived at some idea, if not a complete definition, of what an RTS is, we can now look at the options available when implementing one. There are, very broadly, four ways to implement an RTS:

1. A simple processing loop.
2. A background processing loop, with interrupt service code.
3. A multitasking system using a scheduler.
4. A multitasking system using an RTOS.

### A Processing Loop

For a small system of low complexity, the simplest (and, therefore, by definition, the best) approach is a closed processing loop. Each part of the code is processed in sequence endlessly. The loop needs to be infinite, of course, since, in general, an embedded system does not have a logical state which corresponds to the code having stopped.

Although simple, this approach is not without challenges. Typically, the problem is ensuring that particular parts of the code are run often enough which, in turn, depends upon the execution time of other parts of the code. Without care, this method can lead to serious debugging problems especially with timing, which are only located when the code is run on the final hardware. Facilities such as the cycles count of instruction set simulator debuggers may be invaluable here. The worst case execution time of sections of code can be determined long before real hardware is available.

A more sophisticated approach is needed when an application becomes so complex that timing requirements result in certain code sequences being called several times in different parts of the loop. This immediately suggests that other implementation techniques such as those described next are necessary.

### Interrupts

Using interrupts is the next most complex approach to designing an RTS. Typically, there are two circumstances where the use of interrupts may be appropriate: if the servicing of a device is not particularly regular, but must be timely (e.g., a communications application), or when regular processing is required under the control of a real-time clock.

Such a system usually still has a background continuous processing loop, which is interrupted to perform the more urgent work. With a number of microcontrollers, it may be better (from a power consumption viewpoint) to halt the CPU most of the time and perform all the processing in interrupt service routines.

Since high-level languages such as C are normally used to implement modern embedded systems, the ability to write interrupt service routines straightforwardly in C is clearly a requirement. Most modern C compilers, intended for embedded applications, support an additional keyword, `interrupt`, to accommodate this need. (Another article in this book, "Interrupt Functions and ANSI Keywords," covers this topic in more detail.)

An application which uses a number of interrupts (particularly those from a real-time clock) to trigger sizeable sequences of code, can be implemented better using a multitasking approach.

### Multitasking

The concept of multitasking is really quite straightforward: a number of programs (we will call them "tasks") appear to be running on the same CPU simultaneously. This is

achieved by allowing each task to run for a short while and then interrupting it, saving its machine registers' contents, loading the registers with the data for the next task, and continuing processing. The stimulus for the task swap is possibly a real-time clock interrupt; this is called *time-slicing*.

Writing a scheduler to implement time-sliced multitasking is not, in general, very difficult and may be well worthwhile if it provides a simple means of dividing up the CPU's power in a predictable manner. A further consideration is the use of software team members. Since each task can be quite independent from the rest, assigning different tasks to each team member can be an efficient way to divide up the work.

If the multitasking application becomes too complex (e.g., multiple task priorities, mailbox communication, etc.), you should consider the purchase of a commercial real-time operating system (RTOS). (Another article in this book, "Bring in the Pros—When to Consider a Commercial RTOS," looks at this selection process in more detail.)

Implementing an application using multitasking has distinct challenges associated with it. A compiler must be capable of generating reentrant (i.e., sharable) code, as it is very common for functions to be shared between many tasks. Unlike "real" computer operating systems, it is uneconomic for each task to have a complete, private copy of all the code. The programmer is also responsible for generating reentrant codes. Only automatic (and register) variables will be replicated for each instance of the code; the static storage of data must be avoided. Also, any library functions that are called by shared code must themselves be reentrant. Lastly, to keep track of the current task, a machine register may need to be reserved for use by the scheduler.

Reentrant code generation, reentrant libraries (within the bounds of ANSI compliance), and register reserving are all supported by most embedded C compilers.

The debugging of a system using an in-house implemented task scheduler has its own set of challenges. Primarily, these difficulties relate to task interaction and the tracking of shared code. Since a conventional debugger has no "knowledge" of the scheduler, it cannot assist with debugging any aspect of task control or interaction. Furthermore, even something as simple as an execution breakpoint is a problem, if it is set on some shared code. When the breakpoint is hit, the code could have been executing in the context of any task and may not be the task currently being debugged. (Further articles in this book, "Debugging Techniques with an RTOS" and "A Debugging Solution for a Custom Real-Time Operating System" address this topic in more detail.)

### Using an RTOS

For a more demanding application, where task scheduling and interaction are very complex, a commercial real-time operating system (RTOS) is needed to marshal the power of a modern high-end microprocessor. Such systems typically provide multipriority task scheduling, sophisticated intertask communications, dynamic task creation, and deletion

and memory allocation. All of this is performed in a manner which is appropriately fast and predictable for a real-time application.

An RTOS package is likely to include a number of optional components beyond the kernel, such as protocol stacks, file systems, graphics, etc. Also, a debugger, tailored to the needs of the RTOS, is an essential.

## 6.2 Visualizing Program Models of Embedded Systems

*This article, based upon one by Richard Vlamynck (now with Nebula E.D.A.) in* NewBits *in 1996, takes another look at the core features of real-time systems. The use of the term "model" is interesting, as this term gets used in a variety of ways. Here it refers to an approach to system implementation. Richard uses a novel "question and answer" format, which I have left unchanged. (CW)*

### Which programming model is best for building real-time systems?

The answer depends on the kind of system being built. There are two basic programming models for building real-time systems, each with its advantages and disadvantages. The first model views the real-time application as a single thread of execution, while the second model views the application as multiple threads of execution.

### What purpose do models serve for a real-time system?

A model is a preliminary work or construction that serves as a plan from which an end product is drawn; models aid in testing and perfecting the final product. More importantly, the model describes the system, accounts for its known properties, and can be used for further study of its characteristics. The real-time engineer uses program models to develop software and hardware tuned to the real-time system under consideration. A model allows the engineer to predict and meet performance constraints and functionality objectives.

### What differences exist between models, and what gains require what sacrifices?

Some programming models allow the engineer to write programs more easily, but such programs are harder to debug. In other cases, the program can be harder to write, but the system is easier to debug. Some models allow the programs to run faster, but at the cost of consuming more memory resources. Still others are more robust and require less maintenance.

### What is the single-threaded programming model?

The single-threaded programming model has many uses for building small real-time systems. Write a `main()` function that calls other functions and does work. The program executes instructions one at a time—a "single thread of execution" that can be traced through the program.

### What are the advantages and disadvantages of the single-threaded programming model?

Single-threaded models are usually fast and simple to program and reprogram. New functions plug in easily, changing the nature of the machine's response. The disadvantages of a single-threaded program lie in its restricted usefulness in limited application domains; it is not easily reprogrammed "on the fly." This makes it difficult to adapt a running system to a different behavior or a different environment.

### Is a polling loop a single-threaded program?

Yes (see Figure 6.1). Visualize the function `main()` coded as an endless loop, running forever in an embedded system. This programming model applies to certain classes of real-time systems. Consider a simple petrochemical blender that opens or closes valves in response to some other control program. In such a case, a simple polling loop sufficiently implements the system. A polling loop requires little effort to write, but demands resources to maintain if the system gets too complicated. Accurate timing predictions based on the model of a polling loop are difficult for embedded systems.

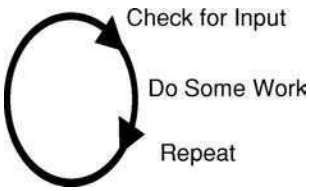



*Figure 6.1: Polling loop visualization*

### Is a state machine a single-threaded program?

Yes (see Figure 6.2). A state machine, though single threaded, has a greater complexity. A state machine is generally a "Moore machine" or a "Mealy machine"; either is programmable in software. This programming model yields a real-time system that is more difficult to write than a polling loop but that is usually easier to maintain as the system gets more complex.

### What is a multi-threaded system?

Visualize a multi-threaded system as a multiple number of polling loops or state machines, all potentially running at the same time (see Figure 6.3). A multi-threaded system advantageously lets the embedded systems engineer break up a system in different ways than would be possible with a single-threaded programming model. An engineer can conceptualize parallel rather than linear flows of work. The engineer writes multiple programs that can all potentially run at the same time. Each program has an entry point similar to the one that `main()` possesses in a single-threaded program. Each program can be coded as an endless loop or state machine or merely work and then exit.

### What are the advantages and disadvantages of the multi-threaded programming model?

The multi-threaded model allows the engineer to divide the system's work into logical sections, and then write an independent program to process that work. All programs potentially make progress in parallel. The engineer can introduce new models of communication and cooperation among the program tasks due to higher throughput. This must be done with care, as race conditions might be introduced.

### Can multiple threads of execution really run simultaneously on one CPU?

No. All threads appear to execute simultaneously, but at the microsecond level each program runs for an instant only, alternating with some other program which runs for an instant, and so on (see Figure 6.4). All programs make some progress through their code.

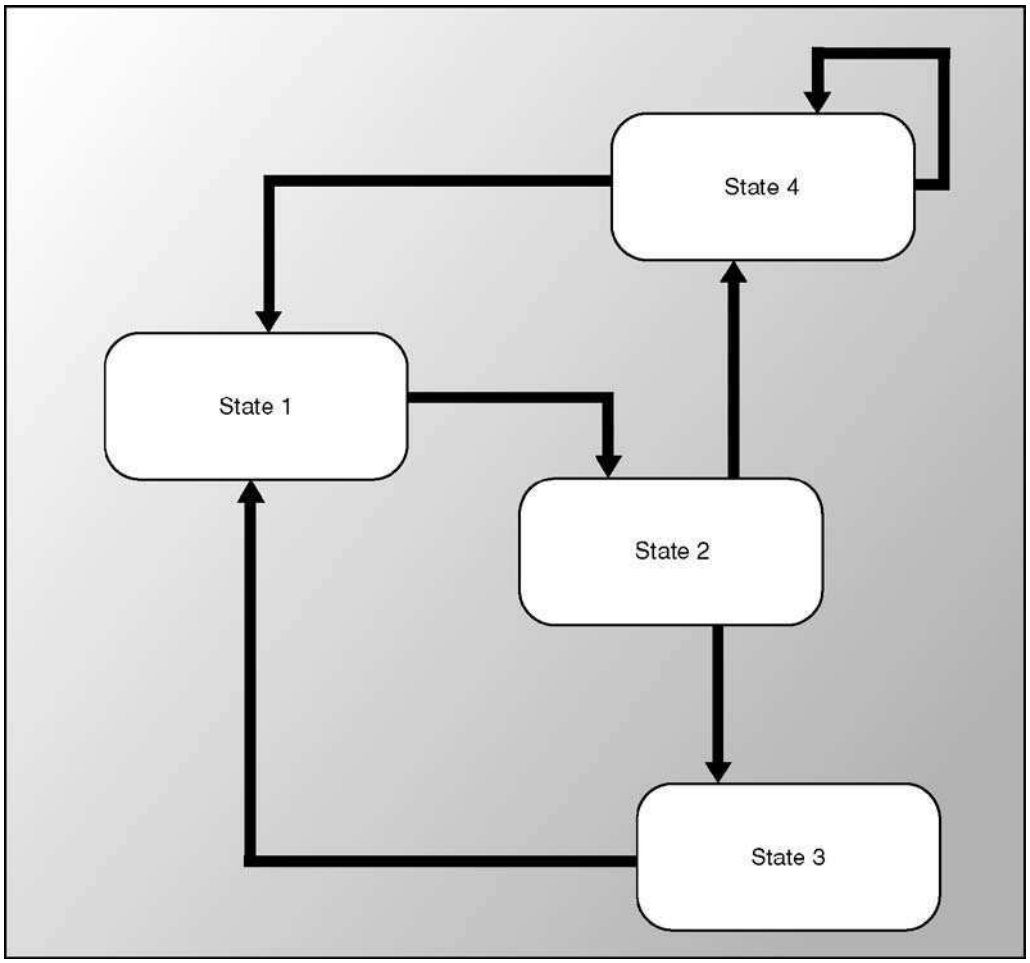



*Figure 6.2: State machine visualization*

### How can a multithreaded environment for a real-time system be acquired?

Broadly, you have a choice between implementing it yourself and obtaining a commercial RTOS. Take a look at the articles in Chapter 7 of this book where you will find plenty of guidance in making this choice.

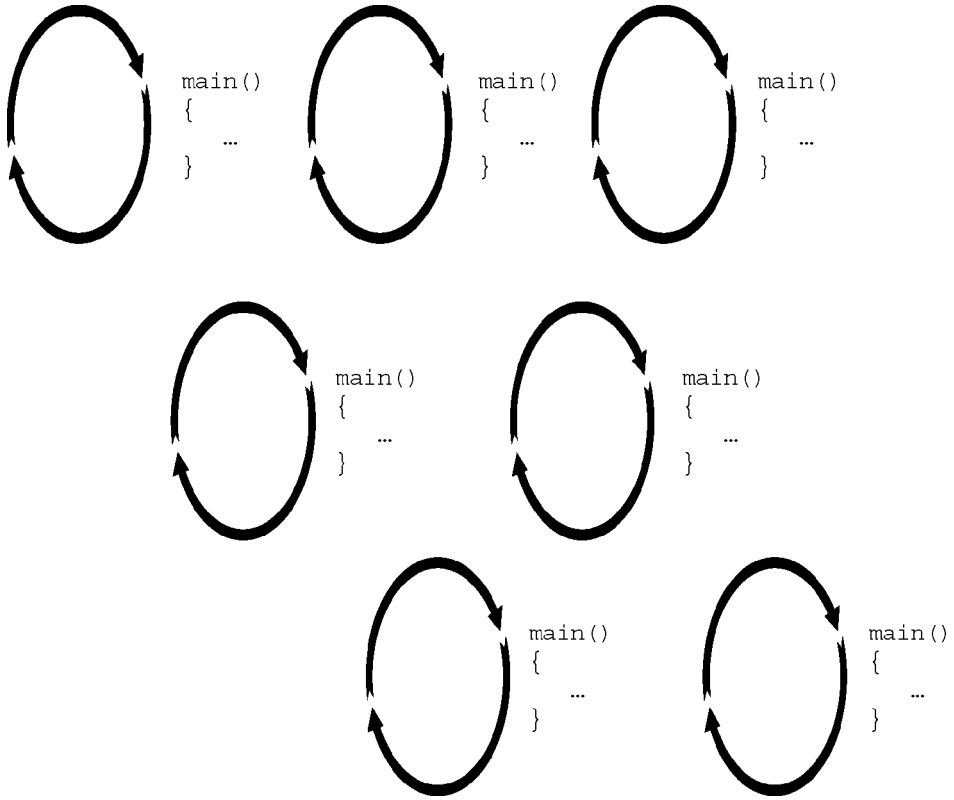



*Figure 6.3: Multitasking visualization #1: many tasks, all with the potential to run at once*

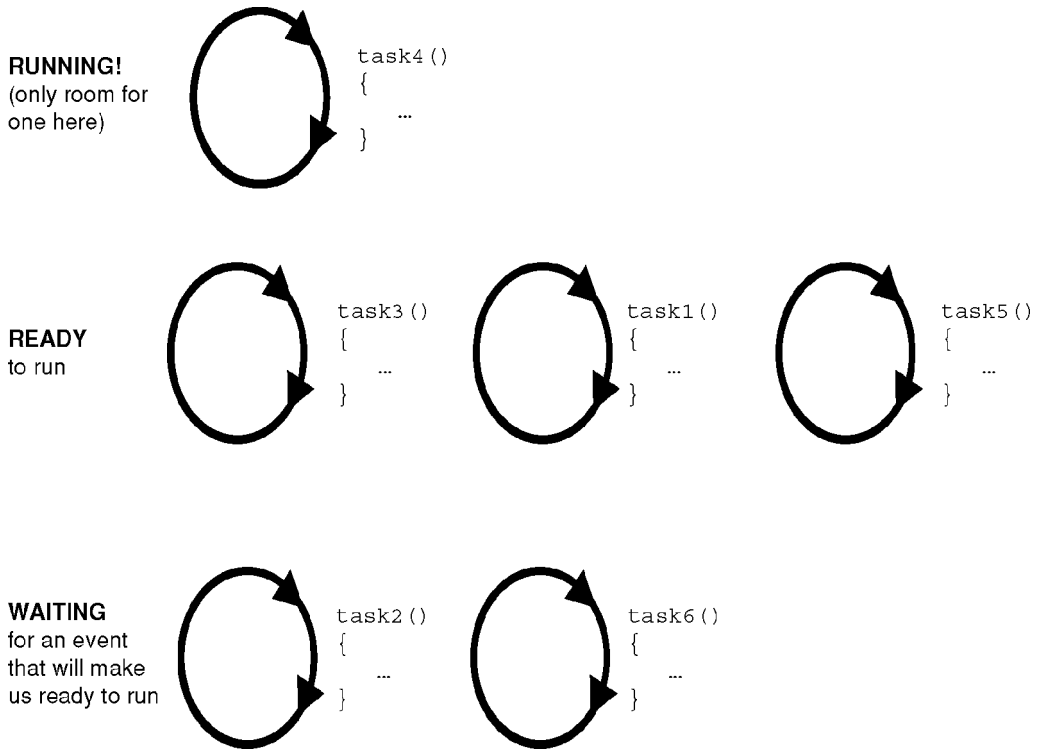



*Figure 6.4: Multitasking visualization #2: one task running and others waiting for an event*

## 6.3 Event Handling in Embedded Systems

*In this article, based upon a piece by Richard Vlamynck (now with Nebula E.D.A.) in* NewBits *in 1996, we will look at the basics of a real-time embedded system. Again, we see Richard's interesting "question and answer" style. (CW)*

### What is an event?

An event is anything that needs the attention of the embedded system. Embedded systems exist to service events. When an event occurs in the real world, an embedded system must recognize the event and take some action in a timely fashion. Embedded systems are often called *event-driven* systems.

Consider if a thermocouple needs to input an updated temperature to the embedded system, or perhaps a transducer must interrupt and let the embedded system know that some pressure vessel has reached an upper or lower limit. At a higher level, an event may signal the fact that some software function has made an error, and an exception needs to be raised.

### Is a signal the same thing as an event?

In a general sense, the words "event," "interrupt," "signal," and "exception" all refer to the same thing: something important happened and we have to take care of it. At a high level of abstraction, a UNIX software signal is the same thing as a Motorola 68040 hardware interrupt: something happened "out there," and our system needs to respond to that time-critical event.

### What events are the most time critical?

Usually, the most time-critical events in real-time embedded systems are associated with exceptions. In this case, we will be less abstract and more specific and take the meaning of the word "exception" to mean only those events that are handled directly by the CPU (microprocessor) hardware.

### What does the microprocessor do when it detects an exception?

When an exception occurs, the microprocessor hardware itself stops the program thread that was running and starts running a different program thread. Generally, the microprocessor hardware selects the new program thread from an array of pointers to "exception handlers." This exception handler table is located by a pointer in a special register in the microprocessor. Because the response to an exception is mostly hardwired by the microprocessor itself, that response can occur very quickly, usually on the order of a few microseconds. This timely response is exactly what the embedded systems engineer wants.

### Are all exceptions the same?

Exceptions are broadly classed as synchronous or asynchronous exceptions. From the point of view of the embedded control program, an exception that is synchronous

occurs as a result of something that it did, and an exception that is asynchronous occurs as a result of something that someone else did.

### What is a synchronous exception?

A synchronous exception occurs when the program thread that is running either causes some error to occur or deliberately causes a trap. The two most classic program errors are when a thread tries to calculate the results of a bad arithmetic expression, causing a divide by zero exception, and when a thread de-references a bad pointer, causing a bus error exception. In each case, the microprocessor stops that thread and vectors to a new thread that will service the exception. Some software systems provide the means to trap to system services; in this case, a program thread will execute a machine instruction that deliberately causes a synchronous exception associated with an exception handler which will invoke some system service.

### What is an asynchronous exception?

An asynchronous exception occurs when the program thread that is running gets interrupted by some external event. That is why an asynchronous exception is usually referred to as an *interrupt*. A classic example of an interrupt is when data arrives on some input/output (I/O) channel. The program thread has no idea when data may be received or transmitted from any number of I/O channels, so the embedded control program (program thread) just runs along doing its work and depends upon the interrupt mechanism to service events that happen on an unpredictable basis.

### How do interrupts get generated and serviced?

Most microprocessors have a special input pin or pins which are used by external hardware to signal the microprocessor that some external event needs attention. These are the interrupt pins. For example, when a data packet arrives on a network device such as an Ethernet chip, the device usually does some pre-processing of the packet and then applies a signal to an interrupt pin on the microprocessor to get its attention and interrupt whatever software is currently running. The microprocessor then saves the program thread that was running and vectors to the interrupt handler which is associated with the interrupt that was generated by the Ethernet chip. The Ethernet interrupt service routine runs to completion, doing whatever work needs to be done to take care of the arrival of the packet from the network. When the interrupt service routine is finished, it executes a special instruction which signals the microprocessor that it wants to return from exception. The processor-dependent return instruction will perform some hardwired machine instructions, which will return control flow back to some other (usually non-interrupt handler) program thread.

### What state gets saved by the CPU?

When we say that the microprocessor saves the program thread that was running, we mean that it saves only enough machine state in order to start running that thread again later. For example, on the CPU32 series of the 68K family of microprocessors, the

machine state that gets saved by the microprocessor interrupt vector service algorithm includes the User Stack Pointer, registers D0 through D7, registers A0 through A6, SR, PC, and a Format/ID word. It pushes this data onto the supervisor stack.

### Is the machine state the same as the thread state?

The machine state that is associated with a thread could be much less than the full state of that thread. For instance, the machine state saved by the microprocessor interrupt mechanism may be only the general CPU registers and the program counter, but the full state of a thread (in a multi-threaded environment) may also include things like the thread-ID number, its priority, and the ID numbers of the system resources owned by that thread. This extra thread state may or may not have to be saved during exception handling, depending upon how well the kernel or operating system was optimized for a particular microprocessor.

We should now understand basically what happens when an exception fires: the hardware saves some thread state and vectors us to an exception handler. We now need to address the two related issues of how to implement the exception handler and how much work it should do. Today, an embedded systems engineer should have the choice of low-, intermediate-, and high-level languages, which are assembly, C, and C++, respectively.

### Should I write exception handlers in assembly language or in C?

Most exception handlers are implemented in either assembly or C language, for reasons of expediency. This brings up the related topic of how much work should be done in the exception handler. Interrupts need to be handled very quickly, and you never know how soon you will get the next interrupt, so the less work you do in the interrupt service routine, the better off you are.

### How do I avoid doing work in the exception handler?

One way of avoiding work in the interrupt service routine (or the synchronous exception handler) is to acknowledge the event that caused the exception by servicing the interrupting hardware device so that it squelches the interrupt signal and then defers all the rest of the work associated with that interrupt to a "higher" level of software. In the Ethernet example above, this could mean sending the freshly input data to a task for processing.

What this means in the world of an RTOS is that an interrupt or synchronous exception handler can post an event to a system object and wake up a high-priority task. Thus, a system object (such as a mailbox, queue, or event flag) can be used to associate an exception with a particular task. An interrupt handler can get in and out very fast because of the services provided by an RTOS.

## 6.4 Programming for Interrupts

*Interrupts are an aspect of programming which may not be familiar to every software engineer, but embedded developers need at least a basic understanding of the concepts. This article is based upon one of the "Byte Site" pieces that I did for* New Electronics *magazine in 1996. (CW)*

Most embedded systems need to work in real time; they need to respond to external events in a timely fashion. There are a number of ways that such a system may be implemented:

1. The code may be written as a large loop, which continuously goes around checking inputs and setting outputs. For a simple system, this is by far the best approach. For a more complex requirement, it may be less appropriate, as the code could become less readable and maintainable and the servicing time for inputs may be extended.
2. The program may be implemented as two loops, with control being exchanged between them (co-routines). This yields more readable, more responsive code at the expense of initial programming complexity. This approach is a first step toward the implementation of a multi-tasking scheme.
3. A multi-tasking operating system may be used, which permits the code to be divided into a number of discrete programs that appear to run concurrently. This model is advantageous for more complex systems, where the functionality may be readily divided up into distinct processes. This structure can also ease the division of programming labor across a team of engineers.
4. Non-time-critical code may be placed in a loop, with the input/output response managed by interrupts.

In the remainder of this article, we will look at the last of these options and consider the programming requirements for handling interrupts.

For some devices, interrupts are referred to as "exceptions." In general, these terms may be considered to be synonymous.

### Setting up Interrupts

Given that the hardware has been correctly configured to trigger an appropriate interrupt upon detection of an external event, three aspects of interrupt handling software must be considered:

- Interrupt service routines.
- The interrupt vector.
- Initialization and enabling of interrupts.

### Interrupt Service Routines

The code executed as a result of an interrupt being serviced is called an "interrupt service routine" (ISR). This routine is similar to a normal subroutine (or C function), except for two special requirements:

- An ISR must save the "context" (contents of CPU registers, etc.) of the code which was running when the interrupt occurred and restore it when it has finished.
- At the end of an ISR, instead of the usual "return from subroutine" instruction, a special "return from interrupt" instruction is commonly required.

These special requirements, together with the need to maximize execution performance of an ISR, have resulted in it being common to write the code in assembler. Nowadays, however, the quality of code generated by compilers is perfectly adequate for use in most ISRs. Additionally, compilers often implement the `interrupt` modifier (as an extension to ANSI C), which enables a function to be compiled as an ISR.

Here is a simple example:

```
interrupt void alpha(void)
{
      indata = device_data;
      device_control = 0x80;
      ei();
}
```

The function `alpha()` is implemented as a very simple ISR, which just reads from a data register, writes to a control register, and returns (reenabling interrupts). The "function" `ei()` may actually be a `#define` macro, which expands into an assembler insert; this minimizes runtime and stack overhead. Note that `alpha()` is a `void` function, as it has no means to return a value and has a null parameter list, as it cannot receive parameters from anywhere.

### Interrupt Vector

Most microprocessors support multiple interrupts; the 68000 family, for example, supports up to 256 types of interrupt. The usual way that these are managed is to have a table of addresses of the ISRs, the interrupt (or exception) vector.

Traditionally, all coding that appertains to interrupts has been performed in assembler. However, this is rarely necessary and a high level language (C or C++) may generally be used. The interrupt vector may be coded in C as an array of pointers to functions, thus:

```
void (*interrupt_vector[])() =
{
      alpha,
      beta,
      gamma
};
```

This interrupt vector has three entries, which point to three interrupt service routines, implemented as C functions: `alpha()`, `beta()`, and `gamma()`. All that is necessary is to ensure that the array `interrupt_vector` is located at the appropriate address, using the linker. Although the C syntax for pointers to functions is arcane, this code is easier to read and maintain than its assembly language equivalent.

### Initialization

It is frequently necessary to initialize I/O devices to function in "interrupt mode"; the precise requirements vary from one device to another. This process is normally just a matter of writing a few values into control registers, which may be readily accomplished in C.

Most microprocessors also require interrupts to be enabled by executing an "enable interrupts" instruction. In C, this may be achieved by calling a library function, or, better, by including some inline assembler code (e.g., `asm("EI");`), if this facility is supported by the compiler.

### Conclusions

Interrupts are an important facet of a real-time program. Programming to accommodate them need not be a complex task, being readily accomplished in C using modern development tools.

# 7

# *Real-Time Operating Systems*

*There is nothing mandatory about using an RTOS, but most larger embedded systems tend to include a kernel or OS of some kind. The topic could easily fill a book by itself. The articles in this chapter look at various aspects of using, choosing, or implementing an RTOS and take glimpses at the internal workings.*

## 7.1 Debugging Techniques with an RTOS

*Debugging is a very broad subject. Typically, embedded software engineers appreciate only a small subset of the gamut of possibilities. In particular, debugging with a real-time operating system presents a wide range of options. I have made presentations on this topic at numerous seminars and conferences and authored many papers and articles. A piece that I did in* NewBits *in early 2003 provided the basis for this expanded article. (CW)*

### Introduction

Most modern embedded system designs make use of an RTOS. This permits the software designer to employ the multi-task paradigm to distribute the available processor resources across the required functionality. At the same time, the opportunity arises to distribute the detailed design, programming, and debugging effort across a team.

An RTOS may be developed in-house or it may be a commercial product, licensed for use in the design at hand. In either case, the multi-tasking environment introduces some new challenges. Not the least of these is debugging.

This article aims to provide a complete overview of the issues, scope, and limitations of debugging when an RTOS is in use.

The term *task* is used throughout. This may variously be translated into *thread*, *process,* or *program*, depending upon the specific RTOS in use. The precise semantics are, for the most part, not relevant in this context.

### Multiprocess Concept

The key function of an RTOS is to allocate time to each task in the system in a rational way. There are a variety of mechanisms whereby this is achieved. The simplest is a round-robin, where each task simply passes control on to the next. The next possibility is some kind of time slicing, where the RTOS allocates an equal-sized period of time to each task. Commonly, an RTOS has a much more complex facility, where each task has a priority and is allocated resources accordingly.

The perception of this allocation of processor time is important in the use and design of debugging solutions.

### *Apparent Simultaneity*

Superficially, it would seem logical to keep in mind the mechanism by which the RTOS works: quickly swapping between tasks, so that they give the appearance of executing simultaneously (see Figure 7.1). After all, this is a reflection of what is really happening. However, this model does not prove to be very useful. With the possible exception of simplistic round-robin schedulers, it is never really possible to predict which task will run next. So a programmer cannot profit from this model.


*Figure 7.1: Apparent simultaneity*

A debugging solution, built with this concept in mind, is likely to be "task-aware"—primarily focused on the current task and the need to isolate it from the rest of the system. This is rather inflexible.

### *True Simultaneity*

Even though it is not a true reflection of reality, it proves much more useful to conceive of a model where all the tasks are running simultaneously (see Figure 7.2). This is what most programmers practice because it is the only way to write secure code with proper management of shared resources.

A debugging solution, designed with this mind-set, lends itself to multi-task debugging—viewing a number of tasks and their interactions—which is much more flexible. Furthermore, the model is scalable upwards to a system that contains multiple processors, which really do execute concurrently.


*Figure 7.2: True simultaneity*

## Execution Environment

The conventional, and only really viable, configuration for debugging an embedded system with an RTOS, is to connect a host computer (PC or UNIX workstation, running the debugger) to the target device. The target may be real hardware or perhaps a host-based simulation.

### *Simulation*

A simulation of the target device, running on the host computer, may be useful in a number of circumstances:

- Before the hardware is ready
- Before the hardware is reliable
- Before the hardware is available in volume

Two types of simulation are available when debugging with an RTOS:

- **Instruction set simulation:** Where the simulator accepts the real code, built for a real target, and simulates its execution instruction by instruction. The execution speed is relatively slow, but it is a very precise environment with great opportunities for nonintrusive "instrumentation" of the code (i.e., monitoring the detailed execution behavior of the code, without actually changing it). Although slower than real time, the temporal model may be accurate and has the unique possibility for "stopping real time." Since the entire functionality of the processor is simulated (perhaps along with some of the surrounding hardware), there is no reason why a complete system—RTOS, drivers, and application code—cannot be run.
- **Native system execution:** Where the code is built to actually run on the host computer. This results in a very good execution speed—probably similar to or faster than the real target hardware would be—but with a different real-time profile. A couple of alternative approaches may be taken to facilitate host execution of the target code: a special version of the RTOS runs as a host process; alternatively, the RTOS calls may be mapped onto calls to the host OS.

#### *Hardware Available*

At some point, real hardware becomes available and/or limitations of simulation are reached. The hardware may be:

- Prototype hardware
- Production hardware
- Evaluation boards (similar architecture to final target)

Although apparently better than simulation, using actual hardware is not without problems—hardware glitches and faults and visibility of execution are among them. So, even when hardware is at hand, simulation may remain useful to address certain types of bugs.

If target hardware is to be used, a suitable connection to the host computer must be established.

### Target Connection

In broad terms, there are two types of host/target connection that may be employed for debugging:

- **Dedicated debugging connection:** The link has no additional function, other than debugging.
- **Communications link:** May be used for other purposes as well as debugging.

In some cases, both may be useful.

### Dedicated Debugging Connection

The use of a host-target connection, which is dedicated to debugging, is becoming quite common. Typically, this is based upon the JTAG standard, which, while never intended for this purpose, has been found to be ideal for many such applications. Prior to JTAG becoming common, Motorola (now Freescale) introduced Background Debug Mode (BDM), which worked in a similar way. Almost all new processor designs incorporate a JTAG interface for debugging.

A JTAG connection is quite cheap and easy to incorporate into a design. There are only a small number of signals—it is a synchronous serial protocol—and low-cost connectors are used. Some kind of adapter is required to interface to the host computer. This may be a simple "intelligent cable," costing $100–$1,000; it may be an interface box, which includes other functionality (e.g., trace), costing $1,000–$10,000.

A major advantage of this type of debugging connection is that it will even function with a target that is barely working. So long as the processor has power and a valid clock, a JTAG connection may normally be established. Of course, little progress may be made until the memory system is also functional.

Typically, a target processor is "frozen" (i.e., not executing code) while it is communicating with the host over a JTAG connection.

### Communications Link

Historically, the use of a conventional communications link for debugging was very common. Generally this would be a serial line (RS232, RS422) or Ethernet. However, there are many other viable possibilities: USB, PCI Bus, Bluetooth, IRDA, and 802.11.

A key, but obvious, requirement is a working target. The hardware must be fully functional—the processor must be capable of executing code and the communications interfaces working. The software must, of course, be working to the extent that the communications protocols can be supported. This leads to an interesting "chicken and egg" issue: if this is the debugging interface, how can the drivers and protocol stacks be debugged? The answer is a combination of simulation and the use of a dedicated debugging connection, which is ideal for such "low-level" debugging.

This type of RTOS-aware debugging is necessarily more complex. This need not be a great concern to the user of a commercial RTOS product but would probably discourage the user of an in-house kernel.

If the communications link is required for the application, sharing it for debugging purposes may be very attractive. Otherwise, the overhead of the additional hardware and software may be disproportionate to the benefits gained.

The big advantage of RTOS-aware debugging over such a link is flexibility. The target may continue to run while the debugger communicates with it. This is called *run mode debugging.*

## Debugging Modes

There are broadly two modes of RTOS-aware debugging: stop mode and run mode. Stop mode is almost always an option. The availability of run mode (as well) may be advantageous.

### *Stop Mode*

This mode, which is sometimes also called *freeze mode,* occurs when the entire system is stopped—when no code executes at all—when a breakpoint is hit or the operator intervenes.

Stop mode is a totally host-resident debugging implementation, with no overheads on the target at all. Typically, the target connection is JTAG.

Stop-mode debugging is quite satisfactory for most situations and is only problematic when:

- The bug being sought is closely tied in with the dynamic interaction between tasks.
- Halting the system, as a whole, would cause disadvantageous side effects (that may hamper locating the bug).

This debugging mode is ideal for use with an in-house RTOS because no target support is required, and the implementation may be quite straightforward.

With simulation, stop-mode debugging is sufficient for any conceivable multi-tasking debugging scenario.

### *Run Mode*

Run mode is a more sophisticated RTOS-aware debugging setup because it allows just parts of a system to be stopped; for example:

- Just the current task stops (others continue execution).
- A defined group of tasks stop (others continue execution).
- All tasks stop (but interrupts are still serviced).

This facilitates some subtle debugging, which may not be possible with stop mode.

Run mode needs sophisticated software support on the target—a debugging monitor. This is most likely to preclude its use with an in-house RTOS. A memory overhead is also incurred.

Run mode requires a working communications link to the host computer. This may be a further overhead and a debugging challenge in itself (to get the communications working).

### RTOS-Aware Debugging Functionality

RTOS awareness may mean different things to different people and is very dependent upon the tools and RTOS in use. The two key areas of functionality are the ability to view data, in an RTOS-aware fashion, and the control of RTOS functions and objects.

#### *Viewing*

All of the information concerning the status of a running system is contained in its memory. Meaningful viewing of this data is simply a matter of knowing where it is located and how it is structured and formatted.

For tasks, it is necessary to be able to view a list of tasks in the system, with their running state (typically "current," "ready," "blocked," or "suspended") and priority (see Figure 7.3). For each individual task, it is desirable to be able to view local variables, registers (including the program counter), and stack.

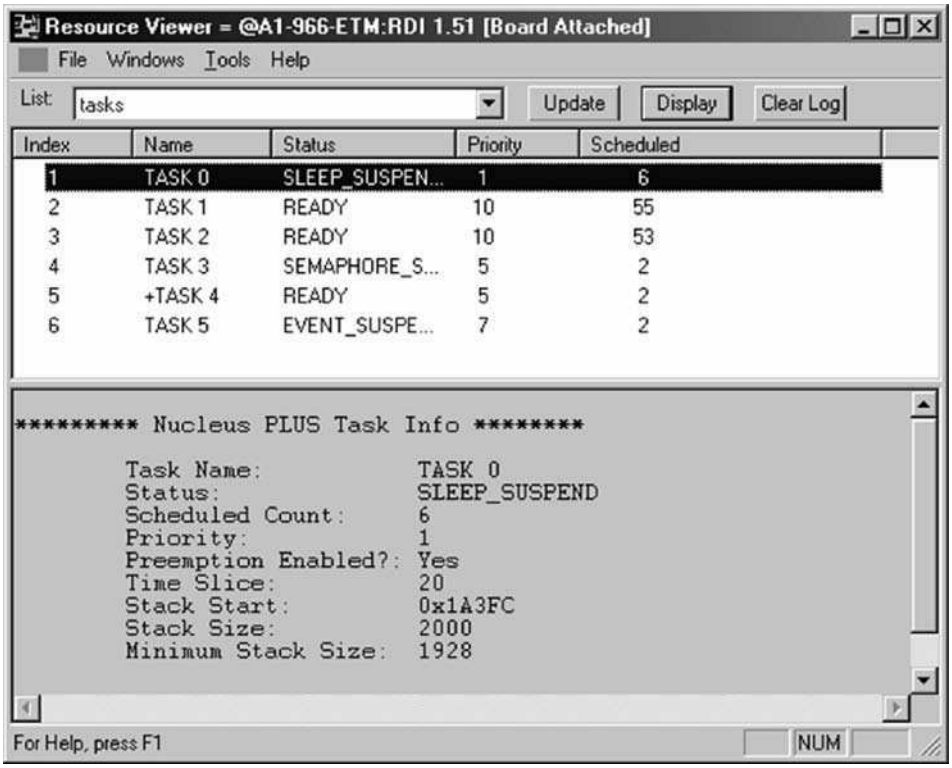



*Figure 7.3: Task information display*

Lists and information about other RTOS objects (e.g., mailboxes, queues, semaphores, mutexes, and messages) are also required.

### *Controlling*

A task-aware debugger may also exercise some control over the target system. There are various possibilities for controlling tasks:

- A task may be suspended—not available to be scheduled.
- A new instance of a task may be spawned (in an RTOS that supports dynamic task creation).
- A task's priority may be adjusted.
- A task may be paused, while others continue to execute (if run-mode debugging is available—see the section "Run Mode" earlier in this article).

Additionally, it is useful to be able to stop the whole system (i.e., suspend the scheduler), when using run mode.

Breakpoints may also be applied with task awareness (see the section "Task-Aware Breakpoints," later in this article).

## Shared Code

In C, it is very common for a function to be called from several other functions. When using an RTOS, the calling functions could be executing in the context of different tasks. Thus, the called function may be in use more than once "simultaneously," which is not really a problem. The only precaution that should be taken is to ensure that the function is reentrant—that is, all its data space is dynamically allocated, so that no static variables are used (see Figure 7.4).

If a system requires multiple tasks, all performing the same operations on different data, there is no need for multiple copies of the code. A single copy will suffice. The task may be instantiated multiple times (resulting in multiple task control block entries in the RTOS), each with its own data. See "Shared Code to the Rescue" at the end of this article.

## Task-Aware Breakpoints

If any code in an RTOS-based application is shared, then debugging is greatly eased by the availability of task-aware breakpoints. Such a breakpoint is similar to any other, except that it is qualified by the identifier of the task (or maybe tasks) in which it is required to be active.

In fact, a breakpoint can (almost) never be truly "task aware." The apparent

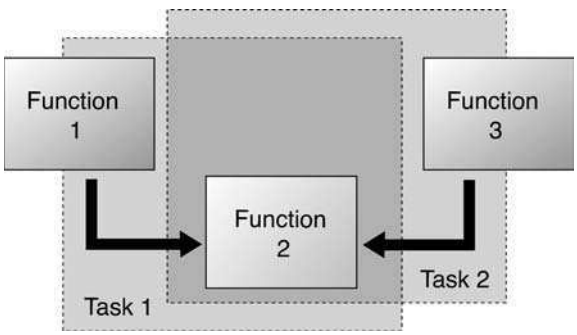



*Figure 7.4: Shared code*

awareness is the result of the action taken by the debugger when a breakpoint is hit. A decision is made as to whether execution should continue (the current task is not the one in which the breakpoint is required) or stop.

This task-context decision always represents a (usually small) time overhead, which has various levels of impact, depending upon its implementation. This overhead is called *breakpoint latency*. Broadly, the task-context decision may be made on the target (by a monitor program running there) or on the host (by the debugger itself).

#### *Target*

If the task context is evaluated on the target, the breakpoint latency is reduced. The delay may actually (only) occur at the time a breakpoint is hit, or it may impact the task-context switch each time the task is scheduled. This is an implementation detail.

In either case, a monitor program is required in the target, which may be quite complex to implement. However, the resulting facility is most flexible.

#### *Host*

The alternative is for every breakpoint to halt the target and for the task-context decision to be made by the debugger. Even if this decision is made very quickly, the breakpoint latency will tend to be higher because of the host-target communications.

With this approach, no support is required at all on the target, and the coupling between the debugger and the RTOS is much looser. Since the implementation is quite simple, it is an ideal choice for an in-house RTOS.

#### *Breakpoint Scope/Action*

It is important to differentiate between the scope of a breakpoint and the action that it will take.

A breakpoint's scope is the task or tasks in the context of which it will be activated. This is only relevant if code is shared between tasks. Typically, a breakpoint may have the scope of a single task, several instances, or all instantiations of the shared code.

The action of the breakpoint is what happens when an active breakpoint is hit. Possibilities include:

- Stop the system (this is the only option if stop-mode debugging is used).
- Stop just the scoped task.
- Stop a number of tasks (normally including the scoped one).
- Stop all the tasks (but interrupts continue to be serviced).

### Dependent Tasks

It is very common for task interdependencies to occur in an RTOS-based application.

A simple case occurs when one task is generating data that is sent to another for processing. These tasks are interdependent (see Figure 7.5). The Receiver depends on the

Sender for data to process; the Sender is dependent on the Receiver to accept the data that it has generated (otherwise, its buffer will fill up). This second case is very relevant to debugging. If a breakpoint results in (only) the Receiver task being stopped, the Sender task will rapidly reach an unstable state.

What is required is a means by which a breakpoint on the Receiver task will also result in the Sender task being stopped. This is accommodated by a synchronized breakpoint.

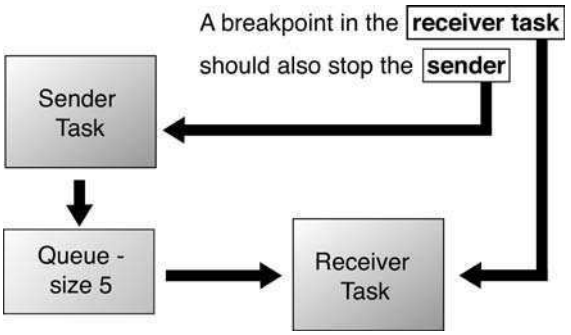



*Figure 7.5: Dependent tasks*

### *Synchronized Breakpoints*

The idea is simple: when a task-aware breakpoint is set in one task, a list of other tasks is provided; these other tasks should also be halted when the breakpoint is hit. The additional tasks may be other instances of the scoped task or they may be different code entirely (or both).

Synchronized breakpoints can be implemented only when run-mode debugging is in use because the remainder of the system continues to execute.

It is possible to implement this facility on the host or on the target. A host implementation is relatively simple but results in a small delay (latency) while the synchronized task list is processed; this results in each of the additional tasks over-running slightly, which is rarely a problem. A target implementation may be much more complex, but latency is reduced.

## Memory Management Units

The use of a memory management unit (MMU), in some form, is common with many modern microprocessors. The necessity of using an MMU may be to implement a simple inter-task memory protection or for the full implementation of a "process" model. The advantages and disadvantages from the design and implementation perspective are widely documented, but the debugging implications are rarely considered.

### *Simple Memory Protection*

An MMU may be used in a relatively simple way to afford memory protection between tasks. Each task is permitted to access only specific areas of memory. Thus, each task is protected from interference by others and its own erroneous behavior (e.g., illegal memory references, stack violations, etc.).

From the debugging perspective, this use of an MMU is not a significant problem. The debugger simply needs to deactivate the MMU to access all of the memory which has no impact upon the integrity of the software itself.

### The Process Model

An MMU may be used to remap the memory address space for each task so that it appears to have control of a complete processor. Such a task is commonly referred to as a *process*. It offers greater flexibility and protection, but complicates debugging. The debugger needs access to the remapping information so that it can use the MMU to access the memory space of each task.

It is possible to have a "hybrid" memory model, where each process itself contains multiple threads of execution (as in Windows and UNIX). This, in turn, provides the opportunity for a hybrid debugging capability: a debugger, which is associated with a single process in the system and provides thread-aware debugging of its code.

## Multiple Processors

The deployment of multiple processors in embedded systems is increasingly common. This may be by means of multiple boards in the system, multiple devices on a board, multiple processor cores on a chip, or any combination. From the debugging perspective, architectural details are relatively unimportant.

The processors may be all the same type—an array of processors may be a useful solution to certain types of problem. Alternatively, there may be a mixture of architectures, where different processor capacities (8, 16, or 32 bit) or capabilities (e.g., DSP) are useful.

Debugging, with multiple processors, may be approached in various ways. Multiple debuggers could be employed—one for each processor—but this approach could be complex because each is likely to be a different tool. There may also be connection capacity issues. A single debugger is possible, if the debugging architecture of choice accommodates the "true simultaneity" paradigm. This results in a less-steep learning curve (only one tool) and opens the possibility for advanced debugging functionality, like synchronized breakpoints.

Of course, multiple processor systems are just as likely to use an RTOS and, hence, have particular debugging needs. This can get more "interesting" because multiple RTOS products may actually be deployed in a single system. This would be a challenge for any debugging technology—a challenge but not an insurmountable one.

## Conclusions

It is clear that debugging, when an RTOS is in use, may be complex business, and the development of suitable tools can be nontrivial. As systems become increasingly complex, the capabilities of the debugging tools need to track the needs of embedded software developers.

Understanding RTOS debugging is important for both developers and managers. Even if the functionality is transparent (because it has be addressed by the RTOS/tools vendor), the developer can use this knowledge to appreciate the possibilities and limitations

of the technology. A manager can efficiently deploy resources and make informed purchasing decisions.

### Shared Code to the Rescue

An embedded developer contacted their RTOS/tools supplier with a problem: his design had a fixed amount of code memory, which was all used, and further enhancements were required to the application.

The support engineer looked at what was being done. He tried improving compiler optimization, which was helpful, but nowhere near enough extra memory was freed. Then he took a closer look at the application, made a fairly small change to the structure of the code, and reduced the code memory requirements by *nearly 90%!*

The application was some kind of data router that handled 10 channels of data. Each channel was handled identically by its own task. Originally, each task had its own copy of the code. The modification that saved so much memory was to share one copy of the code between all 10 tasks.

This introduced a new problem, however. Previously, a breakpoint, placed in the code of one task, would only take effect in the context of that task. The code was not shared and, hence, only executed in the single-task context. Now the breakpoint would trip regardless of which task utilized the line of code.

The solution was a debugger with task-aware breakpoints, which the RTOS/tools vendor could readily supply.

So the story had a happy ending for all concerned.

## 7.2 A Debugging Solution for a Custom Real-Time Operating System

*Users of in-house real-time operating systems are challenged when it comes to debugging. In a piece written for* NewBits *in 1998, Robin Smith (who is now with Sun Microsystems) and I described how to effectively utilize a debugger with a custom RTOS. I have adapted the material for this article. (CW)*

A discussion of task-aware debugging is worthwhile when a commercially supported real-time operating system (RTOS) is in use. It is an entirely reasonable expectation that the RTOS vendor is able to provide a complete debugging solution with a choice of host- or target-based technology. If an in-house designed ("custom" or "home brew") RTOS is being employed, it is unlikely that the development of a custom debugger is also feasible.

The only real possibility is to select a debugging technology that allows significant user-customization. This necessitates a host-based approach, but this is generally quite satisfactory and a great improvement on a total lack of task-aware debugging.

The remainder of this article will describe the basic details of such an implementation.

### Implementing Task-Aware Debugging

To illustrate an example of the implementation of task-aware debugging, we need to select a target microprocessor, a debugging technology, and define an RTOS.

#### *Example Target Device*

No specific processor is considered because this is not necessary for the discussion. The assumption is simply made that the device has a 32-bit architecture. None of the ideas presented are at all specific to any particular device.

#### *Example Debugger*

The specific debugging technology employed is also unimportant. For the purposes of describing RTOS awareness implementation, we have assumed that the debugger simply has a scripting facility that looks a lot like C language.

#### *Example RTOS*

Obviously, every custom RTOS is different, and it would not be useful to describe a particular example here. However, it turns out that there is very little that we need to know in order to present the ideas.

For the purposes of this article, assume the following about the RTOS:

- Each task has a task-control block (TCB), which describes characteristics of that task and holds its current status.
- The format of the TCB is illustrated in Figure 7.6.
- The TCBs are assembled into a table that is a doubly linked list with `NULL` pointer terminators.

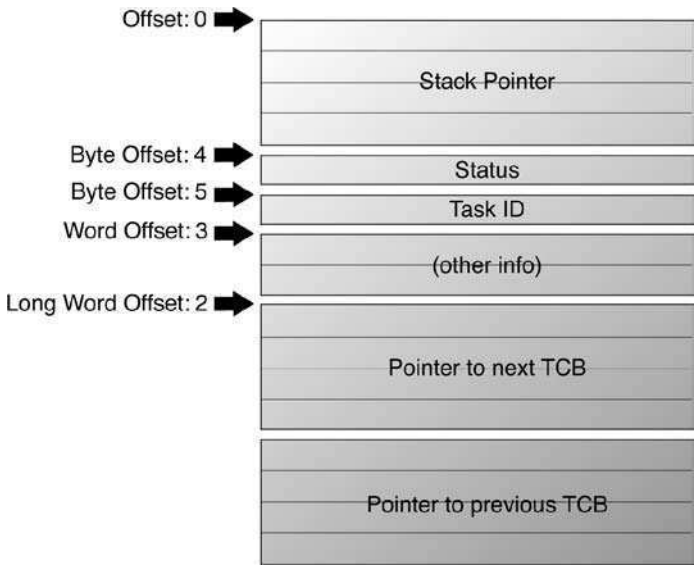



*Figure 7.6: TCB layout*

- The names of pointers to the start of the TCB table and to the TCB table entry for the currently running task are `TCBList` and `TCBCurrent`, respectively. And these symbols are accessible to the debugger.

### Task-Awareness Facilities

In this article, the implementation of task-awareness will be limited to two specific facilities:

- Displaying status of all the tasks in the system
- Task-aware breakpoints

Given an understanding of these two facilities, you will be well equipped to design and implement similar and additional facilities for your custom RTOS.

#### *Task Information Display*

To implement a task information display, the first requirement is a set of utility scripts that extract the necessary information from TCB entries:

```
void *GetNextTCB(TCB)
int *TCB;
{
      return(*(TCB + 0x2));
}
```

This returns a pointer to the next TCB in the chain, which enables the list to be "walked." It returns a `NULL` pointer when the end of the list is reached. A similar script could be implemented to traverse the list backwards, if that were required.

```
int *GetStackPointer(TCB)
int *TCB;
{
      return(*(TCB + 0x0));
}
```

This returns the value of the task's stack pointer. This information can be used to gain access to the context information (registers, etc.) of the task, which were pushed onto the stack, noting that it is assumed that stack entries for this processor are aligned on integer boundaries (32 bits in this case).

`GetID()` and `GetStatus()` each return other useful information about the task:

```
char GetID(TCB)
unsigned char *TCB;
{
      return(*(TCB + 0x5));
}

      char GetStatus(TCB)
      unsigned char *TCB;
{
      return(*(TCB + 0x4));
}
```

The task information display command is implemented using a further script:

```
int GetTaskInfo()
{
      unsigned char id;
      unsigned char status;
      char *pp;
      unsigned int sp;
      void *tcb;

      tcb = TCBList;

      printf("ID TCB      Status StackPointer\n");
      printf("-- ---      ------ ------------\n");
```

*Continued*

```
        while(tcb != 0)
        {
              id = GetID(tcb);
              status = GetStatus(tcb);
              if (status == 0)
                  pp = "Ready";
              else
                  pp = "Wait ";
              sp = GetStackPointer(tcb);
              printf ("%02u %08X %s %08X\n",id,tcb,pp,sp);
              tcb = GetNextTCB(tcb);
        }
}
```

This makes use of the utility scripts to traverse the TCB table and display information about each of the tasks in the system.

A useful enhancement would be the addition of some code to detect when `tcb` has the value of `TCBCurrent` and maybe add a `"*"` to the listing line. This would highlight the currently running task.

### *Task-Aware Breakpoints*

With many debuggers, when a breakpoint is set, a script may optionally be associated with it. When the breakpoint is hit, the script is executed. If the script returns a value that may be interpreted as `TRUE` (non-zero), execution continues when the script has completed. The user may be quite unaware that a breakpoint has been hit. If the script returns `FALSE`, execution is suspended in the usual way.

This script may be used to implement task-aware breakpoints:

```
int fortask(id)
int id;
{
         unsigned char taskid;
         void *tcb;

         tcb = TCBCurrent;
         taskid = GetID(tcb);
         return(!(id == taskid));
}
```

Every task has a unique identifier (ID). The task ID is passed as a parameter to the script. Using the current task TCB pointer, `TCBCurrent`, the ID of the current task is compared with that specified; execution is suspended only when they match.

This script could be enhanced to issue an appropriate explanatory message when it stops the execution. Optionally, a facility to display instances where the breakpoint was hit, but execution was not stopped (i.e., a different task was active), could be added for diagnostic purposes.

## Conclusions

When using any RTOS, it is desirable to have an RTOS-aware debugging solution. If a commercial RTOS technology is being employed, a range of debugging tools should be available from the RTOS vendor. If a custom or non-commercial RTOS is used, a good debugging solution may be implemented using the techniques described in this article.

## 7.3 Debugging—Stack Overflows

*This article is based on a short piece, which appeared in the Winter 2003 issue of* NewBits, *by Larry Hardin, customer support manager at Accelerated Technology. The ideas tie in well with my article, "Self-Testing in Embedded Systems" in Chapter 3. (CW)*

Let's talk about one of the types of problems we see in support regarding debugging—stack overflows. We'll discuss a few causes and then propose suggestions, based upon our experiences, to avoid the problem.

First, using `printf()` as a debugging aid. Although common in a Windows environment, this practice may present problems in an embedded system. Be aware that using library calls may bring in other modules to support that call, and `printf()` is no exception. Math subroutines may also be needed. And although it's not as common, the I/O functions provided with some toolkits may still not be reentrant. Keep these two factors at the back of your mind.

Monitor recursion carefully. Sometimes the level of recursion isn't factored correctly and each function call is pushing data onto your stack. Be sure to use stack-checking during development, if your design includes recursion! If your kernel doesn't have a stack-checking capability, write your own. Fill your stack with a pre-set pattern and monitor your stack fence. Your stack should always have a bit of "extra" space factored in. So, if you see the pattern cleared up to the stack fence, you may well have just blown that stack. Blowing a stack can either be easy to find or a bear to uncover—depending upon what data structure (and what part of that structure) your application just overwrote.

Some kernels incorporate a history-saving capability. Functions make an entry into a buffer at function entry, so a record of execution is formed. This can be a useful tool to show where you've been at crash time. If you aren't using an analysis tool that shows this information, peeking at the memory of the history buffer may help you locate the faulty function/code.

### Conclusions

Although the use of multiple stacks with an RTOS can introduce difficult-to-locate errors, some care and forethought can prevent the majority of such problems.

## 7.4 Bring in the Pros—When to Consider a Commercial RTOS

*Selling any product can be a challenge in the modern world. You need to get the right price, features, and performance for your customers, and you need to understand how to defeat your competition. In the commercial real-time operating system business, most of this is still true, but with one twist: sometimes—actually quite often—our customers are also our competitors! I am often asked why one should buy an RTOS, when writing one in-house is a possibility. I have made many seminar presentations and conference keynotes on this topic, and I wrote about my ideas in* NewBits *in 2003. That piece was the basis for this article. (CW)*

Although any self-respecting commercial real-time operating system supplier might wish to deny the possibility, writing your own RTOS is an option. It is an option that is exercised by many developers, even though a wide selection of RTOS products is on offer. In this article, we will review the wisdom of the "do-it-yourself" approach and consider how this compares with a commercial RTOS product.

### Commercial and Custom RTOSes

Some surveys suggest that as many as half of all current embedded designs are implemented using an in-house, custom RTOS. The reasons for making this choice, despite a great many commercial products being available, are varied. To help understand how such decisions are made, we need to investigate the pros and cons of each approach.

### Advantages of a Commercial RTOS

For a commercial RTOS product to be attractive, it must be well established and reliable. Clearly this is largely down to the company that is supplying it. If they are a secure, well-known firm of reasonable size, confidence is increased. Of course the product should be mature, with a sizeable user base. A "one-man shop" can hardly engender such confidence.

Both the purchaser and supplier of an RTOS are expressing a long-term commitment. The user is committing expertise and investment going forward. The vendor is committing to support new processor architectures and the latest technology, as it comes along.

Documentation is important. A vendor should be quite prepared to provide copies to prospective customers. In past years, there were excuses (e.g., shipping costs, lead times); now a PDF by return email is a reasonable expectation. If the source code is provided, or at least available, it may be regarded as additional documentation (but not a replacement for it!), so long as it is readable and is well commented.

You need RTOS-aware debugging. Stop mode is essential; run mode may be useful too. A commercial product should have an adequate provision for these capabilities.

A commercial RTOS is not just a kernel. All the surrounding technology—the layered products—is likely to be available off the shelf. These include networking protocols, file systems, graphics, and Java. Of course, they must be options. If a file system, for example, is not required for an application, there is no reason why it should be purchased or be included in the RTOS memory footprint.

A difficult area of any embedded software design is the low-level interface to hardware—the drivers. A commercial RTOS will be provided with a selection of drivers for standard and integrated devices. Although this provision cannot encompass custom hardware, the RTOS vendor will most likely provide tools or templates to make driver development as straightforward as possible.

### Commercial RTOS Downsides

The decision to not use a commercial RTOS is influenced by a number of possible downsides. In some cases, they may be valid, but others are misconceptions.

An initial, and very sensible, consideration is often cost. Specifically, for many applications there are concerns about on-going license costs. For a high-volume, low-price embedded product, per-unit production costs are critical. It is unlikely that paying a royalty on an RTOS would be acceptable, and a royalty-free RTOS makes sense. However, the business model for a low-volume, high-ticket product is quite different, and a "pay-as-you-go" royalty fee may be quite appropriate. For other applications, other business models (e.g., subscription plans) may be relevant.

The key to success is flexibility. Every embedded system is different, and the RTOS vendor needs to accommodate those differences in their product offerings.

Some engineers are nervous about their lack of internal knowledge of a commercial RTOS. This really should not be a concern. You use a PC without needing to understand the intricacies of Windows; you can drive a car without ever lifting the hood. Why do you need to know exactly how an RTOS works, so long as it performs as specified? It may even be argued that employing staff in possession of this internal knowledge is an unnecessary expense. Of course, the availability of source code may alleviate these insecurities.

Suppliers of any product are keen to get vendor lock-in from their customers, and RTOS providers are no exception. On the other hand, their customers want to maintain flexibility and may feel that such a lock-in is a drawback. Although there is some validity in this concern, migration from one RTOS to another need not be a very big issue. Also, selecting a product that adheres to a standard (e.g., POSIX or OSEK) gives the flexibility to switch suppliers if the need arises.

The most fallacious argument against a commercial RTOS is that it will intrinsically carry the burden of excess functionality. Although it may be true that an RTOS possesses many capabilities that are not required in a particular design, just about every modern commercial RTOS product is scalable—that is, only the required functionality is included in the final memory image. Take a look at Figure 7.7 to see how this works. Of course, the implementation of scalability may be more efficient in some products than others, and this should be verified. There is the additional argument that this excess functionality is being paid for, which is true but irrelevant because the costs are factored over a great many designs (from multiple customers).

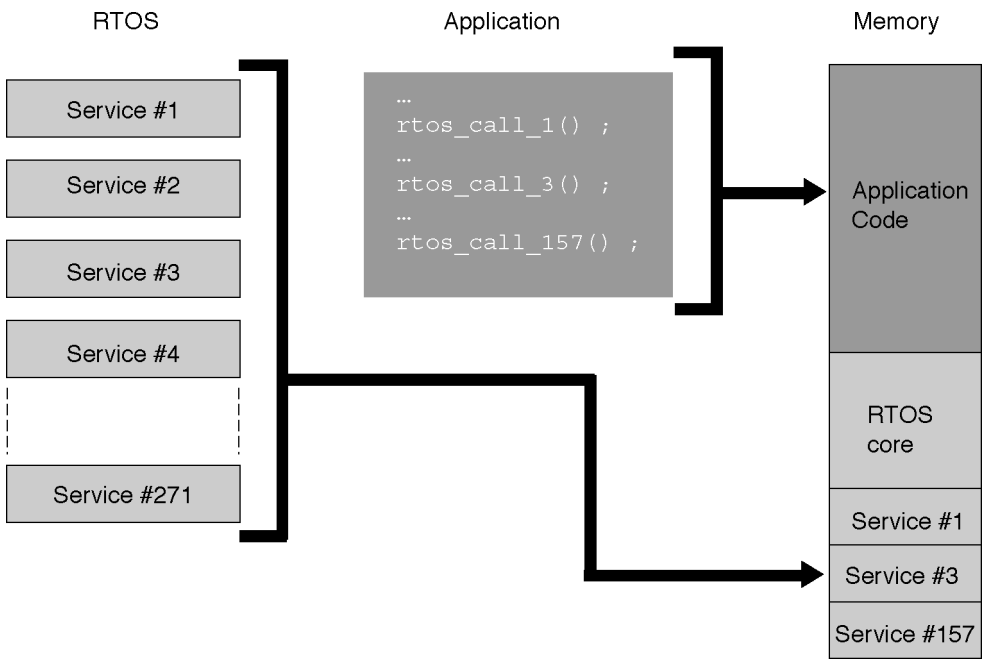



*Figure 7.7: RTOS scalability*

### Why Write a Custom RTOS?

The primary motivation for creating an in-house RTOS is to retain control. If it is developed in-house, obviously all the code is owned and internal knowledge is retained. Although this is true, it presupposes that there is a stable team who understands and can maintain the RTOS. Since this requires a specialized skill set, this may be an expensive proposition.

Of course, there is the argument that there are no on-going license costs. This is true, but as we have seen, this may also be the case with a commercial, royalty-free RTOS.

Engineers often argue that an in-house RTOS is an exact match to their needs. This argument is hard to sustain because a properly scalable commercial RTOS will scale to precisely meet their requirements. Most often, the functionality of a custom RTOS is limited by what can be achieved within the available time/budget.

### Reasons Not to Create a Custom RTOS

There are a great many factors that make the development of an in-house RTOS questionable.

As usual, the analysis starts with money. An RTOS has a development cost associated with it. This cost may be very substantial, but not visible, as it gets "lost" in a project's

budget. This is bad management and it is illogical. It is not normal practice to pay for capital equipment and other reusable resources (like an RTOS) on the back of one project, when it may benefit many others (now and in the future).

The costs continue, if you take into account long-term support of the RTOS. This cost may escalate if there is a need to retain specific skilled staff for the purpose.

Given that the costs of developing and supporting an RTOS are understood and accepted, there is another requirement: debugging. Embedded software developers spend a lot of their time debugging. The debugger really needs to have some kind of RTOS awareness. It has been suggested that the cost of developing a debugger from scratch may by ten times that of developing an RTOS. There may be an option to adapt a commercial debugger to work with a custom RTOS—this should certainly be considered.

Although there may be a clear intention to develop a single RTOS, there is a distinct danger that, in a less strictly controlled development environment, multiple implementations will result. (See the "Once Upon a Time" sidebar on the next page.)

If the requirement is for just a kernel, then writing it may be a contained and manageable task, but will it stop there? It is rare for modern embedded systems to be that simple. There is generally some requirement for additional layered products—protocols, graphics, and so forth. This could be accommodated in three ways:

- Write these too, but that may be a lot of work.
- Persuade a commercial vendor to port to your RTOS, but that would be difficult and expensive.
- Port an open-source product to your RTOS, which could be as hard as the first option.

Whatever device is being used today, it is certain that a different processor architecture will be used in the future. Such a change brings with it many challenges. Since an in-house RTOS is likely to be written using a significant amount of assembly language, its portability is going to be limited. Additionally, such code may include endianity dependencies and will also rely on the interrupt structure of the processor.

A final concern about developing an in-house RTOS is to consider core competencies. If a company's business is developing networking hardware, for example, their goal is to address that business as effectively as possible. They aim to produce the fastest, cheapest, most functional (or whatever) routers on the market. Getting distracted by spending resources developing software, which could be readily obtained off the shelf, is not playing to the strengths—the core competencies—of the company. (See the "Painting the Car Red" sidebar later in this article.)

### A Middle Ground?

In this article, we have considered two options: licensing a commercial RTOS or writing one in-house. Is there another possibility?

Many users think there is and look at open source solutions, which seem to give many of the advantages of an in-house solution without the problems. Although it is sensible to investigate all possibilities, some key questions must be answered:

- **What about support?** It is good to get the source code for free, but how much investment in time and effort is required to understand what is happening. Some firms offer support at a price—on Embedded Linux, for example. This cost is often comparable with the license and support cost of a commercial RTOS.
- **How fit is the product for the intended purpose?** If your needs are for real-time performance, is Linux a good option, given that it is not intrinsically real-time? It can be made to behave in a real-time fashion, but this renders the wide range of available hardware drivers quite useless. Their availability is one of the perceived advantages of the Linux approach. If memory is limited, more care is needed to ensure that the final code only includes the required functionality.
- **What is the legal status of an open-source RTOS?** This should be clear enough, because a license still needs to be signed in order to deploy the RTOS. However, this does not seem to be quite so clear-cut, and at the time of writing, some high-profile legal wrangles are making the headlines. The outcome of these disputes may have a knock-on effect upon every embedded system shipped with an open-source RTOS.

### Conclusions

It is fair to say that, as I am employed by an RTOS and embedded tools vendor, my view is biased. However, it is certainly clear that, in the vast majority of situations, there is very limited justification for deploying a custom RTOS in a new project.

#### Once Upon a Time

There was a young engineer working on embedded software (although this was so long ago that the word *embedded* had yet to be coined). He was working on small, 8-bit systems built around the Z80.

One day, he was starting a new project and concluded that a small kernel would be useful—just a time-sliced task swapper really. He spent a couple of weeks working on this project, resulting in about 2 K of (assembler) code, which worked well. The project was completed with no problems.

For the next project, he decided to use the kernel again, but he spent a few days incorporating new ideas for enhancements to the kernel into his code. The updated kernel was used in the project, and again, it was completed without undue difficulties.

By the time the third project came along, this engineer felt he knew a lot about kernels and didn't hesitate before including yet more ideas before utilizing his kernel again. Once again, all went well with the project.

At this point, it is reasonable to feel that it was high time this young engineer had a "reality check"; things have been going just too well for him. That's just what happened.

The hardware from the first of these three projects was enhanced, and this meant that revisions to the software were necessary. So our hero got out the listings and considered what was to be done. But he had a shock: what was this strange kernel that someone had used?

Of course, he had been the victim of "creeping elegance." He set out to create a kernel but ended up with three. Sure, they were related, but the urge to "do a bit better" meant they were not particularly compatible.

At around this time, the engineer (who will continue to remain nameless) left the company to join an embedded tools and RTOS vendor, leaving his three kernels behind him. Since he had put some effort into documentation, he did not leave his employers with too big a maintenance problem, but that was more luck than judgment.

### Painting the Car Red

In the modern business world, successful companies are the ones that focus on what they do best and excel at it. The embedded systems business is no exception, but we can look elsewhere to find everyday examples. Take the automotive business, for instance.

Henry Ford famously promoted his company's flexibility by offering the Model T in any color, "so long as it's black." This would not work in today's car market, where the choice of color and the quality of the paint finish are key attributes in a car-buying decision.

But how many car makers manufacture their own paint? Answer: none (essentially—there may be the odd maverick). Car makers always outsource it. They are very careful where they get their paint from and specify it in very great detail, but they leave paint making to the paint manufacturers and get on with making cars.

This approach is applied to many aspects of an automobile: the upholstery, the electronics, the suspension, even the engine and gearbox. Some manufacturers license an entire car design from elsewhere if their core competency is finishing and production. We can learn from the car makers.

## 7.5 On the Move

*It is frequently assumed that, having utilized a particular real-time operating system, you are locked in and that changing the RTOS will be problematic. I wrote a piece for* NewBits *in late 2004, upon which this article is based, to dispel some of the myths. (CW)*

### Migrating from One RTOS to Another

It is inevitable that, at some time or another, an embedded software team will face the prospect of moving from using one real-time operating system (RTOS) to another. This migration may be brought about by many factors, including:

- RTOS availability for new chip architecture
- Customer demands
- Adherence to standards
- Change in technical requirements
- Change in commercial business model

In this article, we will look at a number of facets of such a migration and its impact on embedded software development. For example:

- How can migration of code be managed?
- What about migration of skills?
- What preparations can be made for this inevitable migration?

In general, these matters are independent of whether the migration is between commercial RTOSs or from an in-house kernel to a bought-in product.

### Code Migration

An initial challenge, when adopting a new RTOS, is the body of existing code, which is written against a specific application programming interface (API). Clearly this code has to be modified or adapted in some way to accommodate the API of the new RTOS.

Generally, this is done by hand, which may be quite satisfactory if the application is small or the code that interacts with the RTOS is localized. The difficulty of this task is also strongly affected by the nature of the old and new RTOSs. If they have similar APIs, the job will be easier. It may also be simplified if the "target" RTOS has a rich, orthogonal API because this is intrinsically more expressive.

Some tools are available to assist with this endeavor, but their availability is limited to specific RTOS migration paths. A creative way of addressing this problem is the use of a "wrapper."

### Wrappers

A wrapper is simply a layer of software between the RTOS and the application code. It exposes an API and maps calls to this interface onto the actual API of the underlying RTOS. This process is illustrated in Figure 7.8. Here, call `1` to an old RTOS is mapped

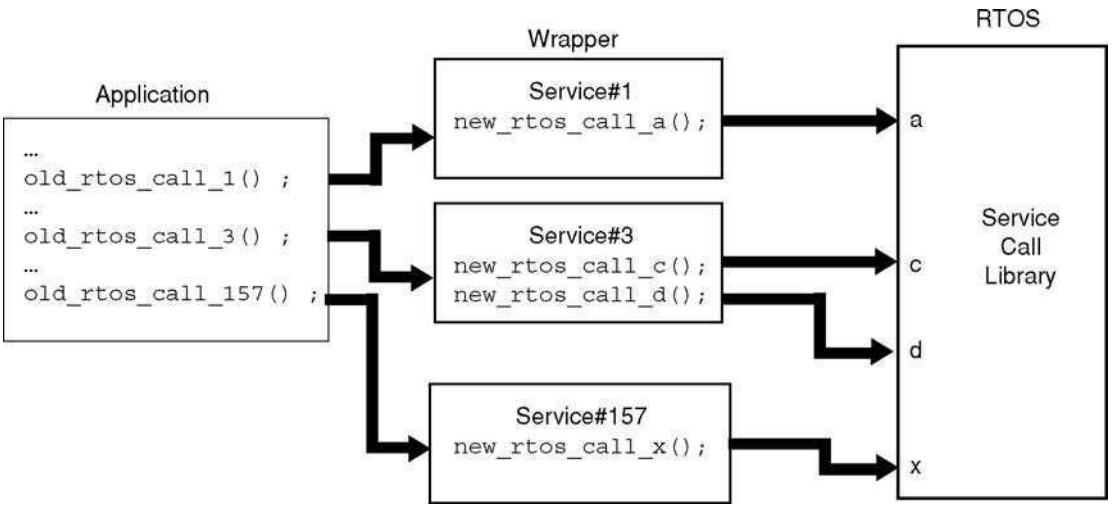



*Figure 7.8: API wrapper implementation*

onto call `a` of the new one; similarly, call `157` is mapped onto call `x`. Call `3` in the old RTOS does not have an exact equivalent in the new RTOS but results in two calls to services `c` and `d`. In some cases, additional processing of data may be required, and "straight" mapping may not be possible.

### *Strategies*

A number of different strategies may be employed in the use of wrappers, depending upon particular circumstances:

- **Map an old RTOS API onto a new one:** This obvious approach may be very successful because it directly addresses the problem at hand. It is somewhat inflexible, but, for a one-off migration, the inflexibility may not be important. Once an application has been migrated to a new RTOS in this way, new code can make use of the native API of the new RTOS. References to the old RTOS gradually will "fade away," and eventually, the wrapper may be discarded (see the section "API Availability" later in this article).
- **Design a "neutral" API and implement a wrapper for each selected RTOS:** This preemptive strategy is used by some developers to facilitate relatively easy changes of the RTOS. The intention is to avoid locking in the application code to a particular vendor's API.
- **Use a wrapper to implement a standard API on a proprietary RTOS:** This strategy is similar to the preceding one, but it leverages accepted standards. Some commercial implementations of standard APIs are implemented in this way. See the sidebar "RTOS Standards" later in this article.

### Implementation

The obvious way to implement a wrapper was illustrated previously in Figure 7.8. The wrapper is simply a library of functions that answer the required API and, in turn, make calls to the underlying RTOS. Done properly, such a wrapper is scalable in a way similar to most modern RTOS products. It may be possible to optimize this implementation by having the wrapper make jumps, instead of calls, to the RTOS service calls. The return from the actual service call will go back to the calling applications code.

If the wrapper is implementing an API that is not very different from the existing one, it may be possible to achieve a satisfactory result using C language `#define` macros. This is likely to introduce little or no overhead.

C++ offers a couple of different approaches to implementing an RTOS wrapper. A series of classes, representing various RTOS objects, may be created with all the API references hidden inside the class member functions. So, for example, creating a task may simply be a matter of instantiating a task object—the constructor would make the actual RTOS call to create a task. Alternatively, application-oriented classes may be employed, where the RTOS specifics are again contained, localized, and hidden within member functions.

### Overhead

What does it cost? Nothing comes free in this world. So it is reasonable to expect that a wrapper will have a memory and/or performance hit. Unless a pure macro-based approach is viable, that indeed will be the case.

Taking the example of the call-based approach, every RTOS service call will incur an additional call/return time overhead. As previously commented, this overhead may be reduced if a jump/return sequence can be implemented. The memory overhead may be minimized by implementing the wrapper as a library (so that it scales automatically and no excess code is included).

In most cases, a modest overhead is entirely acceptable. In some cases, applications actually run faster using a wrapper on top of a modern RTOS than they did with the old RTOS.

### Challenges

The level of difficulty in creating an RTOS wrapper is broadly governed by how well the architecture of one RTOS maps onto the other. Very large conceptual differences would render a wrapper implementation impossible or, at best, very inefficient. In most cases, the challenges are associated with specific details of the mapping that affect the wrapper. For example:

- **What kind of task identifier is used?** This may be a name, a number, or a pointer to a data structure. It may even double as the task priority.
- **How many priorities does the RTOS support?** If the new RTOS has less than the old one, this challenge could lead to difficulties. If the RTOS supports

more priorities, should the translation spread the old ones over the new, wider range? There is also the issue of how priority is represented; do priorities start from 0 or 1? Is 0 the highest or lowest priority?

- **How are other facilities implemented?** Even if both RTOSs have the same facilities, there will be implementation differences. For example, if the old RTOS has bigger mailbox data structures than the new RTOS, the wrapper may need to manage the splitting of data over two mailboxes. This kind of mapping is messy but manageable.

### *API Availability*

It is important to appreciate that using a wrapper for moving code from an old RTOS to a new one should normally be regarded as a temporary measure. As illustrated in Figure 7.9, there are three stages:

1. At the beginning, there is just the application code and the old RTOS. The wrapper is implemented to facilitate the use of the new RTOS; the application code is unchanged.
2. Over time, the application code is enhanced, making use of the native API of the new RTOS. The calls using the old API, via the wrapper, may be selectively replaced to optimize the code.
3. Eventually, no references to the old RTOS will remain, and the wrapper may be discarded.

The visibility of the native API of the underlying RTOS can also be useful when a wrapper is being used to implement a standard API. It may happen that the standard API lacks a particularly useful feature, which may be found in the RTOS API. Using a wrapper means that such a feature may be employed.

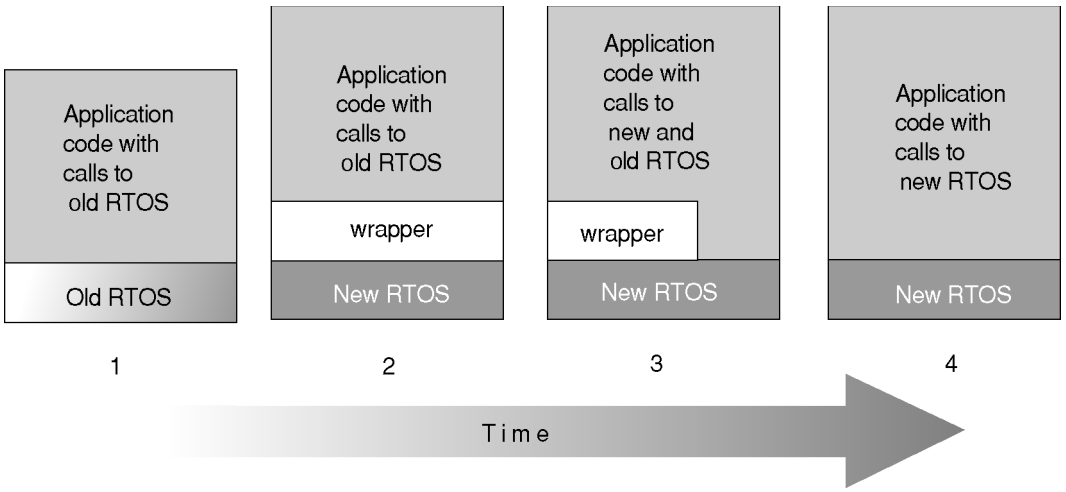



*Figure 7.9: API availability*

It should be remembered that an API is not necessarily confined to the kernel—it extends to all the "middleware" (layered products like networking, file system, graphics, etc.). So the use of a wrapper on top of a commercial RTOS with a very wide range of middleware permits unfettered access to all of this additional functionality.

### Drivers and More

One of the toughest issues with migrating between RTOSs is the "low-level" code—startup, drivers, and so forth. There is little that can be automated in this area. The good news is that, if you are moving to a commercial RTOS product, the RTOS vendor will certainly be able to help. Standard devices should have drivers available off the shelf. And there certainly should be tools or templates to help with custom hardware.

### Debugging Issues

An alternative, somewhat creative, approach to RTOS migration is illustrated in Figure 7.10.

The idea is to accept the fact that debugging matters and make it a priority, which can be viewed as a three-stage process:

1. A custom RTOS is in use, and there are no debugging tools.
2. Adapt a commercially available debugger that already has kernel-awareness capabilities to work with the custom RTOS. Work with this setup for a while to become proficient with the sophisticated debugger.

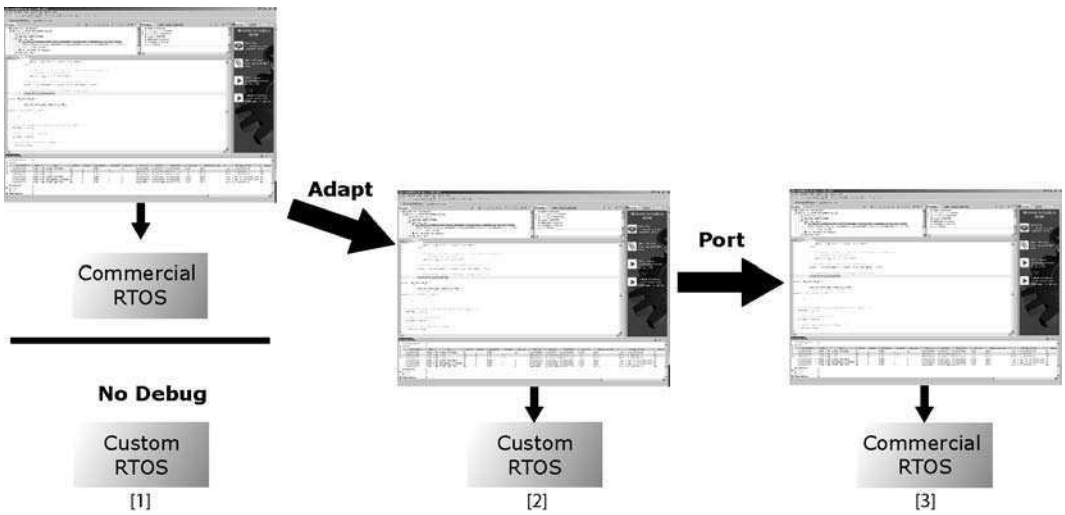



*Figure 7.10: Debugging driving RTOS migration*

3. Move to the new RTOS and reconfigure the debugger. It is at this point that subtle bugs may be introduced (in the migration process). It is, therefore, very fortuitous that a good debugger and the skills to use it are on hand.

## Conclusions

Migration from one RTOS to another, for various reasons, is an inevitable event for embedded developers from time to time. It need not be overly traumatic if the right strategies are adopted and a forward planning approach adopted. Use of a standards-based RTOS may be a preemptive action, which renders migration unnecessary.

### RTOS Standards

Standards are a good thing—such a good thing that, in the RTOS world, we have a bunch of them. A variation on an old joke, but an illustration of the clear fact that, for embedded systems, one size does not fit all.

A big attraction, from the user's perspective, of a standards-based RTOS is a high degree of vendor independence. Suppliers are forced to compete on support, customer service, and product quality, instead of simply locking in their customer base.

Of course, code portability is cited as important, but it is not as significant as portability (and, hence, availability) of skills. Code portability is most useful when considering the licensing or reuse of code from outside of the development team. The topic of standards-based RTOSs could fill a lengthy article, but here we will limit ourselves to a brief discussion of key examples:

#### POSIX

This is the standard UNIX API and is, hence, understood by many programmers who are used to working on the desktop. A real-time variant of the POSIX standard was defined to accommodate the requirements of embedded applications.

#### micro-ITRON

In Japan, more than half of embedded software developers work with TRON standard operating systems. The micro-ITRON standard is, therefore, well established and also of interest to companies outside of Japan who collaborate with Japanese development teams.

#### OSEK/VDX

OSEK was developed by the main automotive companies in Germany, along with the University of Karlsruhr. Later this standard was merged with VDX, which had been developed by their counterparts in France. OSEK/VDX is now very widely used in the automotive industry in Europe and increasingly popular elsewhere in the world.

The standard addresses much more than just the API. All of the key features of an RTOS are covered:

- The kernel architecture (OS)
- Interprocess and interprocessor communications (COM)
- Network management (NM)
- System configuration (OIL)
- Debug interface (ORTI)

It is interesting to note that nothing about OSEK/VDX is *specific* to automotive applications. It has useful capabilities for many embedded applications, particularly when they have the following characteristics:

- Safety critical (or sensitive)
- Distributed (multiprocessor)
- Hard real time
- Static configuration (no dynamic object creation/destruction)

A good possible example would be medical electronics systems.

It is not possible to build an OSEK-compliant RTOS using a wrapper on top of an existing (dynamic) RTOS product. The architecture must be intrinsically static to be deemed compliant.

### Java

The Java language was originally designed with embedded systems in mind. Although it has found many other applications, it is still applicable to embedded applications where dynamic changes to code functionality are required.

But it is a language, so why is it being mentioned under the heading "RTOS Standards"? The answer is that, unlike C and C++, Java "knows" about multi-tasking. Multi-threading is incorporated into the language itself (as it was previously in the Ada language). There is, obviously, some underlying scheduler/RTOS, but it is quite invisible to the Java programmer and can be implemented in a variety of ways. So, it is reasonable to regard Java as an RTOS API.

## 7.6 Introduction to RTOS Driver Development

*It is hard to write about device drivers without being very specific about a particular real-time operating system. In a piece for* Nucleus Reactor *in 1997, Neil Henderson aimed to dispel some of the mystique around writing device drivers; this article is loosely based upon that work. (CW)*

Writing device drivers is generally thought of as a difficult job requiring special skills and knowledge. This may indeed be the case for "bigger" operating systems, which take a UNIX-like approach because they are large complex pieces of software. But for a conventional real-time operating system, which tends to be leaner and simpler, this need not be true. The key requirement is an understanding of the basics, which is the theme of this article.

Device drivers generally consist of two pieces: one for task-level processing, the other for interrupt processing. The manipulation of data and the coordination between both pieces is most important and will be addressed in more detail.

### The Two Sides of a Device Driver

Most device drivers are written to support interrupts. True, device drivers can be more efficiently written without interrupts; however, the reality is that most are written to support them. Since it is easier to write non-interrupt-driven device drivers, if you learn how to write one that uses them, all other implementations are trivial.

Let's quickly review the basics of a device driver: initialize, send, receive, control, and close. There, that was rather painless! When you get right down to it, there is not much to a device driver in most cases. Sure, some extra error processing might need to be done, or you may want to provide an `ioctl()` function, but what it all boils down to is that you are sending and receiving data. Not too complicated! So why do we get all tied up in knots when we have to write a device driver? Well, it's generally because, unless you have some sophisticated tools, you cannot see what the device is doing until it works! However, with the right framework, making a device work quickly is quite straightforward.

So what are the two sides? One side interfaces with the tasking environment, and the other side is an interrupt service routine (ISR) and its related processing. What are the potential problems with coordinating the two sides? One is data corruption, and the other is task scheduling. If the ISR is manipulating pointers into the receive buffer, and those pointers are not protected from dual access, then your buffer data or control structures can be corrupted. Likewise, if you do not properly handle scheduling issues, a task waiting for a device driver service can bring a system to its knees.

### Data Corruption

An RTOS is likely to provide interrupt control and protection mechanisms to address this matter. The one you use depends upon circumstances. If you are in an ISR, where you are permitted to call upon the RTOS services, then the protection mechanism is best because it minimizes the time when interrupts are disabled.

### Thread Control

Why is thread control needed in device drivers? Well, it doesn't have to be, but to develop a driver efficiently, you should treat the driver itself as a part of the requesting thread. This means the thread will need to suspend execution until buffers empty (for sending) or data is available (for receiving). Furthermore, the thread control has to be coordinated between the task thread and the interrupt thread. For example, you want to receive some characters that are coming from a serial port. You really cannot do any processing in the task thread until those characters are available. So, you invoke the driver to request a character. However, none are currently available so you want to suspend. You need a mechanism to do this. To go one step further, when the character is available, you need the task to be scheduled. You need a mechanism to do this also. The same scenario applies to sending characters when the transmit buffer is full. You may want your task to suspend until there is room in the buffer. Then, when room becomes available, you want to be scheduled so the rest of the characters can be sent.

### Program Logic

To see how the parts of a driver fit together, let's step through a simplistic example. This example is described in pseudo-C code to illustrate the main logic and does not relate to any specific RTOS.

In the applications program, we want to write a string of text to an output device, so we call a driver's `OutputCharacter()` function repeatedly:

```
char str[] = "Hello World!";
char *sptr = str;
while (*sptr)
    OutputCharacter(*sptr++);
```

The driver `OutputCharacter()` function is run as part of the task and looks something like this:

```
void OutputCharacter(char c)
{
    if (PortReadyFlag != TRUE)
    {
          AddToPortWaitList(ThisTaskID);
          SuspendTask();
    }
    PORTOUTPUT = c;
}
```

The code checks whether it can output the character now. If it cannot, it sets up a mechanism so that the ISR can alert the task when the port is ready and goes to sleep until the port is available and then does the output.

The ISR for this write port is the other part of the driver:

```
interrupt void WritePort(void)
{
    TASKID tid;

    PortReadyFlag = TRUE;
    tid = GetFromPortWaitList();
    if (tid != NULL)
    {
         PortReadyFlag = FALSE;
         ResumeTask(tid);
         RunScheduler();
    }
}
```

The ISR checks whether any task is waiting for the port to become available. If there is one, it wakes up the task.

The design of this simplistic example driver assumes that the device can accept a character from any task at any time—there is no control of "ownership" of the port, just a list of tasks pending upon it. This alternate logic would be straightforward to implement.

### Conclusions

Historically, writing device drivers has been an arduous task. This stems from the fact that many real-time operating system environments are closed and there's an inclination to use UNIX-like facilities to manage device drivers. In an RTOS designed for deeply embedded applications, a much more open approach is necessary. No nasty tables should have to be maintained or special hoops that have to be jumped through—just simple templates that make creating custom device drivers rather less of a chore.

## 7.7 Scheduling Algorithms and Priority Inversion

*For many embedded applications, even those that are truly real time, a deep understanding of how an operating system functions is not required. When it is, the topic can quickly become very complex. This article, based upon a piece by James Ready (now with Montavista) in* NewBits *in 1996, introduces some of the concepts. (CW)*

### Introduction

A simple way to define a real-time system is one where the right results must be delivered on time. The objective of a software engineer writing a real-time application is to design the software so that the right results are delivered on time. In this article, we will explore the issues surrounding the development of a real-time application and discuss the available theory and technology that can facilitate the software engineer in building such a program successfully.

### Real-Time Requirements

The development of a real-time application is essentially an exercise in resource allocation, and in the reduced case, the resource to be allocated is CPU time. For a given application, with a given workload, on a given processor, the trick is to find some ordering of execution such that all the activities that the software must perform are completed within the deadlines associated with each activity. The set of rules with which the activities are ordered for execution is called a *scheduling algorithm*.

Historically, the development of these rules has been done heuristically and validated by (hopefully) exhaustive testing. Thus, the development of real-time systems was often more an "art" and less of an engineering discipline. However, the United States Department of Defense (DOD) and other organizations who used real-time systems had a considerable interest in seeing that the development of real-time systems become far more of a science and that some scientific reasoning could be applied to their development. Consequently, over the past twenty-five years, considerable funding of research has been invested in the development of real-time scheduling algorithms.

### Scheduling Algorithms

A number of scheduling algorithms have been developed that allow precise statements to be made about the timing correctness of the real-time application. Typically, an analysis can be made that will indicate that for a given type of application, characterized by its workload, on a CPU of a known capacity, the deadlines will be met completely, or a deadline or deadlines will be missed at some time. As is typical in an emerging technology, only the simplest kinds of applications can be scheduled completely, and scheduling algorithms for more complex and dynamic applications remain research studies. What follows is a brief summary of two well-known scheduling algorithms and the type of workload that they can handle successfully.

### Rate Monotonic

In applications where the workload consists of a set of periodic tasks each with fixed-length execution times, the Rate Monotonic Scheduling (RMS) algorithm can guarantee schedulability. The RMS algorithm simply says that the more frequently a task runs (the higher its frequency), the higher its priority should be. Obviously the most frequent task has the highest priority in the system. If an application is naturally completely periodic or can be made so, then the developer is in luck because the RMS algorithm can be employed and a provably correct assertion can be made about the application meeting its deadlines.

### Deadline Driven

For applications with a mix of periodic and aperiodic tasks, or where the execution length of a task may vary with time, the Deadline Drive algorithm can be used. Here the criterion for scheduling the next task to run is based on finding the task with the earliest deadline. Upon completion, the next earliest deadline task is selected and run.

## Implications for Operating Systems and Applications

The discussion of scheduling algorithms so far has centered upon a fairly high-level view of the application software, and no particular details of the runtime environment have been assumed. For example, we have not indicated whether an operating system supports the application.

But, just as the time behavior of the application must be understood to select the right scheduling algorithm, if we do employ an operating system, its own time-dependent behavior also enters into the equation. For example, if the underlying operations of the operating system are unpredictable because of some memory allocation/deallocation strategy, it may well be impossible to reliably schedule the application. In other words, the operating system behavior violates the predictability required by the scheduling algorithm we hoped to use.

It turns out that the problem of operating system interference is quite real, even with an operating system that claims to be real time. This interference can occur within the operating system itself, or be induced at the application level by an inadequate capability in the system call set of the operating system.

There is a phenomenon that can occur within an application, called *priority inversion*, which serves as a good example of how a lack of care in developing an application may compromise the use of the powerful scheduling algorithms we discussed previously.

### The Priority Inversion Problem

The conditions under which priority inversion can occur are quite commonly found in both real-time applications and within operating systems. We will discuss here the occurrence of priority inversion at the application level. The conditions are:

- Concurrent tasks in the system share a resource that is protected by a blocking synchronization primitive such as a semaphore or exchange.
- At least one intermediate priority task exists between any two tasks that share a resource.

The following scenario illustrates how priority inversion occurs (see Figure 7.11):

1. A low-priority task makes a memory allocation call, which in turn uses a semaphore to protect a shared data structure (see part A in Figure 7.11). It takes the semaphore and enters the critical section (illustrated in part B of Figure 7.11).
2. An interrupt occurs that enables a high-priority task (part C). The kernel switches control to the higher-priority task (part D).
3. The high-priority task now makes a memory allocation call (part E) and attempts to enter the critical section. Since the low-priority task currently holds the semaphore, the higher-priority task is suspended (blocked) on the semaphore, and the low-priority task runs again (part F).
4. An interrupt occurs (part G), and a medium-priority task becomes ready to run. The medium-priority task runs (part H) because it is higher in priority than the lower-priority task holding the semaphore and because the high-priority task is suspended on the semaphore.

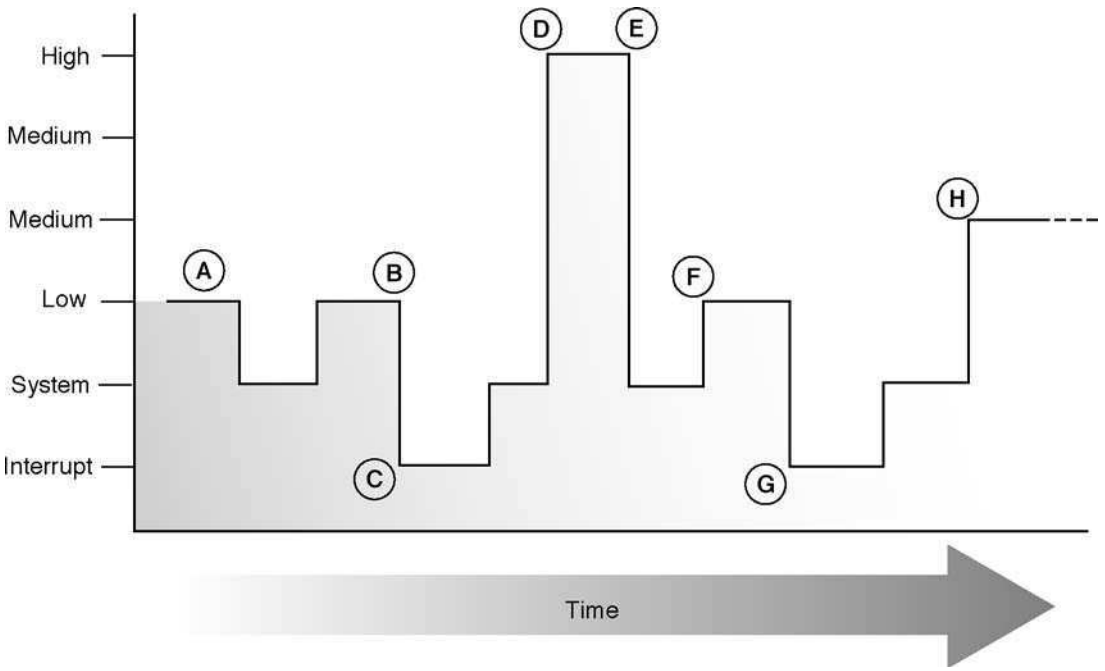



*Figure 7.11: Priority inversion*

5. At this point, *all* tasks of a priority higher than the low-priority task that become ready to run will do so at the expense of the low-priority task (which is how it should be), but they also run at the expense of the high-priority task, which is held up as long as the low-priority task is held up.

In effect, the high-priority task's priority has become lowered or "inverted" to match that of the low-priority task—hence the term *priority inversion.*

If the operating system does not offer any specific facilities to address this problem (priority inheritance), there are three possible solutions:

- Avoid sharing resources between tasks of differing priorities.
- If the operating system allows, turn off preemption of the low-priority task for the critical section during which it holds the semaphore. (This, of course, inhibits multithreading, which is normally undesirable.)
- The low-priority task could raise its own priority while in the critical section (priority inheritance done "by hand").

### Conclusions

The understanding of scheduling algorithms is of fundamental significance to developers of hard real-time applications. In some cases, application developers are required by certification agencies to employ the RMS algorithm because of the mission-critical nature of the applications. Even if the use of these scheduling algorithms is not mandated, they provide a sound basis upon which to build real-time applications such as those found in communications and control systems.

## 7.8 Time versus Priority Scheduling

*This article is based upon a white paper by John Schneider, which he wrote in response to some of the "FUD" (fear, uncertainty, and doubt) marketing of various RTOS vendors. (CW)*

### RTOS Scheduling

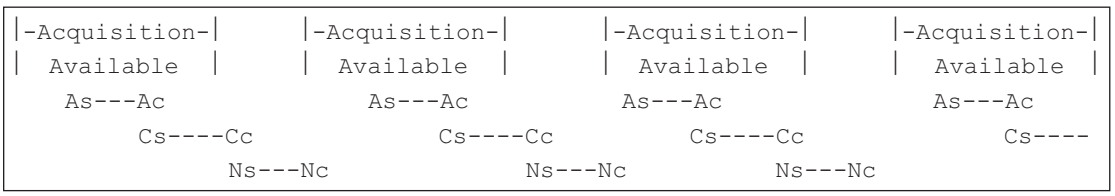
With time-domain bounding, every domain is given a guaranteed time allocation no matter what the priority. If a higher-priority domain runs out of time, it must wait for its next time slice. In a hard real-time system, time-domain bounding also requires the user to plan CPU allocation for the worst-case scenario, with no opportunity to take advantage of the less-than-worst-case scenario.

Most true real-time operating systems use a priority based, preemptive scheduler instead of time-domain bounding. This guarantees that higher priority tasks run when needed. The worst-case example for a priority-based scheduler is when a higher priority thread consumes a large, contiguous block of CPU time. This scenario could essentially lock out lower priority threads causing starvation. In this case, two approaches can be used. The first option is to have the threads at the same priority as the time-consuming thread and allow the scheduler to preempt the time-consuming task on regular intervals to service other threads based on round-robin scheduling. A better option is to have the time-consuming task relinquish the processor to lower priority threads. This can be accomplished by using a *sleep* command at appropriate, noncritical intervals in the code. The *sleep* command will allow lower priority threads to run for a specified amount of time. This implementation allows the user to decide when and where the lower priority threads get access.

The following example of a network connected data acquisition system shows the differences between the two scheduling schemes.

### Perfect World

In a perfect world, a network connected data acquisition system would operate in a manner such that none of the threads interfere with each other and the system is always available when a thread is needed. A sample would never be missed, network access would only happen after the computation is complete, and calculations would always be the same length as CPU cycles. This is what it would look like in a simple timeline:

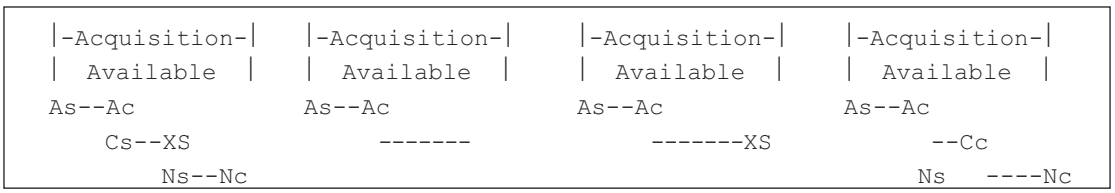


Key

- As: Data acquisition start
- Ac: Data acquisition complete
- Cs: Data computation start
- Cc: Data computation complete
- Ns: Network access start
- Nc: Network access complete

### Real World with Priority Scheduling

In the real world, network access is often asynchronous to the rest of the system and can use up variable amounts of CPU cycles (this is largely due to the nondeterministic nature of TCP/IP rebroadcast schemes and variations in data packet size being sent or received), computations are never identical, and so on. These deviations from an ideal case can be overcome by the use of a priority-based scheduler. In the following case, the data computation has strategically placed explicit sleep calls (noted by XS) in noncritical places in the code. This allows the lowest-priority thread to run until the highest priority thread preempts. When the highest-priority task is complete, the system reschedules based on priority and returns to the data computation thread.




#### *Program Priority*

A typical distribution of task priorities is:

Highest priority: Data acquisition

Medium priority: Data computation

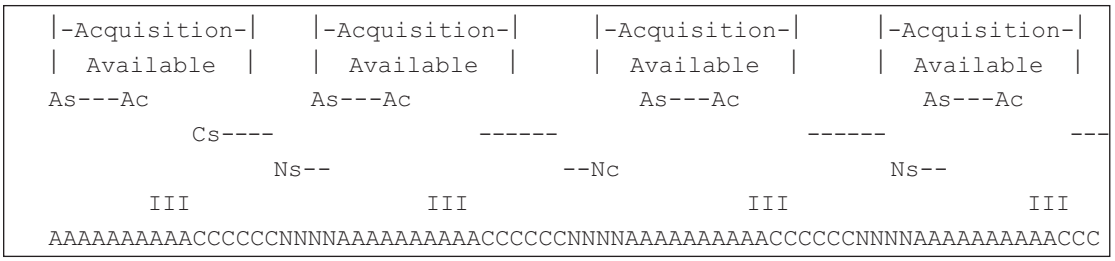
Low priority: Network access

- XS: Explicit sleep
- I: Idle

### Time Domain Bounded Without Relinquish

The timeline that follows shows the same time sequence but split up in a time-domain–bounded system without the ability to relinquish unused portions of the allocated time slice. In this example, we've allocated 50% of the time for the most important task, data acquisition; 30% of the time to the data computation task; and 20% to network access. You can see that the computation never gets completed, which is due to the large block of idle time at the end of the acquisition stage. In our example, we had to

allocate this extra block of time due to the worst-case response time of the hardware. This leaves the user having to allocate for the worst-case scenario and never being able to take advantage of the extra time that results in the lower response times from our system.




A: Data acquisition domain—50%

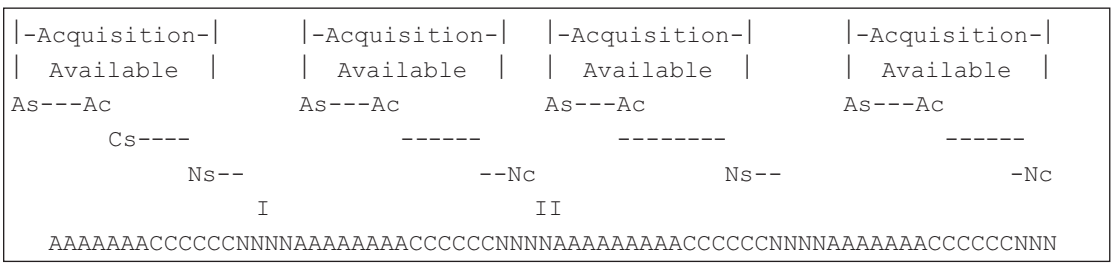
C: Data computation domain—30%

N: Network access domain—20%

I: System idle time

### Time Domain Bounded with Relinquish

Even in a time-domain–bounded system that has the ability to relinquish CPU resources when finished, the lag time shifts to the front of the acquisition while the acquisition task is waiting for the acquisition to be ready. Since the other tasks have used their allotment, they can't run. This leads to wasted time. The timeline shows the identical timeline as the one shown previously but with the ability to relinquish.




### Conclusions

On first sight, a scheme that allocates time in a very ordered fashion may appear to be the most deterministic and maybe the most applicable to a hard real-time system. However, as discussed previously, even in a simple system, time-domain scheduling is unable to cope with changing demands upon CPU resources.

## 7.9 An Embedded File System

*Software engineers, who are new to embedded systems are often rather disconcerted by the lack of a "supporting" environment." Even if there is an operating system (and there may not be one at all), typically it does not have all the capabilities of Windows or UNIX. One of the obvious facilities that are lacking is a file system—normally code is run out of some kind of read-only memory and the only data store is RAM. A seasoned embedded developer would take a different view and be shocked if he were presented with an RTOS that required a file system—he would consider it unbearable overhead. As with many things in embedded systems, different users have different needs, and a file system is a commonly requested option. This article is based upon an* Accelerated Technology *white paper. (CW)*

Embedded systems continue to grow and mature at an astounding rate. Rarely will an embedded system remain stagnant. As time goes on, systems that were simply a CPU with a few peripherals add more peripherals, more powerful processors, and eventually features such as offline storage and networking. In many ways, the evolution of embedded systems has caused a convergence of desktop technology with real-time technology. However, due to the price performance advantages of non-desktop-oriented CPUs and the lack of standardization in the embedded industry, the base software for embedded systems has not kept pace with that of desktop systems. While there are millions of prospective users for desktop software packages, the numbers pale when it comes to embedded software.

One area in which this disparity is particularly evident is offline storage. Desktop systems are widely known for their capabilities in this area. Even though you may be able to embed MS-DOS or UNIX in your embedded system, they are generally too large and cumbersome, or they do not support the CPU that you choose. So, to get the file system advantages of a commercial operating system such as one of these, you must either write the capabilities that you require on your own, or you can purchase an off-the-shelf product from an embedded software provider, usually an embedded real-time operating system provider.

In two situations, an embedded system may use a file system:

- To provide a static store of information so that the embedded system could quickly reconfigure at startup or after a downtime.
- To provide a way to transfer data from an embedded system to a desktop system for data reduction and analysis.

The first requirement can basically be satisfied with a simple file storage facility. In this way, data can be written to an offline device in a flat file structure. However, this type of solution generally does not have the capacity to improve easily with the growing needs of an embedded system.

The second requirement—to transfer data to another system—requires a more sophisticated file structure that is compatible with common desktop platforms. An MS-DOS compatible solution is generally considered most appropriate. It easily handles the first requirement and by its nature provides a complete solution for the second requirement.

### Requirements of an Embedded File System

Aside from meeting the two requirements just mentioned, an embedded file system must permit a real-time system to operate in real time, it must have minimal impact on data and processing requirements, and it must allow multiple tasks to access file information.

Critical activities must be completed in a timely and reliable manner in real-time embedded systems. If they are not, catastrophic results can occur. In a somewhat ludicrous yet illustrative example, if a machine that is responsible for pumping blood through a patient and saving statistical information to a file causes the blood to stop pumping, the result could be fatal. Therefore, the file system must be integrated into the real-time embedded system in such a way that file system activity can be relegated to a low priority when necessary.

Due to the fact that every kilobyte of memory and every instruction of processing time in an embedded system can have a drastic affect on cost, memory, and processing allocation is critical. An embedded file system must use memory wisely and process information quickly.

Most embedded systems, by their very nature, must be multi-tasking. The requirement to allocate CPU time to individual processing requirements based on priority can be very important; in many cases, for an embedded system to operate correctly, it is essential. An embedded file system must be accessible by multiple tasks simultaneously. This means, since a file system is required to have global data structures, they must be protected from reentrancy problems.

### MS-DOS File System Overview

The MS-DOS file system is often referred to as a FAT file system. A FAT is the File Allocation Table, which coordinates access to data on the disk. In addition to the FAT, the file system also includes a boot sector. The boot sector contains information about the disk including its size, the size of the FAT entries (12-, 16-, or 32-bit), and other useful information. FAT12 entries are for small disk media, such as floppy diskettes. FAT16 entries are for medium-sized disks, up to around 2 G. FAT32 entries were generated to accommodate today's large multi-gigabyte disks.

The FAT is made up of a series of pointers that are associated with each cluster on the disk. A cluster is a physical sector or group of sectors where data is stored. The clusters are linked together to make up a file. By reading the clusters associated with a set of FAT entries, you can construct the data in any file.

An embedded file system maintains the FAT on an ongoing basis. Each time a file is written or appended, the FAT is updated in memory. When a file is closed or flushed, the FAT is written to the disk. The FAT can be buffered completely or cached depending on your memory requirements. If it is cached, only portions of the FAT are kept in memory. An embedded file system would generally include facilities to enable you to indicate how large the in-memory FAT will be for any of the disks in your system.

### Long Filenames

With the larger media becoming more widespread in the embedded industry, true MS-DOS compatible file systems must be able to support any new MS-DOS features. Among the most prominent is the need to recognize filenames beyond the "eight-dot-three" format. MS-DOS now allows filenames to contain up to 255 characters, to provide more descriptive file naming. Long filenames are also stored and displayed in a truncated format, to accommodate disk media with earlier MS-DOS versions that do not properly display them in long format. In the truncated format, all characters after the first six characters are truncated, a tilde (~) is added, and a number is assigned preceding the extension dot. For example, the filename, `Encyclopedia Data.doc` could be displayed in its entirety or in the truncated form `Encycl~1.doc`.

### Formatting

When you format a disk, or disk partition, you are writing a boot block and an empty FAT. A file system will probably provide standard functions for formatting floppy (360 K, 720 K, 1.2 M, and 1.44 M) as well as hard disks. To format a disk, you fill in a structure with the disk's characteristics and issue a service call to the embedded file system, which then writes the appropriate information to the correct location on the disk.

### Partitioning

Disk partitioning is not the same as disk formatting. A partition is a physical subdivision of the directory database stored on and managed by a specific directory server. Partitioning is common on large disk media, changing the appearance of one large physical disk into several smaller logical disks. Disk partitioning capabilities are likely to be provided via the IDE driver.

### Devices

Standard device drivers for an embedded file system are likely to include floppy, fixed, flash, and RAM disk drivers. Device drivers require four functions: open, close, perform input/output, and control input/output.

## 7.10 OSEK—An RTOS Standard

*Although the OSEK standard had its origins in the European automotive industry, it is now gaining much wider acceptance. It is finding worldwide application in various transportation applications. There is also significant interest from developers in other safety-critical areas, such as medical electronics. The standard is well worth considering wherever the static architecture and simplicity of OSEK is a benefit. This article provides a brief introduction and is based upon an* Accelerated Technology *white paper by Zeeshan Altaf. (CW)*

### About OSEK

OSEK/VDX is a joint project initiated by some major players in the automotive industry. It aims for an industry standard of an open-ended architecture for distributed units in vehicles. An RTOS, software interfaces, and functions for communication and network management tasks are thus jointly specified.

The term "OSEK" means "**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug" (open systems and the corresponding interfaces for automotive electronics). This particular standard originated in Germany and included BMW, Bosch, Daimler-Benz, Opel, Siemens, VW, and the University of Karlsruhe. The term "VDX" means "**V**ehicle **D**istributed e**X**ecutive." This standard originated in France and included PSA and Renault. The functionality of the OSEK operating system was harmonized with VDX. For simplicity, the truncated term "OSEK" is generally used instead of OSEK/VDX.

The key motivations behind the development of the OSEK specification were high recurrent expenses in development, irregular management of non-application-related aspects of control unit software, and the incompatibility of control units made by different manufacturers due to different software interfaces and protocols.

The OSEK architecture comprises three primary modules:

- **Operating system (OS):** A real-time executive for electronic control unit (ECU) software and the basis for the other two OSEK modules.
- **Communication (COM):** The data exchange within and between control units.
- **Network Management (NM):** Configuration determination and monitoring.

Figure 7.12 shows a graphical depiction of the OSEK architecture.

### OSEK Requirements

The specification for the OSEK operating system includes strict real-time requirements. Memory resource usage must be minimized, and the operating system must be scalable, reliable, ROMable, and cost-sensitive. The software must be developed in a manner to promote portability. It must serve as the basis for software integration from various manufacturers. Finally, the configuration, including scalability and scheduling policies, must be static.

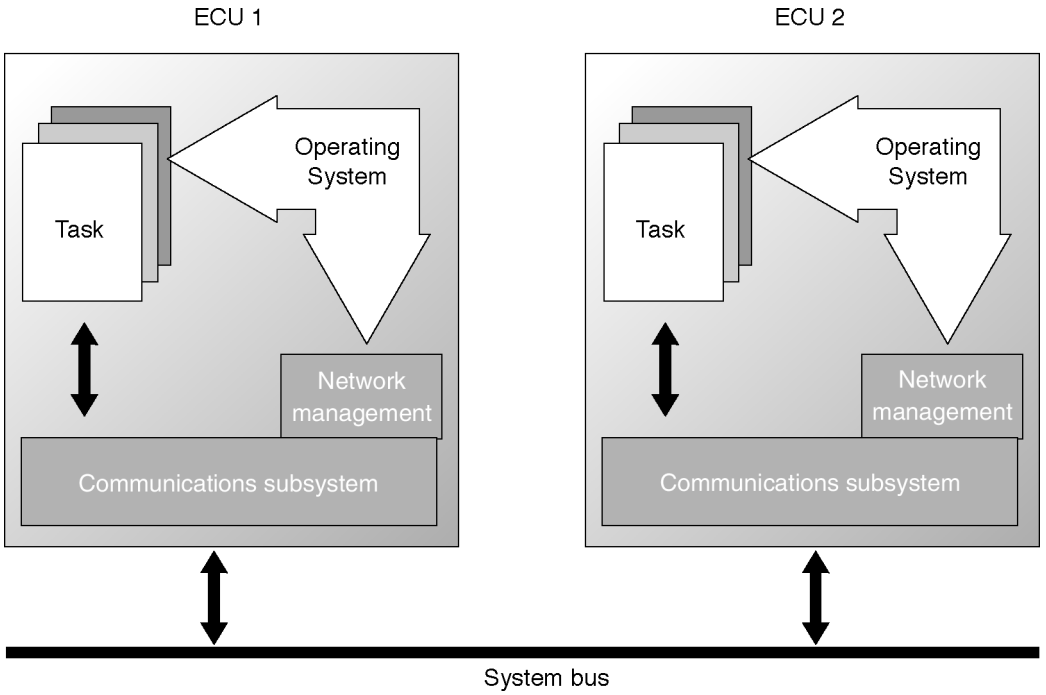



*Figure 7.12: OSEK architecture*

The goals of the OSEK project are to support the portability and reusability of the application software by providing specifications for interfaces that are abstract and as application-independent as possible. Additionally, the specifications of a user interface should be independent of supplied hardware and network applications. The architecture will also be more efficiently designed because of the guidelines. The functionality shall be configurable and scalable to enable optimal adjustment of the architecture to the application in question. Finally, verification of the functionality and implementation of prototypes in selected pilot projects shall be available.

## OSEK Tasks

### *Task Types*

Two task types are defined in OSEK:

- **Basic** tasks release the processor only if:
  - They are being terminated.
  - The system is executing higher priority tasks.
  - An interrupt occurs, and the processor switches to an ISR.
- **Extended** tasks can additionally have a waiting state when they are using the event mechanism for synchronization.

### Conformance Classes

The conformance classes provide convenient groups of operating system features for easier understanding and discussion of OSEK. The conformance classes allow partial implementations, which may be certified as OSEK-compliant, along predefined lines. They also create an upgrade path from classes of less functionality to classes of higher functionality with no changes to the application using OSEK-related features. Four conformance classes are defined for OSEK functionality:

- **BCC1:** Only basic tasks, limited to one request per task and one task per priority (all tasks have different priorities).
- **BCC2:** BCC1 functionality, plus the possibility of more than one task per priority and enabling the multiple requesting of task activation.
- **ECC1:** BCC1 functionality plus extended tasks.
- **ECC2:** BCC2 functionality plus extended tasks: multiple requesting of task activation only for basic tasks allowed.

### Scheduling Policy

Three scheduling methods are defined by OSEK:

- **Nonpreemptive:** A task switch is only performed via one of a selection of explicitly defined system services. Nonpreemptive scheduling imposes particular constraints on the possible timing requirements of tasks. Specifically, the nonpreemptable section of a running task with lower priority delays the start of a task with higher priority up to the next point of rescheduling.
- **Full preemptive:** A task that is presently running may be rescheduled at any instruction by the occurrence of trigger conditions preset by the operating system. Restrictions are related to the increased memory space required for saving the context and the enhanced complexity of features necessary for synchronization between tasks. Access to data, which is used jointly with other tasks, must be synchronized.
- **Mixed preemptive:** A mixture of nonpreemptively and full preemptively scheduled tasks

### Synchronization Mechanisms

Two synchronization mechanisms are defined by OSEK:

- **Resource management:** Controls access to jointly used logic resources or devices.
- **Event management:** Event management for task synchronization—allowed only for extended tasks.

## Alarms

Two types of alarms are defined by OSEK:

- **Relative:** Count values are defined relative to the actual counter value.
- **Absolute:** Count values are defined as absolute values. The count values can be defined dynamically at runtime (variant alarms) or statically at compile time (nonvariant alarms).

### Error Treatment

OSEK implementations normally utilize user-definable hook routines for error handling and debugging. Scalability of error checking includes an extended status for the development phase and a standard status for the production phase, which is generally implemented by answering the OSEK API with macros, which may be expanded in a variety of ways.

# 8

# Networking

*There are a variety of add-on products for real-time operating systems. These are commonly called "layered products" or "middleware." Given that an increasing number of embedded systems are connected to the Internet or to private networks, it is unsurprising that networking products dominate the middleware market. In this chapter, the articles address a wide range of topics all under the heading of Networking, which range from broad introductions to detailed discussion of protocols.*

## 8.1 What's Wi-Fi?

*Wireless networking in homes and offices has become very popular in just a few years. Such networks are constructed from a number of embedded systems. So, of course, interest in incorporating Wi-Fi into embedded devices has soared. In early 2004, I wrote a piece for* NewBits *to broadly explain what 802.11 was all about. That was the basis for this article. (CW)*

When I was a little kid, back in the early 1960s, we had a "trannie"—a transistor radio. This was the cool gadget to have back then—the MP3 player of its day. After all, the transistor was the latest technology, having only been invented some 15 years before. We don't seem to have such a long "lab-to-living room" cycle nowadays! My grandmother didn't have a trannie, but in the corner of her kitchen, she had something much more mysterious: a big wooden box with dials and displays that lit up when you switched it on. Nothing else happened when you applied the power because it needed to "warm up" (I guess we would say "boot up" nowadays—although that term was understood then, it wasn't used in polite company). It was a vacuum tube (in England we say "valve") radio. But my granny called it her "wireless."

So my perception of this term was that it was a logical—receiving a signal with no wires—but rather old-fashioned—dare I say it—antiquated term. It never occurred to me as odd that this word, while logical, did not signify the successor to some kind of "wired radio." Neither did I suspect that this term would take on a whole new lease in life in my adulthood.

Today, we use the word "wireless" to refer to a whole slew of technologies, where one requiring no wires has replaced a wired approach. We are not limited to radio. Many other parts of the electromagnetic spectrum are used: light, infrared, microwave, and so forth. Maybe we are referring to voice or multimedia transmission (see the sidebar "Why Were Phones First?" later in this article), but often the terminology is used in relation to the transfer of data. It is this latter area, wireless data communications, that I will concentrate upon here.

### Wireless Datacom

There is not a single technology that can be labeled "wireless datacom." There are a whole bunch of them. They can be categorized and characterized by their operating range.

### *Wireless Personal Area Network (WPAN)*

This kind of network would typically encompass just a few meters around an individual. Infrared communications (IRDA) and the increasingly popular Bluetooth are common WPAN implementation technologies. ZigBee is emerging as another WPAN option.

### *Wireless Local Area Network (WLAN)*

In this case, the range of the network is something like a building. It could be just part of a building or could encompass a small group of buildings. A shop or restaurant may be covered using a WLAN. Typically such an area is termed a "wireless hot spot." The most likely implementation technology is one of the IEEE 820.11 families, which are described in the sections that follow.

### *Wireless Metropolitan Area Network (WMAN)*

This kind of network is similar in functionality to a WLAN but covers an area the size of a city.

### *Cellular Network*

The voice-oriented cellular networks are gradually becoming more capable in terms of handling data traffic. GSM is very widely used, for example, with GPRS and EDGE available for high-speed data.

These four families can, for the most part, coexist. Having said this, the WMAN domain is being encroached upon from both sides—bigger WLANs and cost-effective cellular services. Some further comments on the comparison between Wi-Fi and Bluetooth follow later.

It is interesting to survey the kitchen table in front of me. There is my GSM cell phone, which uses GPRS to reach the Internet. The phone connects to my Palm using Bluetooth and can connect to either the Palm or my laptop via IRDA. My laptop is connected to the ADSL gateway two rooms away using Wi-Fi, and this last technology is the one that we will now focus upon.

## IEEE 802.11

The 802.11 family of standards addresses WLANs. The term "Wi-Fi" (wireless fidelity) was defined by the Wireless Ethernet Compatibility Alliance (WECA) to describe them. This term has since been widely used and misused. The 802.11 family includes many standards, and, because of its success, more standards are continuously being added. There is a real alphabet soup of 802.11 specifications, but some stand out as worthy of particular attention. Three of them, 802.11a, 802.11b, and 802.11g, are physical layer standards.

### *802.11b*

This standard was the first generation, offering modest speed (11 Mbps). Equipment employing this standard is very widely used. The most widely deployed Wi-Fi standard

today is 802.11b. It utilizes the 2.4 GHz band of the radio spectrum, which is unregulated in the United States. As a result, it shares this bandwidth and is susceptible to interference by devices that include, but are not limited to, cordless phones, microwave ovens, and Bluetooth-enabled devices. The maximum throughput is 11 Mbps, but 6 Mbps is a better estimate of what can be achieved in a real-world deployment.

#### *802.11a*

Confusingly this is the second-generation standard, offering a higher data rate (54 Mbps) on a different frequency (5.4 GHz). The maximum throughput using 802.11a is 54 Mbps, but as with 802.11b, the real throughput achieved will be significantly less than this, around 33 Mbps.

#### *802.11g*

The third-generation standard aims to combine the good things of the previous ones. It provides the speed of 802.11a on the same frequency band as 802.11b. It is backward compatible with 802.11b, so devices that support this standard will be able to communicate with both 802.11b- and 802.11g-enabled devices.

#### *802.11d*

This standard is designed to promote the worldwide use of 802.11. It will allow devices to communicate information about permissible radio channels and acceptable power levels. The 802.11 standards cannot legally operate in some nations. 802.11d adds features and restrictions to allow devices to operate within the local rules. The use of 802.11 is lagging behind North America in those countries whose physical layer radio requirements are different. This standard will obviate the need for travelers to carry and vendors to make country-specific WLAN cards.

#### *802.11i*

This standard provides an alternative to Wired Equivalent Privacy (WEP), which has security holes. Applying the a, b, and g standards, it will provide authentication and encryption procedures.

### 802.11 Basics

The place of 802.11 in the protocol scheme is best illustrated by looking at the mapping onto the ISO/OSI seven-layer model (see Table 8.1). As can be seen from this table, the 802.11 standards address the bottom two layers, Data Link and Physical. The higher layers are satisfied by the standard Internet protocols. In a real system, there may, of course, be multiple Data Link and Physical layers, representing multiple 802.11 variants, other 802 standards, or other (nonwireless) networking technologies.

Table 8.1: ISO/OSI Seven-Layer Model

| OSI Layer | Protocol |
|---|---|
| Application | HTTP, FTP, POP3/SMTP |
| Presentation | DNS, LDAP |
| Session | |
| Transport | UDP, TCP, ICMP, RSVP |
| Network | |
| Data link | Logical link control (LLC)<br>Medium Access Control (MAC) |
| Physical | Physical (PHY) |

In a Wi-Fi network there are broadly two types of devices: station devices (STA), which are users of the network, and access points (AP), which are wireless network hubs. Clearly each of these will require different software support.

Wi-Fi devices can function in one of two modes:

- **Ad hoc mode:** Where two station devices are directly linked.
- **Infrastructure mode:** The usual way of working, where a station device is linked to an access point.

### Wi-Fi and Bluetooth

There is some confusion between the functionality offered by Wi-Fi and by Bluetooth. In reality, they are quite complementary and offer different functionality—each has strengths and weaknesses in different situations.

#### *Range*

Bluetooth is intended to be used over about a few meters; Wi-Fi is intended for areas covering tens of meters.

#### *Topology*

Bluetooth is strictly point-to-point/master-slave and, as such, is very analogous to USB. Although Wi-Fi can be used in ad hoc mode, its real strength is when it is used in infrastructure mode, which is much more analogous to Ethernet, where multiple devices may be connected together.

#### *Power Consumption*

In designing embedded devices, which are wireless enabled, power consumption (= battery drain) is frequently an issue. This is a downside of Wi-Fi, where frequent data exchanges are needed to maintain network integrity. Wireless transmission is power hungry. Bluetooth transmits only when there is really data to send.

### Where Next?

Today we are only seeing the very start of what wireless networking will provide in the future. Connecting together boxes that are recognizable as computers is useful but hardly scratches the surface of the potential. There has been a trend for embedded devices to become "connected"—the next trend is for them to be Wi-Fi enabled. So what about:

- A car alarm system that seamlessly integrates into the security system of the house, office, or parking lot?
- A digital camera that is always visible to nearby computers and immediately uploads and archives its images?
- Printers that you can locate anywhere you want and that can service numerous PC users?
- A network in your house that distributes multimedia content so that the TV can go anywhere there is a power outlet, and you can watch "Friends" on your Palm?

With rapidly decreasing prices on Wi-Fi chips sets and the ready availability of cost-effective, off-the-shelf software solutions, all of these ideas and many more will soon become reality. The rest is up to you.

#### Why Were Phones First?

In the early 1980s, several things came together. Parts of the radio spectrum became available at the same time as the possibility for cheap electronics to exploit these frequencies. All that was needed to make this commercially interesting was a pervasive, ubiquitous technology to replace or augment.

That technology was the telephone. Having been around for about a century, phones were everywhere in the Western world—every home, every office, every street corner. The only limitation to their use was the wire to the jack in the wall. Sure, the wires got longer—the TV or movie image of someone wandering around a room with a phone on a long lead is clear in my mind. But they were still tethered and needed to be set free. Wireless (radio) technology was the answer. First there were domestic cordless phones, then the first cellular phones. Many of these were in cars—hand-held units were somewhat cumbersome. The initial analog technology (first generation—1G) gave way to digital (2G). Now, more advanced digital (2.5G) is common, and the next generation (3G) is all set. In several countries, cell phones already outnumber fixed phones. In developing countries, they are installing cellular infrastructure from day one.

It is a shame that the language has not evolved so well. "Mobile phone" and "cell phone" are very awkward terms, and "cell" is just plain wrong. In Germany, "hand-held phone" is abbreviated to "handy" (yes, an English word). Maybe this terminology could be used more widely.

## 8.2 Who Needs a Web Server?

*It is becoming quite common for web servers—or, more correctly, HTTP servers—to be incorporated into routers, gateways, and other networking equipment. There is every reason to consider doing this for a multitude of other types of embedded systems. In a piece written for the "Nucleus Reactor" newsletter in 2000, Neil Henderson (CEO of Accelerated Technology) gave a very good outline of what was possible and how to get started. This provided the basis for this article. It may be interesting to compare the use of HTTP with SNMP because in this context, their objectives are quite similar. Take a look at the article "Introduction to SNMP" later in this chapter. (CW)*

### Introduction

You may view web servers in the same manner I did before I understood what they could do in an embedded system. In my mind, web servers were located on machines with huge disk drive capacity and served up pages to web browsers. Well, huge disk drives are not necessary, and web servers can do much more than just serve up web pages!

With an embedded web server you can, of course, serve up pages. But did you know you can use a web server to provide an interactive user interface for your embedded system? Did you realize that you could program that interface once and be able to use it independent of the type of machine your user has? Did you further know that you could monitor and control your embedded system from any web browser with very little programming? All of these things are made possible by this very small, very efficient, and very powerful piece of software.

In this paper I will provide information that you will most likely be able to use on your embedded system. All you need is a TCP/IP networking stack, an embedded HTTP server (i.e., web server), and a little imagination. So, let's get started.

### Three Primary Capabilities

Web servers are capable of performing three basic functions:

- Serve web pages to a web browser
- Monitor the device within which they are embedded
- Control the device within which they are embedded

We will be examining these functions in more detail in the remainder of this paper. Here, I will give you a brief introduction to each of these functions so that you can better understand the sections that follow.

#### *Serve Web Pages to a Web Browser*

This is the most fundamental capability of a web server. The web server waits on the network for a web browser to connect. Once connected, the web browser provides a filename to the web server, and the web server downloads that page to the web browser.

In the most basic case, the web server can download simple HTML files (simple because there are no inherent capabilities other than to show information) from within its file system to the web browser. This feature is ideal for downloading user documentation from the embedded system so that it can be used in a web browser.

A more sophisticated and extremely powerful capability is for the web server to download Java programs or applets (encapsulated in an HTML file) to the web browser. Once loaded in the web browser, the Java program or applet executes and can communicate with the target (that contains the web server) using the TCP/IP protocol. The power of this capability lies in the ability to:

- Support legacy applications (existing TCP/IP applications that presently communicate with a Java application that runs in a browser rather than writing proprietary applications for different desktop operating systems).
- Write sophisticated TCP/IP-based applications between a host and server where you control both sides regardless of where the host is running.

#### *Monitor a Device*

Often there is a need to retrieve (i.e., monitor) information about how an embedded system is functioning. Monitoring can range from determining the current pixel resolution of a digital camera to receiving vital signs from a medical device.

By embedding certain commands within an HTML page, dynamic information can be inserted into the HTML stream that is sent to the web browser. As the web server retrieves the file from the file system, it scans the text for special comments. These comments indicate functions to be performed on the target. These functions then format dynamic information into HTML text and include the text into the HTML stream being sent to the web browser.

#### *Control a Device*

HTML has the capability to maintain "forms." If you have ever browsed the web and tried to download something, you probably have seen a form. A form is a collection of "widgets" such as text entry fields, radio buttons, and single-action buttons that can be assembled to collect virtually any type of data.

By constructing an HTML page with a group of widgets, information can be collected from the user in a web browser. That information can then be transmitted to the target and used to adjust or alter its behavior. For example, an HTML page could be constructed to configure a robot arm to move in certain sequences to perform some necessary function (e.g., to bend a piece of sheet metal). This could be done by placing specific text entry boxes in the HTML page that instruct the user to enter a number of specific data points. After being sent to the web server, the data points can then be analyzed by the embedded system's application, validated, and then executed (or, if the data is invalid, to have the user reenter the data) to move the robot arm in the proper directions.

### Web Servers at Work

Once again we will explore the use of the web server based on the three primary capabilities. We will look at the processing that is done on the web server and how information

is supplied both from and to the web browser. I will discuss the ability to serve pages to a web browser. Then, we will progress into the more complex tasks that can be achieved by implementing an embedded web server—namely, using the web server to provide dynamic information to a web browser and using a web server to control your embedded system.

Communication between the web server and the web browser is controlled by HTTP (HyperText Transfer Protocol). HTTP supplies the rules for coordinating the requests for pages to the web server from the web browser and vice versa. The pages are transferred in HTML (HyperText Markup Language) format.

### *Serving Pages*

As discussed previously, the simplest use of the web server is providing HTML pages from the web server to the web browser. This is a straightforward operation where the server maintains a directory structure containing a series of files. The user, from the web browser, specifies the URL (Uniform Resource Locator) that includes the IP address of the web server and the name of the file to be retrieved. The web browser transmits an HTTP packet to the web server with the requested filename. The web server locates the file and sends it to the browser via the HTTP protocol. Finally, the web browser displays the page to the user.

This feature can be used to supply information such as the device's user manual from the embedded system to the user on a web browser. In most web server implementations, the ability to serve pages up to a web browser can be included in an embedded system with little or no coding effort.

### *Using the Web Server to Provide Dynamic Information to a Web Browser*

By manipulating the HTML page that is sent to the web browser, the embedded system employing the web server can supply dynamic information to the user. The web server on the embedded device scans every HTML file that is sent to the web browser. If a certain string is encountered during the scanning process, the web server knows to call a function within the embedded system. The called function then knows how to format the dynamic information in HTML and append it to the buffer being sent to the web browser.

Let's assume, for example, that our embedded system is a router. Let's further assume that we want to display the router's IP address. The complete HTML file to display this information may look something like this:

```
<BODY>The IP Address of the Router is: <!-# IPADDR > </BODY>
```

As the web server scans this HTML, it encounters the `<!-#` symbol, performs a lookup on the string `IPADDR`, and determines that a function `display_IP_addr(Token *env, Request *req)` is to be called.

`display_IP_addr()` may look something like this:

```
/* Create a temporary buffer. */
char ubuf[600];
void display_IP_addr( Token *env, Request *req )
{
      unsigned char *p;
      /* Get the IP address. */
      p = req->ip;
      /* Convert the IP addr to a string and place in ubuf. */
      sprintf(ubuf, "%d.%d.%d.%d", p[0], p[1], p[2], p[3]);
      /* Include the IP string in the buffer on its way to
      the browser. */
      ps_net_write(req, ubuf, (strlen(ubuf)), PLUGINDATA);
}
```

Let's quickly review what we have just done. In the HTML file, we indicate that we want to display the string "The IP Address of the Router is." Additionally, there is a command to display the value of `IPADDR`. It is not evident in what we see here, but the `IPADDR` reference is actually in a table on the target. In the table, `IPADDR` has a corresponding element named `display_IP_addr` that is a pointer to the function call of the same name.

In the code, we assume that the web server has already found the `<!-#>` string and has located the `IPADDR` element in the table. This has resulted in the call to `display_IP_addr()`.

`display_IP_addr()` simply fetches the IP address from the `req` structure, formats it into the easily recognizable four-part IP number and then places the resultant string into the buffer that is on its way to the web browser.

From this simple example we can begin to see the power the web server possesses to transmit dynamic information to a web browser. By using more sophisticated HTML information, elaborate user displays can be created that are exciting and informative.

### *Using the Web Server to Control an Embedded System*

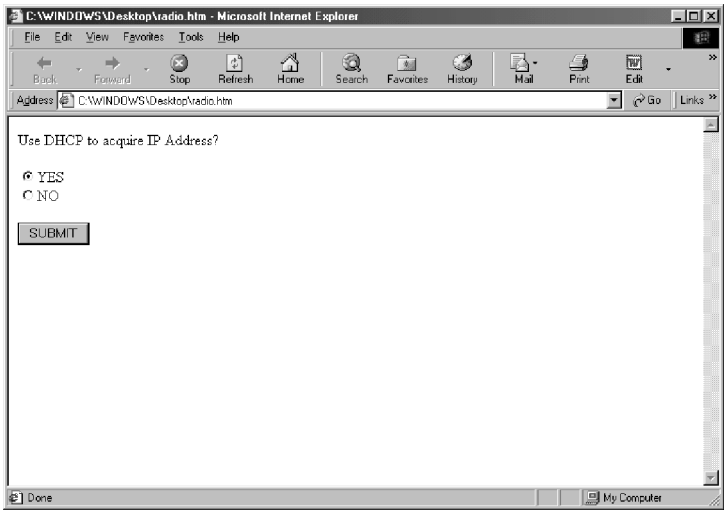
For years, developers of network-enabled products (e.g., printers, routers, bridges) have had to develop multiple programs to remotely configure these devices. Since, in many cases, the products can be used on the Windows, Mac OS, and UNIX operating systems, developers of these types of systems are forced to write applications for all three platforms. Using a web server can reduce this programming effort to developing one or more HTML pages and writing some code for the target. Using this paradigm, the users of the printers, routers, bridges, and so forth simply connect to the device using a web browser. I recently bought a SOHO router that had this capability. An IP address was specified in the

literature that came with the router. I used that IP address in my web browser to communicate to the router's web server. It supplied a full screen of options to configure the router for my particular circumstances. Let's take a minute to look at a simple example of how something similar to this might be accomplished with an HTML file and a little code.

The HTML file will look as follows:

```
<BODY> Use DHCP to acquire IP Address? </BODY>
<br>
<br>
<INPUT TYPE="RADIO" NAME="RADIOB" VALUE="YES" CHECKED>YES
<br>
<INPUT TYPE="RADIO" NAME="RADIOB" VALUE="NO">NO
<br>
<br>
<INPUT TYPE="SUBMIT" VALUE="SUBMIT">
```

The display of this text will look like this:




The code that will be used to process this request may be as follows:

```
int use_DHCP_flag;

int use_DHCP(Token *env, Request *req)
{
```

*Continued*

```
    /* Verify that we are looking at the right "command" */
    if(strcmp(req->pg_args->arg.name,"RADIOB")== 0)
          /* Should we use DHCP? */
          if (strncmp(req->pg_args->arg.value,"YES",3) == 0)
                /* Yes, use DHCP */
                use_DHCP_flag = TRUE;
}
```

Once again, we will review the elements just illustrated. First of all, let's look at the HTML. We have three sections in this file separated by two line breaks. The first section is simply a prompt for the user. The second section is the code necessary to display the radio buttons as shown in the browser display shown previously. The third section serves two functions. First of all, it dictates the drawing of the Submit button. Second it triggers the browser to send the information from this screen to the web server once it has been clicked on. For our discussion, the format of the data in the packet sent from the web browser to the web server is unnecessary. However, as you can see in the preceding function `use_DHCP()`, the information is easily provided to a function that is capable of executing upon the user's request—in this case, for the router to use DHCP to acquire its IP address.

### Brief Summary of the Web Server's Capabilities

We have looked at three distinct capabilities of the web server: transmitting HTML pages to a web browser, providing HTML files with dynamic information in them to a web browser, and using a web browser to command or control an embedded system. The examples and explanation of these features is simple. However, their use is limitless!

I have given presentations to hundreds of people on the benefits of an embedded web server. In those presentations, I always emphasize the importance of imagination in the use of this software. For about 20 K of code and a little effort, you can build systems that have sophisticated user interfaces allowing your users to understand, utilize, and control your embedded system.

What has been discussed thus far in this paper are the basic capabilities of the web server. In the section that follows, we will look at some additional capabilities that a specific implementation of a web server may or may not have.

### What Else Should You Consider?

As you continue to pursue information about and use embedded web servers, you will find that vendors of commercial packages will vary in their offerings. Some things that you should look out for are:

- Authentication (security)
- Utilities for embedding HTML files

- File compression
- File upload capabilities

HTTP 1.0 provides for basic network authentication. If you have ever tried to access a web page and received a dialog box that asks you to enter your network ID and password, you have seen the use of this capability. You should verify that the package provides the ability to add and delete users to the username/password database in the web server. In some cases, you will be required to add the code to do this.

In general, most embedded web servers will be using a very simple file system that resides in memory. Vendors that provide support for this should also provide support for building that file system on your desktop so that it can be included in ROM or Flash on your target. Furthermore, the vendor should also supply an ability to use a more full-featured file system that is capable of handling the myriad of offline storage capabilities available for embedded systems.

If a vendor supports the building of an in-memory file system (files included in ROM or Flash) as just discussed, they should also provide a file compression capability. HTML files can become large and consume a lot of space in an in-memory file system. The compression capability should be able to compress the files while building the in-memory file system and uncompress the file when requested from the web server.

HTML 3.2 provides for the uploading of files from the web browser's host machine to the web server. A vendor supplying a reasonable implementation of a web server should also provide the ability to support this feature.

### Conclusion

Web servers will continue to proliferate in embedded systems. The capabilities afforded by this technology are as broad as the imagination of embedded developers like you. This is a technology that can be harnessed to build sophisticated user interfaces to embedded systems, maintain a local repository for user documentation, allow users of the embedded system to control it, and much more.

As the ubiquity of the web continues, embedded devices connected to it will increase. We know of no better method to monitor and control an embedded system right now than a web server. I hope you have the same excitement for this technology as I do. Most of all, I hope that you will be able to employ it in your system to its maximum advantage.

This paper was written primarily to introduce you to this technology and give you some idea how it might be beneficial. Hopefully it has encouraged you to get those creative juices flowing and find a way to use this great technology.

## 8.3 Introduction to SNMP

*This article is based on "Getting Started with Enterprise MIB Design" by Richard Vlamynck (now with Nebula E.D.A.), which appeared in* NewBits *in 1996. SNMP is widely used, but it is instructive to compare the approach with the use of an embedded web server—see "Who Needs a Web Server?" earlier in this chapter. (CW)*

### Why SNMP?

SNMP (Simple Network Management Protocol) is a buzzword in the networking world. The key to understanding and using SNMP is the Enterprise MIB. An Enterprise MIB is a Management Information Base that you design and implement in an application-specific manner and customize for your real-time embedded system.

Why in the world would you want to manage a network? There are a number of situations in which it might be useful. For example, you may be building an embedded real-time system that is a network router, and you would like to have the ability to manage the router or a group of routers from a single or distributed control station. As another example, you may be building an embedded real-time system that is a networked color laser printer, and you would like to manage that resource using an off-the-shelf graphical user interface.

In considering why you may want to have SNMP embedded in your real-time system, ask yourself these questions:

- How do I currently manage networks in general?
- How do I currently manage the nodes on networks?
- Would I like to be able to allow for control of routing?
- Would I like to locate the nodes that violate protocol standards?
- Would I like to have some fancy off-the-shelf software (perhaps a GUI) for managing all the traffic that is on the network?

If you are building a networked resource, you may need to have a standard protocol for communicating information from one node to another node on the network.

### The Role of Network Manager

Managers need to manage heterogeneous networks. The managed machines might not even use a common link-level protocol. Computers and embedded systems that are controlled by SNMP might be anywhere in the network, and end-to-end transport connection is required.

### Architectural Model

The design architecture of SNMP is the client-server model. The client-server model can also be called the *manager-agent model*. It consists of:

- A program manager (client) on one machine that manages several other machines.
- A program agent (server) that runs on each machine that might be managed (host or gateway). See Figure 8.1.

Note that there is one client for many servers (hosts or gateways). Several different clients might manage the same servers (some only get the information, some change selected information on the gateway, etc.).

### A Common Misperception

A common misunderstanding in SNMP implementation is the belief that all the user needs to do is get the agent on the hardware, and SNMP is ready to go. This is incorrect.

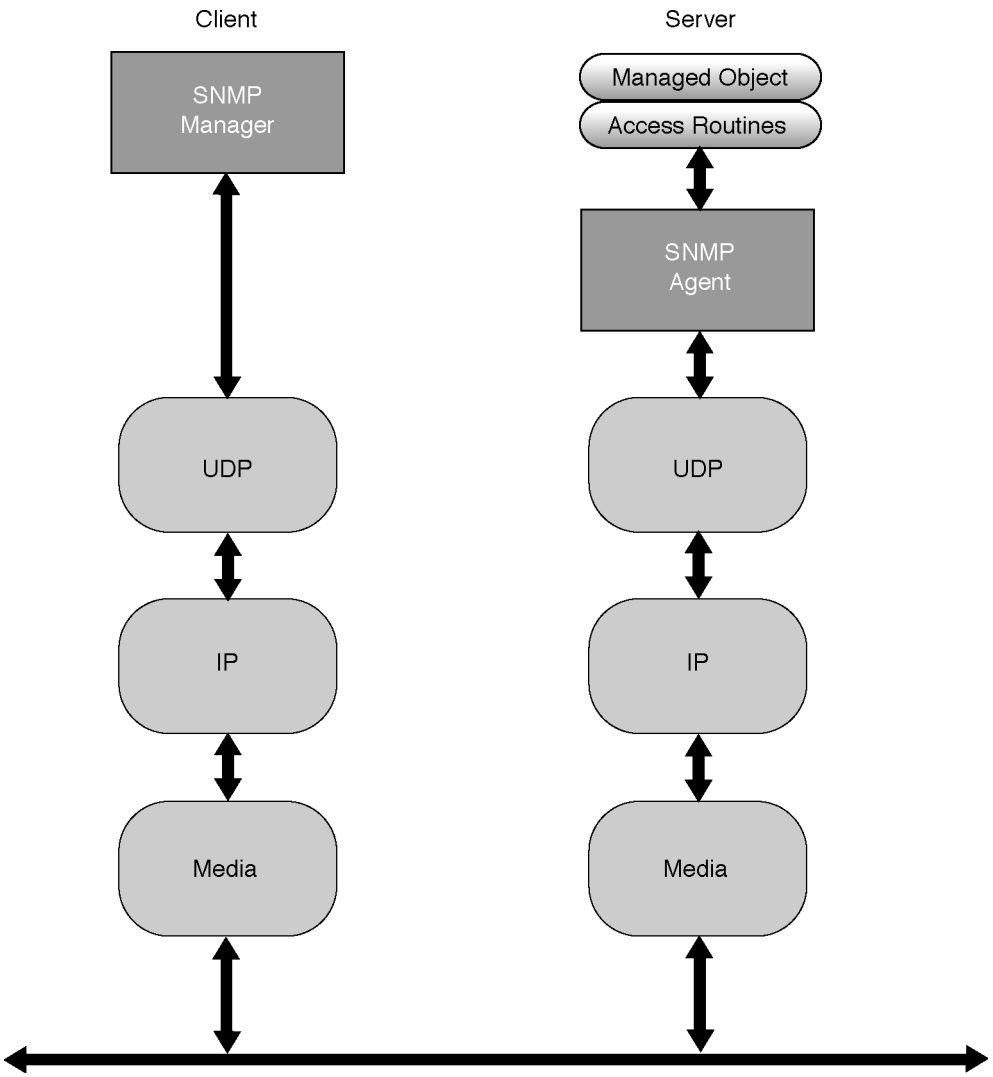



*Figure 8.1: SNMP architectural model*

The standard MIB cannot cover everything that a real-time embedded system designer might require. Therefore, the system designer needs to create a MIB for the proprietary equipment and protocols.

Your RTOS vendor provides a kernel, but you must write the embedded application yourself. That is self-evident. You can purchase SNMP, but you must write the MIB yourself. SNMP, like a kernel, is a building block you can use to add value to your embedded system.

### Application-Level Manager-Agents

There are distinct advantages to running the SNMP managers and agents at the application level:

- One set of protocols is used for the networks.
- There are no limitations in gateway hardware.
- A uniform approach is established by using the same protocols.
- The manager is using IP for communication without knowing which physical attachments are needed for the gateway or the network.

There is one drawback to this scheme. In the case of a damaged data structure, table, etc., it is impossible to reach that machine and run the server application (agent) again without intervention.

### How to Write Your MIB

Most SNMP implementations support MIB-II, which is used for monitoring generic configuration variables. It is only a starting point for building your embedded system. The embedded system designer must create and debug an applications-specific Enterprise MIB.

There is a general approach that applies to building any Enterprise MIB with SNMP:

1. Write the ASN.1 specification of your Enterprise MIB.
2. Import that specification file into the GUI (usually via the "file import" button).
3. Use the MOSY (Managed Object SYntax-compiler) compiler (or similar) to prepare the MIB for use on the target.

### Terminology

There is some minimal terminology that you must know to be conversant with other engineers about SNMP and Enterprise MIBs. The following list is a good starting point for the terms and definitions:

**TCP/IP** (Transmission Control Protocol/Internet Protocol): Set of communications protocols for networking dissimilar systems.

**SNMP** (Simple Network Management Protocol): A standard protocol that provides monitoring of IP gateways and the networks to which they attach. This standard

defines a set of variables that the gateway must keep and specifies that all operations on the gateway are a side effect of fetching or storing to the data variables.

**Gateway:** A dedicated computer that attaches two or more networks to route packets from one network to the other. Gateways route packets to other gateways until the packets can be delivered to the final destination. For example, an IP gateway routes IP datagram packets among the networks to which the gateway connects.

**Router:** A machine configured to make decisions about which of several paths network traffic will follow. Within the TCP/IP environment, an IP gateway routes datagram packets using IP destination addresses.

**Client-server model:** An interaction concept in one type of distributed system. At the application level, network nodes are termed either "client" or "server," depending on what they are doing at any particular instance. A program at one site sends a request to a program at another site and awaits a response. The client is the requesting program; the server is the program that replies. Client software is traditionally easier to build than server software.

**Manager-agent model:** An interaction concept in one type of distributed system. At the application level, network nodes are termed either "manager" or "agent." The manager sends a request PDU (Protocol Data Unit), and the agent replies with a response PDU. To translate this to the client-server model, the SNMP manager is a client, and the SNMP agent is a server.

**Transport layer:** The level of operation that connects heterogeneous networks through gateways. Internet manager software needs to control and examine IP gateways to provide end-to-end transport connections. This concept will work only if everything functions correctly. For example, if the operating system crashes, the gateway's routine table gets damaged, etc., and it would be impossible for the application-level manager program to work properly.

**CMOT** (CMIP/CMIS over TCP): ISO CMIP/CMIS network management protocols that manage gateways in a TCP/IP internet. This standard is recommended along with SNMP.

**MIB-II** (Management Information Base): The set of variables (database) that a gateway running CMOT or SNMP maintains. Managers can fetch and store data into these variables. MIB-II is an extended management database that contains variables used exclusively by SNMP and not shared with CMOT.

**MIB-II-OIM** (Management Information Base): The set of variables used exclusively by CMOT and not shared with SNMP.

**SMI** (Structure of Management Information): A specification of the rules used to define and identify MIB variables.

**ASN.1** (Abstract Syntax Notation 1): A formal language to provide notation used in documents that humans read and an encoded representation of the same information used in communication protocols. In both cases, this formal notation removes any possible ambiguities.

## Conclusions

SNMP is a very useful tool for keeping track of node activity on a network. The key to using SNMP successfully is to design and implement your application-specific Enterprise MIB.

### Basic Overview of SNMP

The key points to understand about SNMP:

1. SNMP agent is a server for managed objects called MIBs.
2. SNMP manager is a client.
3. Typically, a developer purchases a network management GUI from one vendor and SNMP agent software from another (usually the RTOS supplier).
4. The managed object, the MIB, is application-specific. It must be created, designed, built, and debugged by the developer.
5. The bulk of the work falls to the embedded systems developer to create and build the managed object, the application-specific MIB.
6. The developer does not change MIB-II; it is simply extended by writing an application-specific Enterprise MIB. This extension is straightforward because, lexically, the Enterprise MIB follows the generic MIB-II. When the users are at the end of MIB-II and do "get next," they "fall into" an Enterprise MIB.

## 8.4 IPv6—The Next Generation Internet Protocol

*Other articles in this chapter discuss strategies for dealing with the limitations of Internet addressing: DHCP and NAT. Here we take a look at the real solution: IPv6. This article is based upon an* Accelerated Technology *white paper written by Glen Johnson and Tammy Leino. (CW)*

### Limitations of the Internet Protocol

The Internet Protocol (IP), developed during the mid-1970s, is the backbone of a family of protocols that includes TCP, UDP, RIP, and virtually every other protocol used for Internet communications. The current version of the IP, version 4 (IPv4), has been in use for more than 20 years. IPv4 has proven to be amazingly adaptable over time. However, the demands placed upon the protocol at its inception pale in comparison to the demands of the millions of hosts that are now connected to the Internet, and IPv4 is finally beginning to show some chinks in its armor. IPv6 deals with many of the shortcomings of IPv4 and introduces some new features. This paper discusses three of the major problems addressed by IPv6.

#### *Depleted Address Space*

The main motivation for replacing IPv4 with something better is that the IPv4 address space will ultimately be exhausted. Estimates for the total depletion of the IPv4 address space vary quite widely from 2005 until 2018. Most estimates put the date around 2008 to 2010. Despite the disagreement on when the address space will be depleted, most agree that it will definitely happen unless something better is put in place. Compounding the problem is the uneven distribution of the IPv4 address space. The shortage of addresses will be felt first in Asia, followed by Europe, as those areas received a much smaller number of addresses—9% and 17% of the address space, respectively. Compare that figure to North America's 74% of the address space. The transition to IPv6 is well under way in Japan and the rest of Asia, and it is beginning in Europe as well.

#### *Flawed Addressing Architecture*

IPv4 addresses do not provide an efficient and scalable hierarchical address space; that is, it is impossible for a single high-level address to represent many lower level addresses or networks. To picture what a hierarchical address space looks like, think of the telephone numbering system. Just by looking at the area code, one can immediately determine what city or region to route the call to. It is not possible to look at a portion of an IPv4 address and make such a judgment. Therefore, routing becomes increasingly complicated and expensive as the size of the Internet grows.

#### *High Cost*

Another criticism of IPv4 is the high cost and maintenance requirements of networks. A significant percentage of the cost of administering an IPv4 network is incurred in the initial configuration of network hosts. IPv4's limitations also aggravate the task of renumbering network devices, which is cumbersome to network administrators.

## Introduction to IP Version 6

IP version 6 (IPv6) is a new version of the Internet Protocol, designed as a successor to IPv4. One of the myths associated with IPv6 is that the only reason to adopt IPv6 is the impending depletion of the IPv4 address space. The expanded address space of IPv6 is not the only improvement made in the protocol, however. IPv6 does solve the IPv4 address problem, but it also improves upon the current Internet Protocol in other areas including: improved addressing architecture, a stateless address autoconfiguration mechanism, a less expensive address resolution protocol, header format simplification, the ability to detect and recover from a failed forward route, and an improved method to join and leave multicast groups.

## Dual Stack Eases Transition

The Internet will consist of a combination of IPv4 and IPv6 nodes for an indefinite period. Therefore, compatibility between IPv4 and IPv6 nodes is critical for a successful transition to IPv6. Because IPv6 is not backwards compatible with IPv4, a dual stack approach is needed to enable nodes to communicate over both IPv4 and IPv6 simultaneously. This approach paves the way for transition mechanisms that will enable the Internet to move to IPv6.

Although the dual stack approach is the recommended transition mechanism for networks, IPv6 implementations can also be used in IPv6-only mode for isolated IPv6 networks. This removes the additional overhead of the IPv4 stack for those devices that do not require IPv4 tunneling.

## How IPv6 Works

### *Neighbor Discovery*

Neighbor Discovery solves a set of problems related to the interaction between nodes attached to the same link. It defines mechanisms for solving each of the following problems:

- Stateless address autoconfiguration
- Router discovery
- Prefix discovery
- Parameter discovery
- Address resolution
- Neighbor unreachability detection
- Duplicate address detection
- Redirect

Neighbor Discovery defines five different ICMPv6 packet types. The messages serve the following purpose:

- **Router solicitation:** Hosts send out messages that request routers to generate router advertisements.

- **Router advertisement:** Routers advertise their presence together with various link and Internet parameters either periodically or in response to a router solicitation message. Router advertisements contain prefixes that are used for onlink determination and/or address configuration, a suggested hop limit value, and so forth.
- **Neighbor solicitation:** Sent by a node to determine the link-layer address of a neighbor or to verify that a neighbor is still reachable via a cached link-layer address.
- **Neighbor advertisement:** A response to a neighbor solicitation message. A node may also send unsolicited neighbor advertisements to announce a link-layer address change.
- **Redirect:** Used by routers to inform hosts of a better first hop for a destination.

### *Stateless Address Autoconfiguration*

Stateless address autoconfiguration is a new feature of IPv6 beneficial to network administrators, because it requires no manual configuration of hosts, minimal (if any) configuration of routers, and no additional servers. The stateless mechanism allows a host to generate its own addresses using a combination of locally available information and information advertised by routers and verifies that each generated address is unique on the link.

Stateless address autoconfiguration should greatly decrease the costs of administering an enterprise network. Also, the task of renumbering networks will be simplified since IPv6 can assign new addresses and gracefully time out existing addresses without manual reconfiguration or DHCP.

### *Duplicate Address Detection*

To insure that all configured addresses are unique on a given link, nodes perform duplicate address detection on addresses before assigning them to an interface.

### *Router Discovery*

Router discovery is used to locate neighboring routers as well as to learn prefixes and configuration parameters related to stateless address autoconfiguration.

Router advertisements allow routers to inform hosts how to perform address autoconfiguration and contain Internet parameters such as the hop limit that hosts should use in outgoing packets and, optionally, link parameters such as the link MTU. This facilitates centralized administration of critical parameters that can be set on routers and automatically propagated to all attached hosts.

### *Prefix Discovery*

Router advertisements contain a list of prefixes used for on-link determination and/or stateless address autoconfiguration. Flags associated with the prefixes specify the intended uses of a particular prefix. Hosts use the advertised on-link prefixes to build and maintain a list that is used in deciding when a packet's destination is on-link or beyond a router.

### Address Expiration

IPv6 addresses are leased to an interface for a fixed (possibly infinite) length of time. Each address has an associated lifetime that indicates how long the address is bound to an interface. When a lifetime expires, the binding (and address) become invalid, and the address may be reassigned to another interface elsewhere in the Internet. To handle the expiration of address bindings gracefully, an address goes through two distinct phases while assigned to an interface. Initially, an address is "preferred," meaning that its use in arbitrary communication is unrestricted. Later, an address becomes "deprecated" in anticipation that its current interface binding will become invalid. While in a deprecated state, the use of an address is discouraged but not strictly forbidden.

### Address Resolution

Address resolution is the process through which a node determines the link-layer address (e.g., Ethernet MAC address) of a neighbor given only its IP address. Address resolution is redefined for IPv6 and does not use ARP (Address Resolution Protocol) packets, as is the case for IPv4.

Nodes accomplish address resolution of IPv6 neighbors by multicasting a request for the target node to return its link-layer address. The target returns its link-layer address in a unicast response. By using multicast and unicast addresses instead of the broadcast address, there are fewer needless interruptions of other nodes on the network.

### Header Format Simplification

To simplify and optimize processing of IP packets, a few changes were made to the format of the IP header for IPv6. The length of the IPv6 header is fixed as opposed to the variable length IPv4 header. This helps to simplify processing of IPv6 packets as certain assumptions in the IP processing code can be made. Also, some IPv4 header fields have been dropped or made optional. Most notable is the lack of a checksum field for the IPv6 header. This greatly improves performance in routers. When an IPv4 packet is forwarded by a router, the Time-to-Live (TTL) field must be decremented, which forces the IPv4 header checksum to be recomputed; this is a CPU-intensive operation. Since this field is not present in the IPv6 header, routers simply decrement the hop limit—TTL in IPv6—and forward the packet.

### Neighbor Unreachability Detection

Neighbor unreachability detection detects the failure of a neighbor or the failure of the forward path to the neighbor. Once failure has been detected, an alternate route can be found without interrupting the flow of data from the application's point of view.

### Multicast Listener Discovery

The purpose of multicast listener discovery is to enable each IPv6 router to discover the presence of multicast listeners (i.e., nodes wishing to receive multicast packets) on its

directly attached links, and to discover specifically which multicast addresses are of interest to those neighboring nodes. This information is then provided to whichever multicast routing protocol is being used by the router, to ensure that multicast packets are delivered to all links with interested receivers.

### Tunneling

In most deployment scenarios, the IPv6 routing infrastructure will be built up over time. While the IPv6 routing infrastructure is being deployed, the existing IPv4 routing infrastructure can remain functional and can be used to carry IPv6 traffic. Tunneling provides a way to utilize the existing IPv4 routing infrastructure to carry IPv6 traffic.

IPv6/IPv4 hosts and routers can tunnel IPv6 datagrams over regions of IPv4 routing topology by encapsulating them within IPv4 packets.

Tunneling operates as follows:

1. The entry node of the tunnel (the encapsulating node) creates an encapsulating IPv4 header and transmits the encapsulated packet.
2. The exit node of the tunnel (the decapsulating node) receives the encapsulated packet, reassembles the packet if needed, removes the IPv4 header, updates the IPv6 header, and processes the received IPv6 packet as usual.

IPv6 defines numerous techniques to accomplish tunneling. Two of the most common tunneling techniques are *configured tunneling* and *6to4 tunneling*.

### Configured Tunneling

In configured tunneling, the tunnel endpoint address is determined from configuration information in the encapsulating node. For each tunnel, the encapsulating node must store the tunnel endpoint address. When an IPv6 packet is transmitted over a tunnel, the tunnel endpoint configured for that tunnel is used as the destination address for the encapsulating IPv4 header. Configured tunneling uses IPv6 native addresses as the source and destination addresses of the IPv6 packet.

### 6to4 Tunneling

The IANA (Internet Assigned Numbers Authority) has permanently assigned the prefix `2002::/16` for the 6to4 scheme. The subscriber site is then deemed to have the address prefix `2002:V4ADDR::/48`, where `V4ADDR` is the globally unique 32-bit IPv4 address. Within the subscriber site, this prefix is used exactly like any other IPv6 prefix. The 6to4 address is used as the source address of all communications via the 6to4 tunnel.

IPv6 packets from a 6to4 site are encapsulated in IPv4 packets when they leave the site via its external IPv4 connection. `V4ADDR` must be configured on the IPv4 device.

### DNS for IPv6

To support the storage of IPv6 addresses, the following extensions have been defined:

- A new resource record type, `AAAA`, is defined to map a domain name to an IPv6 address.
- A new domain, `ip6.int`, is defined to support lookups based on address.

#### *IPv6 Extension Headers*

Unlike in IPv4, the IPv6 header is a fixed length. Any additional information that needs to be provided to the IP layer is contained in extension headers appended to the basic IPv6 header. The following extension headers are commonly supported in IPv6 implementations: fragmentation, routing, destination options, and hop-by-hop options.

#### *Ancillary Data*

Ancillary data is used to transfer IPv6 extension headers and additional control information between the application and the network stack via socket options and the "send message" and "receive message" routines. This additional data is used by the local IPv6 stack, intermediate IPv6 stacks responsible for packet routing, and the destination IPv6 stack to properly process the IPv6 packet as is required by the sending application.

Ancillary data can be used to send/receive the following control information to the stack:

- Hop-by-hop options
- Destination options
- Routing header
- The interface index of the outgoing/incoming packet
- The source address of the outgoing/incoming packet
- The next-hop address to use for the outgoing/incoming packet
- The traffic class of the outgoing/incoming packet

### RFC Support

The following RFCs are key to IPv6:

- 1886: DNS Extensions to support IP version 6
- 1981: Path MTU Discovery for IP version 6
- 2080: RIPing for IPv6
- 2373: IP Version 6 Addressing Architecture
- 2452: IP Version 6 MIB for the Transmission Control Protocol
- 2454: IP Version 6 MIB for the User Datagram Protocol
- 2460: Internet Protocol, Version 6 (IPv6) Specification
- 2461: Neighbor Discovery for IP Version 6
- 2462: IPv6 Stateless Address Autoconfiguration
- 2463: Internet Control Message Protocol (ICMPv6) for the IPv6 Specification
- 2465: MIB for IP Version 6: Textual Conventions and General Group
- 2466: Management Information Base for IP Version 6: ICMPv6 Group

- 2710: Multicast Listener Discovery (MLD) for IPv6
- 2711: IPv6 Router Alert Option
- 2893: Transition Mechanisms for IPv6 Hosts and Routers
- 3019: IPv6 MIB for the Multicast Listener Discovery Protocol
- 3056: Connection of IPv6 Domains via IPv4 Clouds
- 3493: Basic Socket Interface Extensions for IPv6
- 3542: Advanced Sockets Application Program Interface (API) for IPv6
- 3810: Multicast Listener Discovery Version 2 for IPv6

## 8.5 The Basics of DHCP

*DHCP is another important technology that has grown largely from the need to conserve scarce IP addresses on networks connected to the Internet. This article is based upon an* Accelerated Technology *white paper written by Steven Lewis. (CW)*

The Dynamic Host Configuration Protocol (DHCP) provides a means for automating the configuration of Internet hosts. DHCP can be used to automatically assign IP addresses, to deliver TCP/IP stack configuration parameters such as the subnet mask and default router, and to provide other configuration information such as the IP address of a print, time, or news server.

DHCP is built on the client-server model. Upon receiving a request from a DHCP client, a DHCP server is responsible for the allocation of an IP address and other configuration parameters the client may require. The server allocates the IP address from a pool of addresses that it is responsible for managing. The client is responsible for requesting each of the parameters that are required for proper operation on each connected network.

The primary goal of DHCP is to reduce the administrative costs of managing a network. By utilizing DHCP, the administrative costs can be limited to the configuration of the DHCP server. The DHCP server will need to be manually configured with one or more pools of IP addresses that can be used to satisfy requests. The DHCP server will also need to be manually configured with the other parameters that network hosts will require. However, there is no need to configure any of the DHCP clients. New network hosts can be plugged into the network and function with no manual configuration. Parameters such as default gateway address, DNS server addresses, and network domain name can be changed at the server for every client on a network that uses DHCP rather than requiring a manual change of each host.

### A DHCP Server

#### *Flexible Address Assignment*

Typically, when a client requests an IP address, a DHCP server will return one of many IP addresses from a pool. For most network hosts this is fine. However, it is desirable for a known IP address to be given to servers. A DHCP server may perform both dynamic and static address assignment, and both methods can be utilized simultaneously. It is also possible to configure a DHCP server with multiple noncontiguous blocks of IP addresses. This is useful when multiple blocks of addresses are used for dynamic address assignment, which are separated by one or more addresses that are reserved for static address assignment. Finally, the lease times can be configured to match the dynamics of the local network. On a very stable network, the lease times can be very long. On networks that are very dynamic, it is generally better to shorten the lease times.

#### *Support for Multiple Subnets*

Multiple subnets can be serviced with a single DHCP server. Through the use of multiple network interfaces and/or use of relay agents, the DHCP server can be configured to

handle DHCP requests from clients on multiple network subnets. Also, each subnet can be fully configured with its own set of network parameters or can use certain parameters from a global configuration, which can be used for parameters that are common across all subnets.

### Theory of Operation

There are four transactions that occur during a successful DHCP operation. All DHCP operations begin with a client broadcast of a `DISCOVER` message. The `DISCOVER` message is formatted with client characteristics, such as the client's hardware address and client ID, that will enable a DHCP server to differentiate between clients. By default, DHCP `DISCOVER` messages are sent to port 67, the well-known DHCP server port. The server will reply to `DISCOVER` requests from clients on any network serviced by the server. If the server decides that the client is one that it should service, the server will choose an IP address to offer to the client. After selecting an IP address for the client, the server will broadcast an `OFFER` message. The `OFFER` message will contain the selected IP address, lease times, and very basic network parameters. By default, `OFFER` messages are sent to port 68, the well-known DHCP client port.

Upon accepting the `OFFER`, the client will broadcast a `REQUEST` message. The `REQUEST` message will contain the offered IP address, as well as requests for each of the network parameters that it desires. The `REQUEST` message will be sent to the server port 67. If the client receives more than one `OFFER` message in response to its `DISCOVER`, only one will be chosen.

When the server receives the `REQUEST` message, it will determine whether the requested IP address is still available and whether the requested network parameters are valid. Then it will reply to the client with an `ACK` message. (Note that some clients, such as certain Microsoft DHCP clients, when restarted, will not send out a `DISCOVER` message; rather, the client will jump right to a `REQUEST`. Therefore, if the lease had expired while the client was restarting, the server may have provided this address to another client. The server will send a `NAK` to the requesting client to let it know that address is not available, and the client will send out a `DISCOVER` message.) The `ACK` message will contain the requested IP address and the data corresponding to the requested network parameters. The `ACK` will be broadcast to client port 68. Once the client receives the `ACK` message, it is free to use the IP address and network parameters for the duration of the lease. Figure 8.2 illustrates the process of acquiring an IP address via DHCP.

Although the DHCP client can use the IP address only for the duration of the lease, the client will request that the lease be extended as necessary. The expiration of the lease provides a means for IP addresses to be returned to the pool of available IP addresses in the event that the client is removed from the network. The DHCP protocol defines a mechanism for lease renewal and milestone times for clients to adhere to when attempting to renew an IP lease.

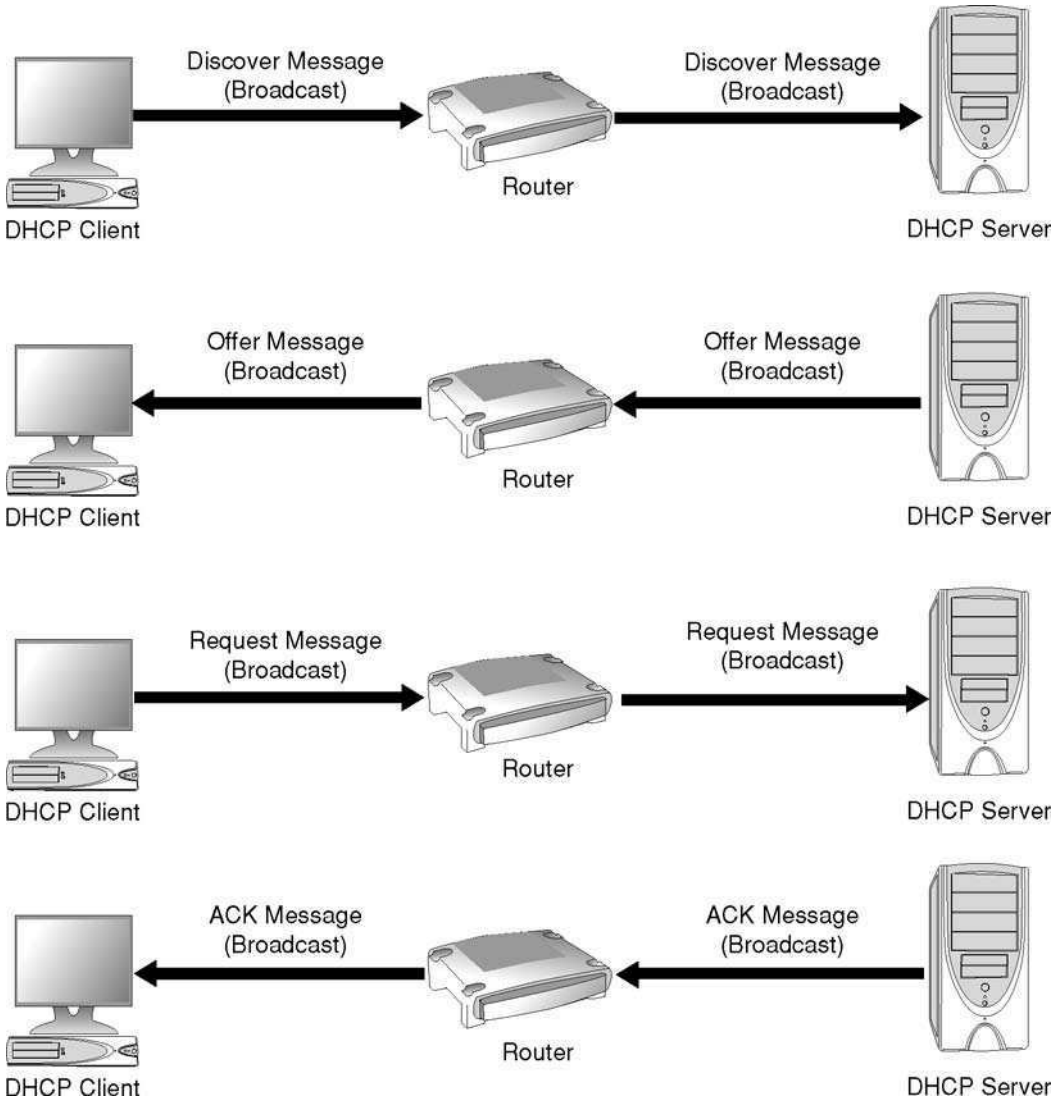



*Figure 8.2: Acquiring an IP address via DHCP*

When the client was presented with the `OFFER` and `ACK` messages, the total lease time for the IP address was provided. This lease time is determined by the administrator of the DHCP server and can vary from infinite to less than a day. In addition to the total lease time, two other lease parameters were provided to the client: a renew time and a rebind time. The renew time is the time at which the client should attempt to renew the lease with the server that currently holds the IP lease. To renew the lease, the client would unicast a `REQUEST` message to the server that provided the IP address that it is

using. The renewal time is usually 50% of the total lease time. This is done so network traffic, due to DHCP messages, will be minimized and to provide the client with a standard. The client will periodically attempt lease renewal until success or until the rebind time is reached.

If all attempts to renew the lease fail, the rebind time will eventually be reached. At this point the client will attempt to extend the lease with any server. During the rebinding time, the client will broadcast a `REQUEST` message to server port 67. The `REQUEST` message will contain the IP address that the client is currently using, in hopes that some other server will be able to extend the lease for the client's current address. The rebinding time is usually 87.5% of the total lease time. If the client is unable to extend the lease before the total lease time has expired, the client must discontinue its use of the IP address.

The simplest case for the use of a DHCP server is the servicing of a single local subnet. This is illustrated in Figure 8.3.

There are several options for the management of multiple subnets with DHCP. The obvious solution is to place a DHCP server on each subnet. This may not be possible for technical or financial reasons.

A second option is to use a DHCP relay agent, which is illustrated in Figure 8.4.

If a relay agent is implemented, it must be configured to forward all DHCP messages to the DHCP server. The relay agent does not keep track of the DHCP messages that it forwards. The relay agent only modifies the DHCP message by inserting the IP address of the interface that received the message into the gateway IP field of the DHCP message and incrementing the hops field. The server will send its replies to the interface of the relay agent that initially received the DHCP message. Since the network segment that is accessing the server through the relay agent is completely isolated from the server, a separate configuration must be used for the IP addresses that will be offered to the clients on that segment.

A final option, for those DHCP servers that have more than one network interface, is to connect the server to each subnet that must be serviced, as illustrated in Figure 8.5.

In this case, the DHCP server must be configured for multiple subnets. When the server is configured in this manner, each DHCP network parameter can differ from one network segment to the next. Also, if it is desired to have a set of network parameters to be constant across all network segments, the network manager may configure a global configuration from which all segments may draw all or some of their parameters.

### RFC Support

The requirements for a DHCP server are outlined in RFC 2131. DHCP options are outlined in RFC 2132. Not all options listed in RFC 2132 are necessarily supported by a given DHCP server.

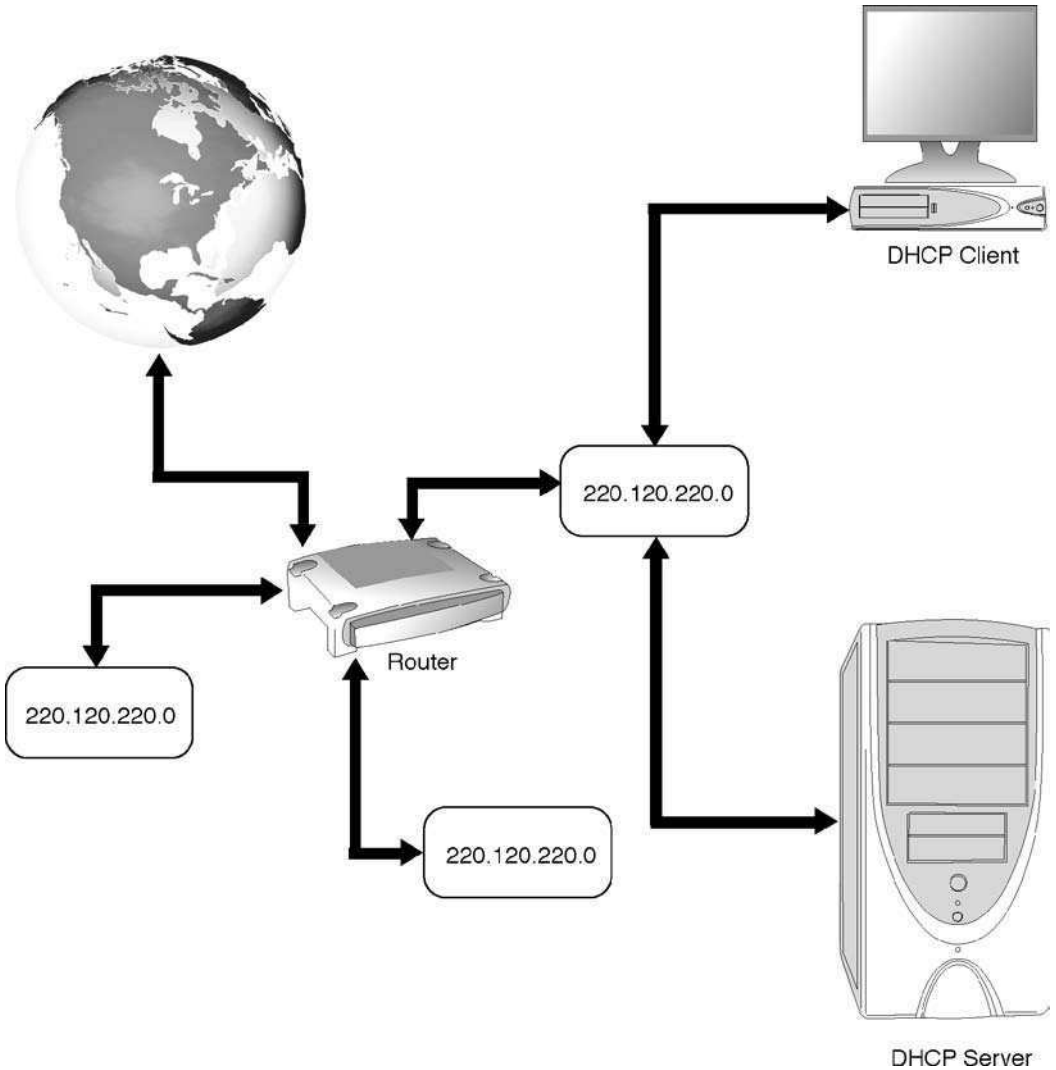



*Figure 8.3: DHCP server servicing a single local subnet*

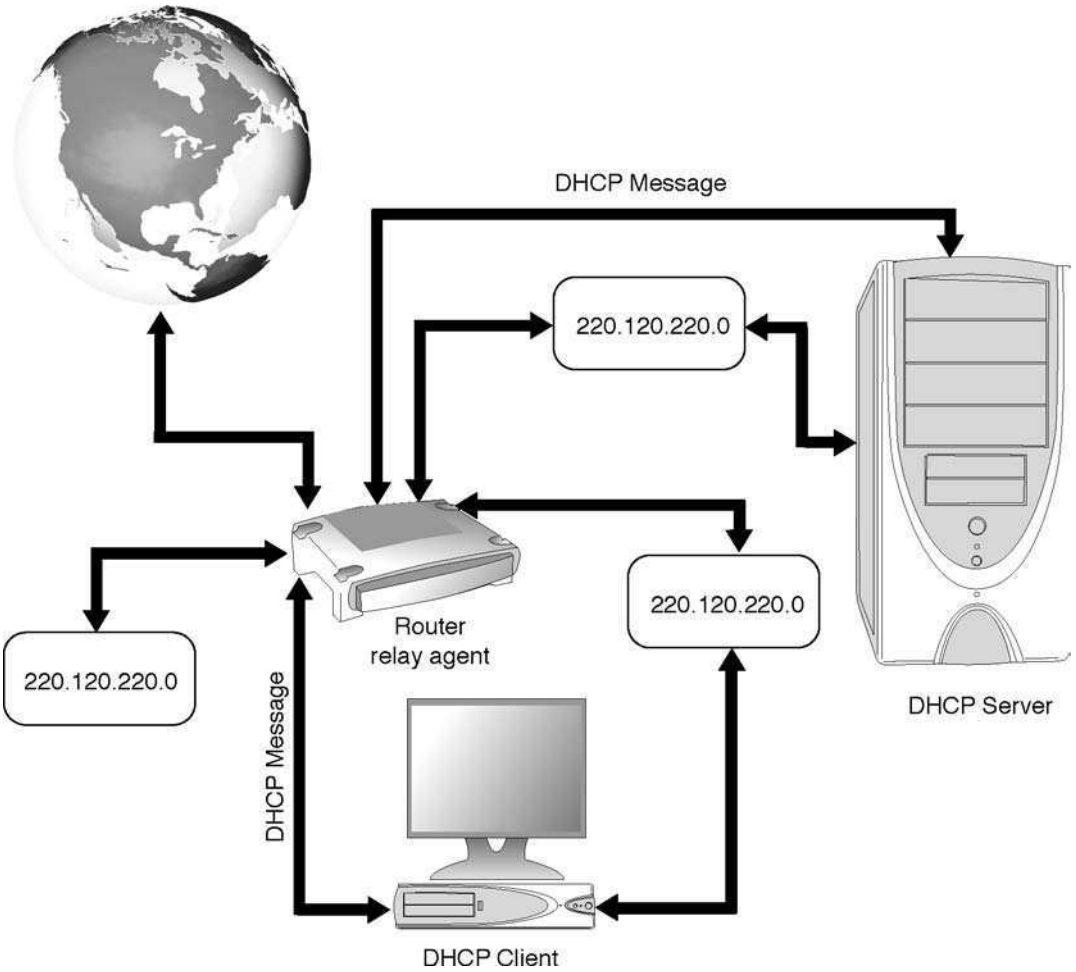



*Figure 8.4: Using a DHCP relay agent*

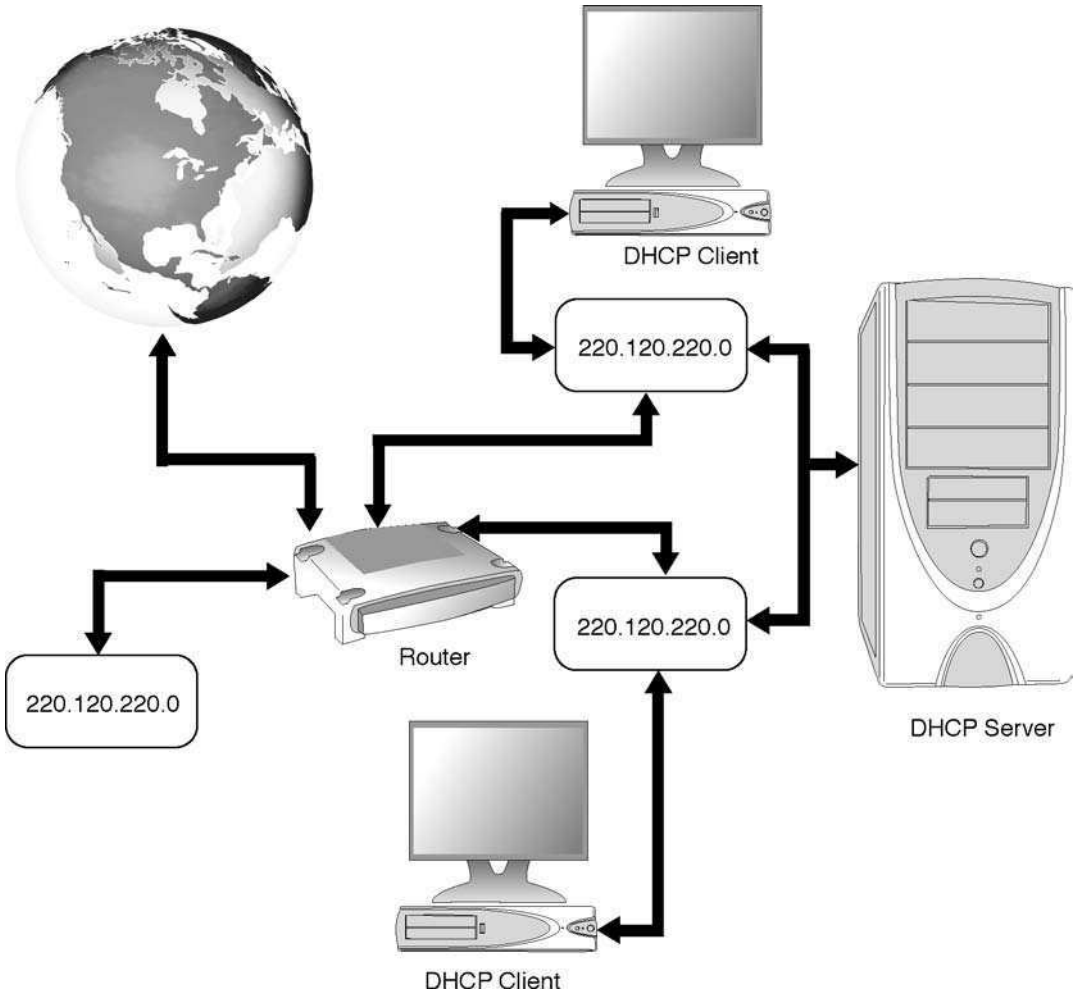



*Figure 8.5: A DHCP with multiple network interfaces*

## 8.6 NAT Explained

*Network address translation is a critical technology, as the Internet becomes ever more widely used, with more devices becoming connected, and the availability of IP addresses is becoming a serious problem. NAT is essentially a "kluge" that deals with the issue extremely well in some situations. Longer term, the next generation of IP protocol, IPv6, is likely to be the solution—this is described in "IPv6—The Next Generation Internet Protocol," earlier in this chapter. This article introduces NAT and is based upon an* Accelerated Technology *white paper written by Glen Johnson and Tammy Leino. (CW)*

The IP Network Address Translator (NAT) protocol is a router protocol that allows nodes on a private network to transparently communicate with nodes on an external network and vice versa. Nodes on a private network have not been assigned a globally unique IP address; therefore, communication with the external network would otherwise be impossible. This transparent communication is accomplished by modifying the IP and protocol-specific headers of packets flowing to and from the private network. NAT solves three common problems with growing networks: shortage of globally unique IP addresses, firewall-like protection for the private network, and flexibility of network administration.

### NAT Explained

There are a variety of flavors of NAT. Basic NAT maps an IP address on the private network to a globally unique IP address. Basic NAT performs translation on only the IP address and requires the NAT router to have a pool of globally unique IP addresses, which can be mapped. Basic NAT also limits the number of nodes on the private network that can communicate with the external network to the number of globally unique IP addresses that are available. This means that in order for five nodes on the private network to communicate with the external network at the same time, there must be five globally unique IP addresses available for translation. While these five addresses are in use, no other nodes on the private network can communicate with the external network.

NAPT (Network Address Port Translator) solves some of the problems with basic NAT and does a much better job of solving the problem of a shortage of globally unique IP addresses by allowing all nodes on a private network to communicate with the external network by sharing a single globally unique external IP address. This is advantageous for homes and businesses with limited globally unique IP addresses, because all users can access the external network simultaneously. NAPT accomplishes this by replacing, within the protocol headers, the IP address and TCP/UDP port number of the private node with the globally unique external IP address and TCP/UDP port number. In other words, NAPT performs translation on the UDP/TCP port numbers as well as on the IP address. With NAPT, the theoretical limit is up to 64,000 simultaneous sessions (address/port combinations) at a time. NAPT is also known as *IP masquerading*.

Bidirectional NAT enables connections to be initiated from hosts on the external network as well as the private network. Specific ports on the NAT router are mapped to services on a private node or server via a portmap service (see the section "The Portmap Service" later in this article). The NAT router relays all matching requests from the

external network to the specific private server. This enables servers on the private network to be accessible to nodes on the external network. For example, an FTP client on the external network could establish a connection with an FTP server on the private network. Without bidirectional NAT support, all connections have to be initiated from nodes on the private network.

Since all connections must be initiated from the private network or registered with the portmap service, NAT provides firewall-like protection for the private network. An intruder would have to first gain access to the NAT router to infiltrate the private network. Also, the size and topology of the private network are hidden behind the NAT router. Note that NAT does not necessarily preclude the need for a real firewall.

No modifications need to be made to the NAT router when a new node is added or existing nodes are removed or reconfigured. This provides for flexibility of network administration.

The theory of operation for NAT is illustrated in Figure 8.6.

The private network `192.168.16.x` is hidden from the external network behind a NAT router. The NAT router has one external interface (`201.100.67.1`) used to communicate with the external network and to protect the anonymity of the private nodes. The NAT router has one private interface (`192.168.16.1`) used to communicate with the private network.

When a private node sends a packet to the external network, the NAT router intercepts the packet and replaces all instances of the private source IP address (`192.168.16.xxx`) and TCP/UDP source port with the external IP address (`201.100.67.1`) and an assigned external TCP/UDP source port. NAT assigns the port number. No user interven-

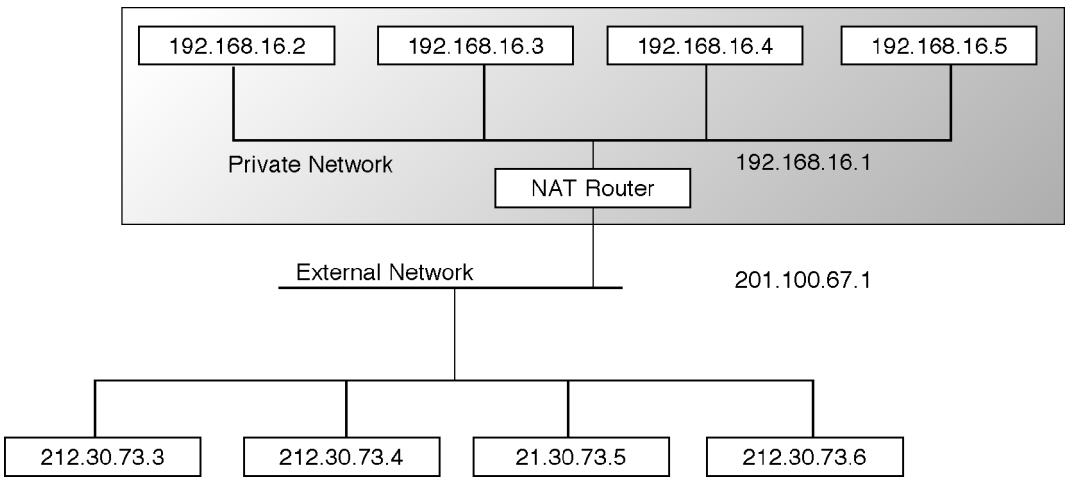



*Figure 8.6: NAT theory of operation*

tion or configuration is necessary for private nodes to initiate communication with external nodes. However, if a server on the private network needs to service clients located on the external network, then the server's port must be registered with NAT via the portmap service.

When an external node responds to a private node or initiates an acceptable connection with a private node, the NAT router intercepts the packet and replaces all instances of the external destination IP address (`201.100.67.1`) and assigned external destination TCP/UDP port with the private IP address (`192.168.16.xxx`) and destination TCP/UDP port.

#### *The Portmap Service*

As mentioned previously, NAT may be bidirectional. This means that servers can be supported on the private network. NAT achieves this via a portmap service, which is used to register services (servers) on the private network as accessible to the external network.

Multiple nodes on the private network may be registered on the same port using the same protocol. For example, multiple nodes may be registered as FTP servers. As requests for connections come in through the NAT router from the external network, NAT will forward these requests in a round-robin manner to the respective servers on the private network. This is done to distribute the work evenly across multiple servers.

Note that since the external network sees the NAT router as the one and only final destination, there is no way to specify to which of the multiple private servers the packet may have been intended. For example, if a certain file is stored on one of three private servers, and an external user FTPs to retrieve that file, it is not guaranteed that the request will be sent to the proper server. If multiple servers of the same type are to be used effectively on the private network they must be mirrored.

### RFC Support

The requirements for a NAT router are outlined in RFC 1631 and clarified in RFC 2663. Since the implementation of a NAT router is so closely related to the private network it is hiding, the RFCs are more informational overviews than stringent requirements documents.

### Protocol Support

NAT may support a wide variety of networking protocols. Note that *support* in this case means that NAT can forward data sent by these protocols from the private network to the external network. Any networking protocol can be executed on the NAT router itself. For example, if TFTP client is not listed as supported by a particular NAT implementation, this means that NAT does not support a TFTP client on the private network communicating with a TFTP server on the external network. However, a TFTP client could execute on the NAT router. This TFTP client could communicate with TFTP

servers on both the private and external network. Examples of protocols that are likely to be supported include: IP, TCP, UDP, DNS, ICMP, HTTP client-server, Telnet client-server, TFTP client-server, and FTP client-server.

### Application Level Gateways

An Application Level Gateway (ALG) is an extension to NAT, which modifies the payload of a packet aside from the IP and/or protocol headers. Note that only those applications that embed IP addresses and/or port numbers within the application payload require an ALG.

Commonly implemented ALGs are:

- ICMP: Provides functionality for ICMP error codes.
- FTP: Provides functionality for FTP PORT and PASV commands.

### The Private Network Address Assignment

The Internet Assigned Numbers Authority (IANA) has reserved the following three blocks of the IP address space for private internets:

- `10.0.0.0 – 10.255.255.255`
- `172.16.0.0 – 172.31.255.255`
- `192.168.0.0 – 192.168.255.255`

An organization that decides to use IP addresses in this address space can do so without coordination with any Internet registry. This address space information is taken from RFC 1918.

# 8.7 PPP—Point-to-Point Protocol

*Although PPP has been around for many years, it has recently become even more common with the adoption of broadband (xDSL) Internet connections. This article is based upon an* Accelerated Technology *white paper jointly written by Glen Johnson, Kevin George, Fakhir Ansari, and Uriah Pollock. (CW)*

## Introduction

PPP (Point-to-Point Protocol) provides a standard method for transporting multiprotocol datagrams over point-to-point links. In the context of a network application, PPP allows IP datagrams to be exchanged with a node at the other end of a point-to-point link. Typically, a client will initiate a PPP connection by using a modem to dial into a foreign server through the public telephone system. However, PPP is also used in environments where the physical medium is not always point-to-point. One such example is Ethernet. The PPPoE and L2TP protocols enable support for transmission of PPP packets over Ethernet.

A PPP implementation may include support for a PPP client and a PPP server, perhaps even being utilized as both at the same time. Applications only have to be aware that PPP is being used as the underlying link-layer driver when establishing and breaking the physical link—that is, during dial-up and hang up. In all other respects, the application is not aware that PPP is the low-level driver being used.

### *Abstracted Link-Layer Interface*

Because PPP is now being adapted for use over various types of physical mediums, including ATM and broadcast mediums such as Ethernet, it is necessary to recognize PPP as providing for communications over logical point-to-point links as well as physical point-to-point links. To provide for flexibility in supporting multiple link layers, the interface to the link layer is commonly abstracted. The interface to each link layer is thus a self-contained module. Serial (HDLC) and Ethernet (PPPoE, L2TP) link layers are examples. This modularization of PPP results in greater system flexibility, efficient code reuse, and hardware transparency for easier application development. It also makes it straightforward for users to plug in support for new link layers; for example, PPPoA (PPP over ATM).

### *HDLC and Modem Support*

PPP originated as a protocol for sending datagrams over serial point-to-point links. These links were usually dial-up links. Today this is still by far the primary use for PPP. As a result, PPP usually includes support for HDLC framing, as well as basic support for driving a Hayes-compatible modem.

## How PPP Works

### *Theory of Operation*

The process of establishing a PPP connection begins with establishing a physical link with a foreign peer. This can be done in one of two ways. When using serial communication

with HDLC, a client will use a modem to connect with a server over standard telephone lines. Once the physical connection has been established, the client and server can exchange packets. When using the optional PPPoE interface, the physical connection is always up. Instead of using a modem to dial into a server, a PPPoE client performs "discovery" to find the PPPoE server. To the application, the connection process looks the same for both PPPoE and HDLC. The PPPoE and HDLC modules take care of the details. However, when HDLC is used over a dial-up link, it is likely that the modem will require some initialization.

Once the circuit between the peers has been created, either via dial-up (HDLC) or other network mediums (PPPoE, L2TP), PPP will begin Link Control (LCP) negotiation with the foreign peer. During LCP negotiation, each peer tells the other what its capabilities are, so that they can make adjustments for each other's limitations. For example, one peer may be configured to send packets no larger than 700 bytes in size, while the other peer may send a full-size packet (1500 bytes).

Authentication is performed during the LCP negotiations. Authentication is optional, and some PPP servers may not authenticate clients. PPP supports a number of user-authentication protocols: Password Authentication Protocol (PAP), Challenge Handshake Authentication Protocol (CHAP), Microsoft PPP CHAP Extensions (MS-CHAP v1), and Microsoft PPP CHAP Extensions version 2 (MS-CHAP v2), one of which is selected during LCP negotiation.

Once the client is authenticated, LCP negotiation will be complete, and the next phase—Network Control (NCP)—will begin. Each higher layer network protocol will have its own NCP specification for use over PPP. PPP is likely to support two NCPs: the Internet Protocol Control Protocol (IPCP) and the Internet Protocol version 6 Control Protocol (IPV6CP). Both are used for configuring the client and server peers with IP address information. This information consists of the client and server's IP addresses, and optionally a primary and secondary DNS address, all of which are typically provided by the PPP server.

All of these stages make up the PPP negotiation, and upon completion, the link is ready for normal IP and/or IPv6 network communication. There are no restrictions on the types and number of applications that can utilize the PPP link. Obviously, available bandwidth will impose some limitation on the number of applications that can effectively utilize the link simultaneously.

Once communications are complete, PPP provides an API for closing the connection (hang up). This step is the same for HDLC, PPPoE, and L2TP, and it is taken care of by the LCP layer of PPP.

### *Application Development*

Developing a network application that uses PPP is not very different than developing one for any other network medium. There are two minor differences. First, if PPP will

be used over dial-up, the modem will have to be initialized. Second, a dial-up step is necessary to establish the link. Note that dial-up is necessary even when PPPoE or L2TP is used. However, in this case, the software virtually performs this step because a modem is not actually dialed.

The responsibilities of the application developer, when creating a PPP link, are limited to initialization of PPP and the modem, assigning minimal properties of the link, and initiating the connection with a peer. The negotiation parameters are set to the most common defaults recognized by most PPP clients and servers. In most cases, there is no need to worry about these details.

The following steps are taken to establish a PPP link within an application:

1. **Device initialization:** This step is not unique to PPP. Device initialization is similar to that of other networking applications. Parameters, such as serial port number, baud rate, and data bits, that are specific to the serial device, need to be set up and registered.
2. **Modem initialization:** If a modem is being used, it must be initialized. This is done using standard AT command strings.
3. **Establish the connection:** Next the link must be established. This step varies for PPP clients and PPP servers. A PPP client must perform dial-up, while a server will wait for a client to dial in.
4. **Disconnect:** When communications on the link are complete, hang up is performed to break the link.

### PPP Details

#### *NCP/IPCP/IPV6CP*

The PPP RFCs define multiple NCPs (Network Control Protocols). IPCP (Internet Protocol Control Protocol) and IPV6CP (Internet Protocol version 6 Control Protocol) are the NCPs commonly supported by PPP implementations. Both are responsible for configuring, enabling, and disabling the IP/IPv6 protocol for use on the point-to-point link.

#### *Authentication*

The following protocols are generally supported by PPP to authenticate PPP clients:

- None (no authentication)
- PAP (Password Authentication Protocol)
- CHAP (Challenge Handshake Authentication Protocol)
- MS-CHAP v1 (Microsoft PPP CHAP Extensions)
- MS-CHAP v2 (Microsoft PPP CHAP Extensions, Version 2)

#### *LCP*

The Link Control Protocol is responsible for establishing, configuring, and testing the data-link connection. Options include:

- Maximum Receive Unit
- Asynchronous Control Character Map
- Protocol Field Compression
- Address and Control Field Compression
- Magic Number
- Authentication Protocol

#### HDLC

HDLC provides support for sending PPP packets over serial links. This is HDLC-like framing as hardware-based HDLC is not used. Instead, a software implementation of the HDLC framer is used as specified by the PPP protocol. Typically, when using HDLC, the target system will use a serial modem to establish a physical connection with another peer over a telephone line. The services necessary to drive a modem are normally supported by PPP. These include dialing, waiting for incoming calls, hanging up the modem, and Caller Line Identification (CLI), if allowed by the underlying hardware.

Alternatively, the HDLC may also support the Microsoft Windows Direct Cable Connection (DCC) protocol. This allows the target system to establish a connection to another peer directly over a NULL modem cable. Using DCC, a phone line and modem are not required to perform network communication over PPP. This method helps to simplify the development environment by removing those unnecessary parts. The DCC protocol can also be used in applications that do not require modem dial-up functionality.

#### PPPoE

PPPoE combines the best points of Ethernet and traditional PPP, allowing a virtual point-to-point connection to be established over Ethernet. Utilizing PPPoE, it is possible to leverage Ethernet for its costs and PPP for the ability to manage user access. Also, solutions that utilize PPPoE present a familiar interface to an ISP's existing customer base, who are used to the dial-up paradigm.

Because of these advantages, virtually all of the ISPs that provide xDSL access require their customers to "dial-up" using PPPoE. As a result, residential gateways will need to include support for PPPoE.

#### Support for Both Hosts and Access Concentrators

Although residential gateways will generally only have to act as a host (client), PPPoE implementations include support for both hosts and access concentrators (servers).

#### Support for MSS Replacement

PPPoE implementations may have optional support for MSS (Maximum Segment Size) replacement. The PPPoE header adds 8 bytes of data to each Ethernet packet. As a result, the effective MTU becomes 1492 instead of then normal Ethernet MTU of 1500.

When PPPoE is used in a gateway, clients on the network will have no knowledge of this fact. When establishing TCP connections, hosts will advertise a MSS of 1500 rather than the correct value of 1492. This can result in the oversized segments getting dropped by a gateway enabled with PPPoE. This is solved by dynamically replacing the MSS in TCP packets with the correct value.

### *L2TP*

L2TP is a Virtual Private Network (VPN) protocol. It has been designed to carry PPP packets over any routed protocol like IP, IPX, and Apple Talk. It also supports any WAN technology including ATM, Frame Relay, and X.25. L2TP can be used as a tunneling protocol over public and private networks.

L2TP may be used in an L2TP Access Concentrator (LAC) and L2TP Network Server (LNS). A LAC tunnels PPP packets over a public network to the LNS, which acts as a gateway to the private network. The LAC may act as a PPP server to which remote clients connect, as shown in Figure 8.7.

Once the remote client connects to LAC, the remote client's PPP packets are then tunneled by the LAC to the LNS, in effect connecting the remote client to the private network. The LAC may also act as the remote client, as shown in Figure 8.8.

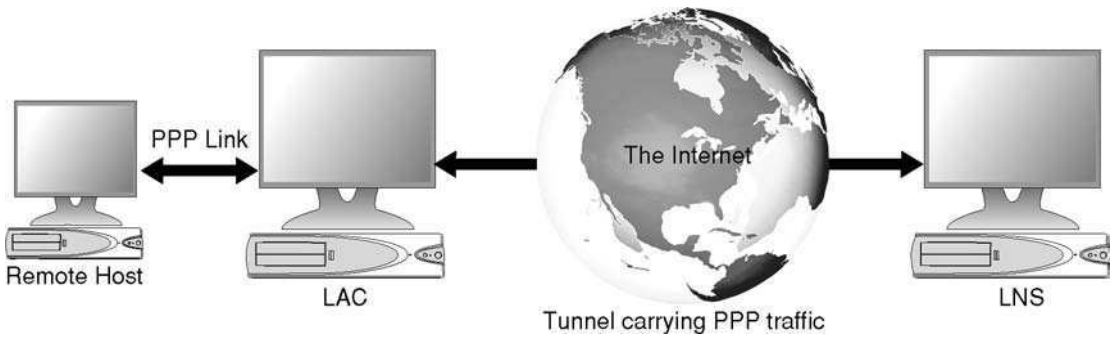



*Figure 8.7: Remote host dials into the LAC to connect to the private network*

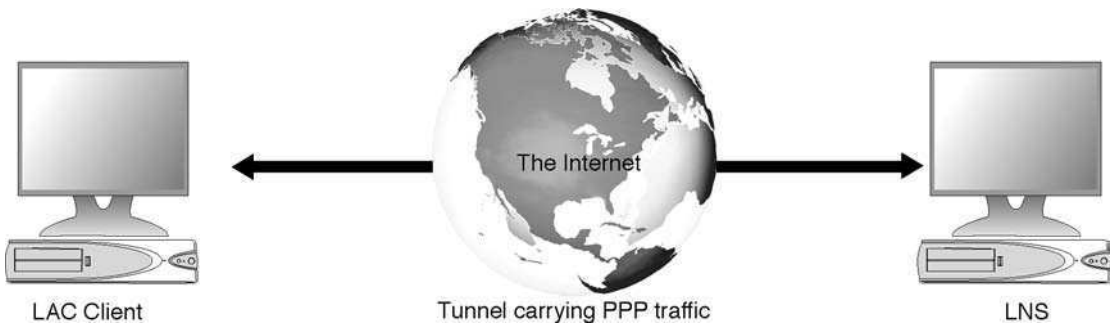



*Figure 8.8: Remote host acting as the LAC to connect to the private network*

### Multilink Protocol

Rising demands for speed, combined with the desire to utilize the existing infrastructure, makes Multilink Protocol (MP) an ideal choice. Multilink Protocol provides greater bandwidth by providing multiple independent PPP links between two network nodes. The multiple links form one virtual link, which is the only link visible to the higher-layer protocols (see Figure 8.9).

This link is used to send and receive data. When transmitting, packets are fragmented and sent over all physical PPP links that exist between the two PPP peers. On reception, these fragments are reassembled into the original packet, which is dispatched to the higher layer protocols.

### PPP MIB Support

A PPP implementation may support the following Management Information Bases (MIBs). These allow remote management of PPP interfaces and users using SNMP:

- LCP MIB: This MIB includes link statistics counters, status variables, and LCP option configuration.
- IPCP MIB: This MIB includes IP information and configuration for each PPP device.
- Security Protocols MIB: This MIB manages authentication for each link, and for each PPP user account.

## RFC Support

Following is the list of RFCs that pertain to PPP:

1332: PPP Internet Protocol Control Protocol (IPCP)

1334: PPP Authentication Protocols (PAP)

1471: MIB for Link Control Protocol of the PPP

1472: MIB for Security Protocols of the PPP

1473: MIB for IP Control Protocol of the PPP

1661: The Point-to-Point Protocol (PPP)

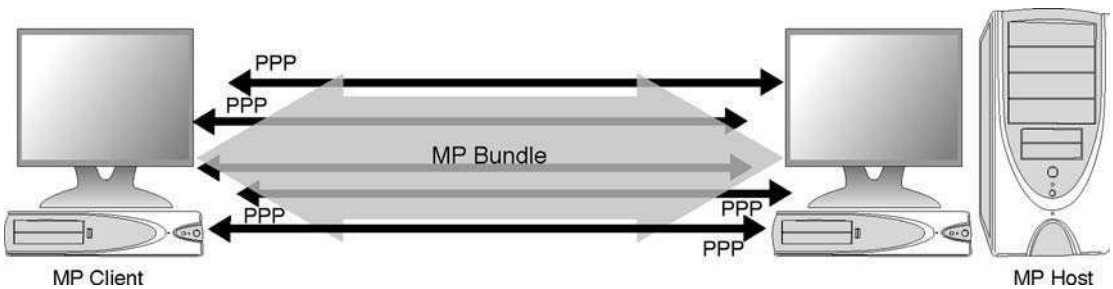



*Figure 8.9: Multiple links form one virtual link*

1662: PPP in HDLC-like Framing

1877: PPP IPCP Extensions for Name Server Addresses (DNS only)

1990: PPP Multilink Protocol (MP)

1994: PPP Challenge Handshake Authentication Protocol (CHAP)

2433: Microsoft PPP CHAP Extensions (MS-CHAP v1)

2472: IP version 6 Over PPP

2759: Microsoft PPP CHAP Extensions, Version 2 (MS-CHAP v2)

The RFC that pertains to PPPoE is 2516: Method for Transmitting PPP over Ethernet (PPPoE).

# 8.8 Introduction to SSL

*Security and confidentiality of data communications are high priorities in modern society. So it is unsurprising that embedded systems have an increasing need to support the appropriate protocols and SSL is a key component. This article, which is based upon an* Accelerated Technology *white paper by Doug Phillips, provides an introduction to the topic. (CW)*

## Introduction

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) provide a secure protocol by which two networked peers may perform encrypted communications. SSL is most commonly used for sending private data from a web browser to a web server. This private data may include credit card numbers or other personal information. Virtually everyone who has browsed the web has at some point used SSL but may not have realized it. When browsing the Internet, if the browser states that it is entering a secure site, then SSL is being used to authenticate the web server and to encrypt the data exchanged.

Netscape created SSL 1.0 in 1994. However, version 1.0 was never shipped. The first working version of SSL was 2.0. This was packaged in Netscape Navigator in late 1994. Netscape continued to work on SSL with the help of outside engineers. A year later, SSL 3.0 was released. In 1996, the Internet Engineering Task Force (IETF) took over responsibility for the protocol from Netscape. The IETF chose to rename the protocol to TLS (Transport Layer Security)—perhaps thinking a name not linked to Netscape would be more acceptable. This newly named protocol can be found in RFC 2246. Since that time, only one version of TLS was released in early 1999. Current widely used standards are SSL 2.0, SSL 3.0, and TLS 1.0. The name SSL is still more commonly used to refer to all versions of the protocol. As a result, within this article, SSL will be used to mean either SSL or TLS.

The web, or HTTP (HyperText Transfer Protocol), is the most common, but not the only, use for SSL. Other protocols such as FTP, SMTP, and POP3 (email) can be modified to utilize SSL, or SSL can be used with basic TCP or UDP communications. Once the data to be sent has been created, SSL encapsulates the data in much the same way TCP would. There is one difference: SSL modifies the data by encrypting it.

Many SSL implementations are based upon OpenSSL. OpenSSL is a free, open source SSL solution. OpenSSL was not developed with the embedded industry in mind. So care is needed for it to be used efficiently in this context.

## How SSL Works

### *Certificates*

The SSL standard specifies not only secure negotiations, but also the ability to authenticate who is on the other side of the connection. This is especially important when sending private data. Authentication is performed through the use of certificates.
A certificate is much like a driver's license. It holds the name and address of the certificate

holder, along with some encryption information and a signature. The signature, however, is not the holder's, but is placed there by a third party known as the Certificate Authority (CA). Verisign is an example of a CA.

The CA places their signature on the certificate to verify that the entity holding the certificate is who they say they are. This signature is really a hash of the certificate itself, then encrypted by the CA. This ensures that no two certificates will be alike, and therefore, a signature cannot be copied from a trusted certificate to a fake certificate.

To obtain a certificate from a CA, it is necessary to create a Certificate Signing Request (CSR). The CSR is simply a certificate without a signature. The certificate has many fields where specific information can be placed to describe the owner of the certificate. When a CSR is created, it can be given to a CA to process. Once the CA signs the certificate, it will be returned and can then be uploaded by the client-server for use as its primary certificate.

### *Initial Handshake*

An SSL session begins with an SSL client contacting an SSL server. This session will be composed of two separate parts. First is the initial handshake. The initial handshake determines if any data will be exchanged, and how that data will be encrypted. The second part will be the actual data exchange.

During the initial handshake, the version of SSL is determined. The valid versions are SSL 2.0, SSL 3.0, and SSL 3.1 (TLS). At this time, certificates may be exchanged. These certificates, as discussed previously, help to verify the client and/or the server. A certificate is required only if the opposing application specifically requests it.

At this point, the client and server must decide upon a public key encryption method. This encryption method will be used to finish the initial handshake. A public key encryption is one where the key to encrypt a message is announced publicly, while the key to decrypt the message is kept private. An example of a popular public encryption is RSA. If certificates are exchanged, the public key will appear on the certificate itself.

Once the public key encryption has been resolved, all further communications will be encrypted. For the public key encryption to be successful, however, a very large key is required. This large key makes encrypting and decrypting very demanding on the processor. To reduce processor usage, a private key encryption will be used to encrypt and decrypt the data to be sent. A private key encryption is one where the key to encrypt and decrypt are the same, and so must not be announced publicly. Since all communications at this point are encrypted with the public key encryption, it is safe to share the key that will be used for the private key encryption. This key can be much shorter than the public key, and thus is less demanding on the processor.

Once this encryption is verified, negotiations can end, and encrypted data communication can begin. Either the server or client may begin sending data. The server and client titles are only designated for the negotiation process.

The initial handshake can fail at any time. This failure can be based on invalid certificates or failure to agree upon an SSL version, public key encryption, or private key encryption.

#### *SSL API*

Many aspects of programming SSL are implementation dependent, but some general guidance may be given here. Though there are many steps in creating a secure communication, the SSL library will handle most of them. The library is first initialized with the application's preferences. These preferences include SSL versions, encryption algorithms allowed, the setting up of certificates, and the handling of buffers among other things. These preferences remain in place until the application modifies them, which is generally unnecessary after the application is running.

For the application to utilize SSL, it must first establish a connection with a peer. This connection is then passed to the SSL library. The library then begins the initial handshake. The library does not return control to the application until an SSL session has been established, or until it is determined that the two peers cannot establish a secure connection.

Once a valid SSL session has been established, the application will need to send all outgoing data through the SSL library. This outgoing data will be encrypted before it is sent to the network. All incoming data must also be received through the SSL library. All incoming data the application receives will be decrypted. Once the SSL session is terminated, and before the connection is closed, the application must notify the SSL library, which will then free up all structures pertaining to the session.

### Some SSL Details

#### *Client and Server*

An implementation of SSL may include functionality to be the client, the server, or both.

#### *Encryption*

It is common to use a separate library for encryption. This library is only accessed by the main SSL library and not by the application itself.

SSL implementations commonly include the following symmetric encryption algorithms:

| Algorithm | Key length (bits) | Mode |
|---|---|---|
| DES | 40, 56 | CBC (cipher block changing) |
| 3DES | 168 | CBC |
| RC2 | 40, 128 | CBC |
| ARC4 | 40, 128 | Stream Cipher |

and the following asymmetric encryption algorithms:

| Algorithm | Modulus (bits) |
|---|---|
| RSA | 512, 1024 |
| DH | 512, 768 |

Several encryption algorithms that are included in OpenSSL are commonly left out of SSL implementations. A complete list of such algorithms is: IDEA, RC5, and DSA. These are encumbered by patent issues. If the user were to acquire licensed software for these algorithms, they could be plugged into the SSL library.

### *Certificate Request*

The ability of a product to create a certificate is fundamental to the idea behind SSL. Once the CSR has been created, it can either be downloaded from the application for signing by a CA, or the application can create a self-signed certificate.

### *SSL and Web Servers*

The most obvious context in which an embedded system can take advantage of the secure socket layer is when it employs a web server. HTTP connections use TCP port 80, as set by the HTTP standard. SSL through HTTP is termed HTTPS and uses port 443. Embedded web servers are normally set up to watch this port for any connections. Once one is made, the web server begins the secure negotiations through the SSL API.

### *Export Restraints*

SSL implementations tend to include encryption above the level allowed by export with approval. When the end product is produced, a strict following of the laws concerning the export of encryption technology is required.

## 8.9 DHCP Debugging Tips

*From his experience in technical support, Dan Schiro did a piece on DHCP in* NewBits *in early 2004, which was the basis for this article. (CW)*

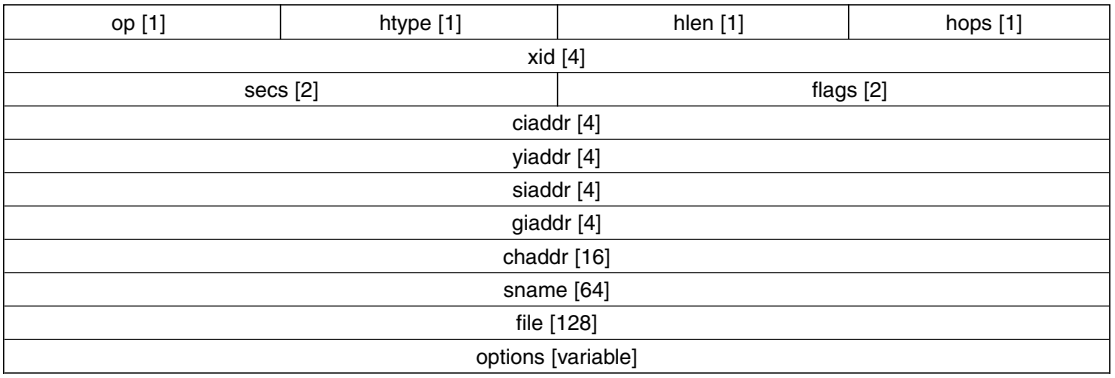
DHCP (Dynamic Host Configuration Protocol) has become a common means of network administration. DHCP is to networking what plug and play is to PCI devices. It allows network administrators to control configuration options remotely and allows end users to move their network devices around subnets without needing to know the intricacies of the network topology. Developers of network devices wishing to utilize a DHCP client need to be aware of a few key features of DHCP to avoid possible troubles in the field.

Before discussing possible issues, an understanding of how DHCP works is needed. DHCP works using a small caveat in RFC 1122: the TCP/IP software *should* accept and forward to the IP layer any IP packets delivered to the client's hardware address before the IP address is configured. This means that as long as the packet has the right MAC address, it will be passed up the stack, regardless of the destination IP, as long as an IP address has not been attached to the stack. For the purposes of the following discussion, only the procedure for obtaining an IP lease for the first time will be discussed. It will also be assumed that the DHCP server is on the same subnet as the client.

DHCP relies on UDP for data exchange. The DHCP client normally uses port 68 and the server uses port 67. These ports must be available for use. These port allocations should never be changed unless required due to special circumstances.

While negotiating an IP lease, the client and server send a packet between them with each of them filling in bits of information as it is passed back and forth. The DHCP packet structure shares the BOOTP packet structure as per RFC 2131(see Figure 8.10).

The client will initiate the communication by sending a `DHCPDISCOVER` message to the hardware broadcast address/IP broadcast address/port 67. It will fill in the `op`, `htype`, `hlen`, `xid`, `flags`, `chaddr`, and `options` fields. The `op` field will be set to 1. The `flags` field will normally be set to 0 by default—this becomes important later. The `options` field will be filled in to specify a message type of `DHCPDISCOVER`. The DHCP server, listening on port 67, will send a `DHCPOFFER` message containing the configuration parameters to port 68. The configuration parameters include `yiaddr` and `options` that typically include the message type of `DHCPOFFER`, the server IP address, the lease length, the renewal timer value, the rebind timer value, and the subnet mask. The `op` field will be set to 2 for a `BOOTREPLY`. The server's message can be sent either to the broadcast address or the address it has filled into `yiaddr`, depending on the value of the `flags` field specified by the client. The server has the MAC address of the client and also knows the IP address it is willing to assign the client. Therefore, it can send a unicast reply to the client using the offered IP address and the client's MAC address. This greatly reduces disruption to other nodes on the network by eliminating broadcast traffic. Problems arise from improper implementation of this concept. Some servers will set the client MAC address and the IP broadcast address as the destination.



| op [1] | htype [1] | hlen [1] | hops [1] |
|---|---|---|---|
| xid [4] | | | |
| secs [2] | | flags [2] | |
| ciaddr [4] | | | |
| yiaddr [4] | | | |
| siaddr [4] | | | |
| giaddr [4] | | | |
| chaddr [16] | | | |
| sname [64] | | | |
| file [128] | | | |
| options [variable] | | | |

| Field | Octets | Description |
|---|---|---|
| **op** | **1** | **Message op code / message type.** 1 = BOOTREQUEST, 2 = BOOTREPLY |
| **htype** | **1** | Hardware address type, see ARP section in "Assigned Numbers" RFC; e.g., "1" = 10 M ethernet. |
| **hlen** | **1** | Hardware address length [e.g. 6' for 10 M ethernet]. |
| **hops** | **1** | Client sets to zero, optionally used by relay agents when booting via a relay agent. |
| **xid** | **4** | Transaction ID, a random number chosen by the client, used by the client and server to associate messages and responses between a client and a server. |
| **secs** | **2** | Filled in by client, seconds elapsed since client began address acquisition or renewal process. |
| **flags** | **2** | Flags. |
| **ciaddr** | **4** | Client IP address; only filled in if client is in BOUND, RENEW or REBINDING state and can respond to ARP requests. |
| **yiaddr** | **4** | "Your" [client] IP address. |
| **siaddr** | **4** | IP address of next server to use in bootstrap; returned in DHCPOFFER, DHCPACK by server. |
| **giaddr** | **4** | Relay agent IP address, used in booting via a relay agent. |
| **chaddr** | **16** | Client hardware address. |
| **sname** | **64** | Optional server host name, null terminated string. |
| **file** | **128** | Boot filename, null terminated string; "generic" name or null in DHCPDISCOVER, fully qualified directory path name in DHCPOFFER. |
| **options** | **var** | Optional parameters field. |

*Figure 8.10: DHCP packet structure*

This can cause problems with the client depending on how the Ethernet driver is tied into the stack. The Ethernet driver will think it has a unicast packet, but the TCP/IP stack will think it has a broadcast packet. The packet could end up being rejected. To fix this, the networking stack should be made to specify a complete broadcast reply.

Once the client receives the DHCPOFFER message, it will send a DHCPREQUEST message to the server including the options it accepts. The client need not accept all options specified by the server. The server will send a DHCPACK or DHCPNACK message to the client either validating the configuration settings the client has chosen or rejecting them. If the client receives a DHCPNACK message, it must start the negotiation process over again.

Since DHCP relies on broadcast traffic, it is restricted to communication within its broadcast domain. Routers typically set the broadcast domains and would not forward broadcast traffic from one subnet to another. This limitation would require each subnet to have its own DHCP server. This is not practicable in today's large enterprise LANs. Often one DHCP server handles multiple subnets requiring the use of DHCP relay agents. Relay agents, while useful, can add confusion while debugging a negotiation issue. Relay agents inspect broadcast traffic for BOOTP/DHCP messages and forward them to a DHCP server from a preconfigured list. When forwarding the BOOTP/DHCP message, the relay agent fills in the giaddr with the IP address of the interface it received the packet on and increments the hop count. It then forwards the packet on to the server. The forwarding process may require the packet to go through multiple relay agents. If a relay agent receives a packet that already has the giaddr field filled in, it simply forwards the packet without making any changes to the giaddr field.

When the DHCP server receives a DHCPDISCOVER that has the giaddr field filled in, it sends the DHCPOFFER message unicast to the IP address specified in giaddr. The relay agent will then receive the DHCPOFFER message and broadcast it on the interface that has the IP address specified in giaddr. The relay agent may also unicast the message to the client depending on the flags value.

Knowing how relay agents work will help debug issues. If the DHCP server is on the same subnet, the giaddr field should not be filled in the DHCPOFFER/ DHCPREQUEST/ DHCPACK messages. If it is, then the lease is coming from another server. If the DHCP server is not on the same subnet, make sure that a relay agent is present, and it is properly configured. Taking simultaneous packet traces on the client subnet and the relay agent configured forwarding subnet will show if the giaddr is properly set in the packet. It will also allow for observation of the traffic coming from the DHCP server to the relay agent.

If the packet traces show the server responding to a DHCPDISCOVER message with a DHCPOFFER message but the stack does not respond, the stack may be silently discarding the packet. This can be verified by placing a breakpoint at the beginning of the code in the stack that interprets an IP address. If this is not called, then the Ethernet driver is not passing the packet to the stack.

Hopefully the preceding information will be helpful in tracking down issues.

## 8.10 IP Multicasting

*Networking is very complex, but anyone involved in embedded software needs to know something about it. Fortunately guys like Dan Schiro, who wrote the 2003* NewBits *piece upon which this article is based, deal with technical support. Those guys are used to explaining things. (CW)*

In a world that demands information distribution in a quick, seamless manner, IP multicasting has stood out as a method of providing data to a large number of hosts without generating proportionally large traffic loads on network infrastructures. IP multicasting utilizes Internet Group Management Protocol (IGMP) to send information to selected hosts only. The highest level of support for IGMPv1 is level 2. Setting up an application to use multicasting can seem daunting, but it becomes a simple task when broken down into initialization, receiving, and transmitting. Note that many of the details of the implementation of multicasting will be dependent upon the specific networking stack. So, some of the guidance provided in the following sections should be taken as an example.

### Initializing Multicasting

IP multicasting provides a means for a network application to send a single IP datagram to multiple hosts. Multicasting differs from broadcasting in that every host on a network segment receives a broadcast packet. This can lead to unnecessary interruptions for those hosts that are not interested in the broadcast. With multicasting, only those hosts that have explicitly joined an IP multicasting group will receive a multicast packet to that address/group. The class D IP address space defines the multicasting IP addresses. These addresses do not define individual interfaces, but instead define groups of interfaces. Hence class D addresses are referred to as *groups*. The class D IP addresses are those in the range `224.0.0.0` to `239.255.255.255`.

Membership in a multicast group is dynamic. An application can join and leave a group on an interface at any time. Applications join or leave a multicast group by utilizing a networking stack service call. Many IP multicasting groups are reserved for specific applications. RFC 1700 provides a current list of registered groups.

All level 2 conforming hosts are required to join the `224.0.0.1` group at initialization. The `224.0.0.1` or "all hosts group" is the group of all hosts on the local subnet. At initialization, a networking stack, which supports IP multicasting, will join this group on interfaces that support multicasting (this would normally be a build option). Multicasting can only be enabled on UDP sockets. This is because UDP is a connectionless protocol. TCP on the other hand establishes a connection with a specific host. Communication over a TCP socket is possible only with that one specific host.

IGMP (Internet Group Management Protocol) is the means by which IP hosts report their host group memberships to any immediately neighboring multicast routers. A networking stack may also have support for IGMP. IGMP is transparent to the user in that no API service calls directly invoke IGMP. Rather, each time an application joins a multicast group,

the IGMP services will be invoked by the IP layer to report the new group membership. Also, if there are any multicast routers on the local network, they will periodically send requests for updated group membership information. At this time, IGMP will report all group memberships to the router. These requests are sent by routers to the group address of `224.0.0.1`, hence the necessity of joining this group during boot up on all interfaces that support multicasting.

It is important to keep definitions straight. In this article, references to multicast packets will mean packets that contain application data and are sent to a multicast group address. IGMP packets refer to packets that contain instructions for multicast group maintenance.

### IGMP Protocol

The Internet Group Management Protocol is a layer 4 protocol and lies at the heart of multicasting. IGMP packets contain the instructions to add and/or remove an interface to or from a multicast group and perform group maintenance functions. IGMP packets do not contain any application data. The IGMP header replaces the TCP or UDP header in the packet structure.

IGMP is an asymmetric protocol and is specified here from the point of view of a host, rather than a multicast router. (IGMP may also be used, symmetrically or asymmetrically, between multicast routers. Such use is not covered here.)

Like ICMP, IGMP is an integral part of IP. It is required to be implemented by all hosts conforming to level 2 of the IP multicasting specification. IGMP messages are encapsulated in IP datagrams, with an IP protocol number of 2. All IGMP messages of concern to hosts include the following fields:

- **Version:** This memo specifies version 1 of IGMP. Version 0 is specified in RFC 988 and is now obsolete.
- **Type:** Two types of IGMP messages are of concern to hosts: 1 = Host Membership Query; 2 = Host Membership Report.
- **Unused:** Unused field, zeroed when sent, ignored when received.
- **Checksum:** The checksum is the 16-bit one's complement of the one's complement sum of the 8-octet IGMP message. For computing the checksum, the checksum field is zeroed.
- **Group address:** In a Host Membership Query message, the group address field is zeroed when sent, ignored when received. In a Host Membership Report message, the group address field holds the IP host group address of the group being reported.

### Implementing Multicasting

Multicasting requires the interaction of the Ethernet driver and the networking stack. How to receive multicast and IGMP packets will be discussed first.

### Receiving

To receive a multicast packet or IGMP packet, the Ethernet driver must be initialized to utilize multicast. Driver initialization varies with the hardware, but one common component is the initialization of filters for handling multicast and IGMP packets. These packets will have a multicast group MAC rather than the individual MAC. The driver needs to initialize the hardware to tell it to filter for packets that have either the individual address or the group addresses that the device belongs to.

Since the driver works at the second layer, it only has access to the Ethernet header. That header contains a type field along with a source and destination MAC field. This means that the driver needs to be able to determine by the destination MAC address if the packet is for itself or another device. The Ethernet driver needs to parse out the individual/group bit in the destination address field. If the destination MAC address is marked as individual, then it is compared against the Ethernet device's MAC address and the packet is accepted if they match or rejected if they do not.

Before discussing destination MAC addresses with the group bit set, it is necessary to discuss how the Ethernet driver maintains the list of multicast groups to which it belongs. Whenever a multicast group is joined, the Ethernet device will generate a multicast MAC for a particular group using a defined algorithm to assure that other devices on the network utilizing multicast will be using the same MAC for the same group. The Ethernet device will then store this multicast MAC for filtering incoming packets. This may be done in a number of ways, such as a hash table.

If the packet's destination MAC address is set as group, then the destination address is filtered against the stored multicast MAC addresses. If a match is found, then the packet is passed into the networking stack, which will determine if the packet is an IP or ARP packet. Since it is either a multicast or IGMP packet, it will be determined to be an IP packet. Then it will determine if the packet is TCP, UDP, IGMP, or ICMP. It is at this point that the processing of multicast and IGMP packets differ. If the packet is a multicast packet—meaning it contains application data—then the packet will be determined to be UDP and will be processed accordingly. If the packet is an IGMP packet—meaning it contains group maintenance information—it will be determined if the IGMP packet is a host membership report or host membership query and be processed accordingly. If the IGMP packet is neither of those two options, it will be discarded.

For an application to receive multicast messages, it must first join a particular multicast group. This is done through a network stack API call. This is likely to include options to allow for adding or dropping multicast group membership, changing the Time To Live on multicast packets, and invoking the necessary functions to update the Ethernet driver's multicast MAC filter. At this point, the stack is able to receive multicast packets for the groups that have been joined.

### *Transmitting*

The transmission side of multicasting is also dependent on the type of packet. If the packet is IGMP, the transmission is transparent to the developer.

To transmit a multicast packet (application data), the developer will use the standard UDP transmission processes with a few modifications. If the device will be a client, a server address structure for the multicast group will need to be created prior to the "send" call. However, instead of placing the server's IP address in the address structure, the multicast group address will be placed in it instead. This will result in the packet being received by all members of the multicast group. The "send" call will be processed as normal. The IP address in the server address structure will be used to resolve the multicast destination MAC address for the header creation—hence the need for each multicast group to use the same multicast MAC. However, the individual host information will still be placed in the source fields of all headers. Following this procedure, it is not necessary for an application that will only transmit to a multicast group to join the group. The only time it is required to join a multicast group is if the application needs to receive messages from the group.

## Bringing It All Together

If the device is to be a server, and depending on the application, then the developer may need to create a client address, in addition to the local address structure, prior to the "receive" call for a particular multicast group. If the application is to respond to the entire multicast group after receiving a message sent to the group, then the local and client address structures must be created prior to the call. In this case, the address structure fields filled in by the "receive" call cannot be used in the following "send" call because they will contain the information for the specific host that sent the multicast packet; they will not have the multicast group information in them. The client address structure will need to be recreated prior to the "send" call so that it will contain all the correct multicast information. If the application only needs to respond to the individual host that sent the multicast message, the standard methods of using "receive" and "send" will suffice.

# 9

# *Embedded Systems and Programmable Logic*

*This last short chapter may look like an afterthought. In a way it is. Given that many of the articles in this book have a historical perspective, I wanted to end with a topic that was clearly forward-looking. Embedded software is a very "fashion conscious" business: new ideas come and go. Some new things take root and become part of the culture (like C++); others fade away. At the time of writing, many exciting technologies are getting a lot of attention—UML and Eclipse, for example, both of which get some coverage in this book. But I figured programmable logic might be fitting. The basic technology has been around for quite some time, but its significance to embedded systems is rather newer. I have a feeling it's here to stay.*

*I made a request to the major FPGA vendors to provide some material for this chapter, and I was delighted to get an enthusiastic response from both Xilinx and Altera. I am pleased to include articles from both companies here. (CW)*

## 9.1 FPGAs and Processor Cores: The Future of Embedded Systems?

*In recent years, FPGAs have become a "big thing" in the hardware development world. For embedded software developers, they have just come on the scene, but they are certainly here to stay and will change the way a lot of systems are designed. In this article, based upon one that I did for* NewBits *in early 2004, I explain what FPGAs are and why they have become of interest. (CW)*

In this article, I will give you an introduction to Field Programmable Logic Arrays (FPGAs) and processor cores, with a particular focus on software development.

### Programmable Logic

I'll start off with the basics. Just what is an FPGA?

To consider this question, we need to look at the various styles of electronic system design and how they have evolved. Board-based design is a simple enough concept. The devices are supported and connected together by a printed circuit board—a PCB. If one or more of these devices happens to be a microprocessor or microcontroller, it is an embedded system. For many applications, this is an efficient and economic way to go about system design. We're all familiar with developing software in such an environment. No surprises here.

The next step is to look at putting lots of system functionality on a single piece of silicon. We refer to one or more microprocessor cores included on a single piece of silicon as a system on a chip or SoC—see Figure 9.1. For high-volume applications, such custom silicon can be very economic. However, SoCs exhibit considerable design challenges. For the hardware designer, the long production lead times mean that heavy use of simulation technology is required to get the design right the first time. For the software developer, constrained memory capacity and limited debugging access to the processor present unique challenges.

Of course, SoCs and boards go hand in hand. The devices are mounted on the board, with appropriate "glue logic" linking them together. Making this logic more efficient to implement was the next challenge. And this led to the development of programmable logic devices (PLDs). The concept of a PLD is straightforward. There is a matrix—an array—of logic elements. They may be connected together in a multitude of different

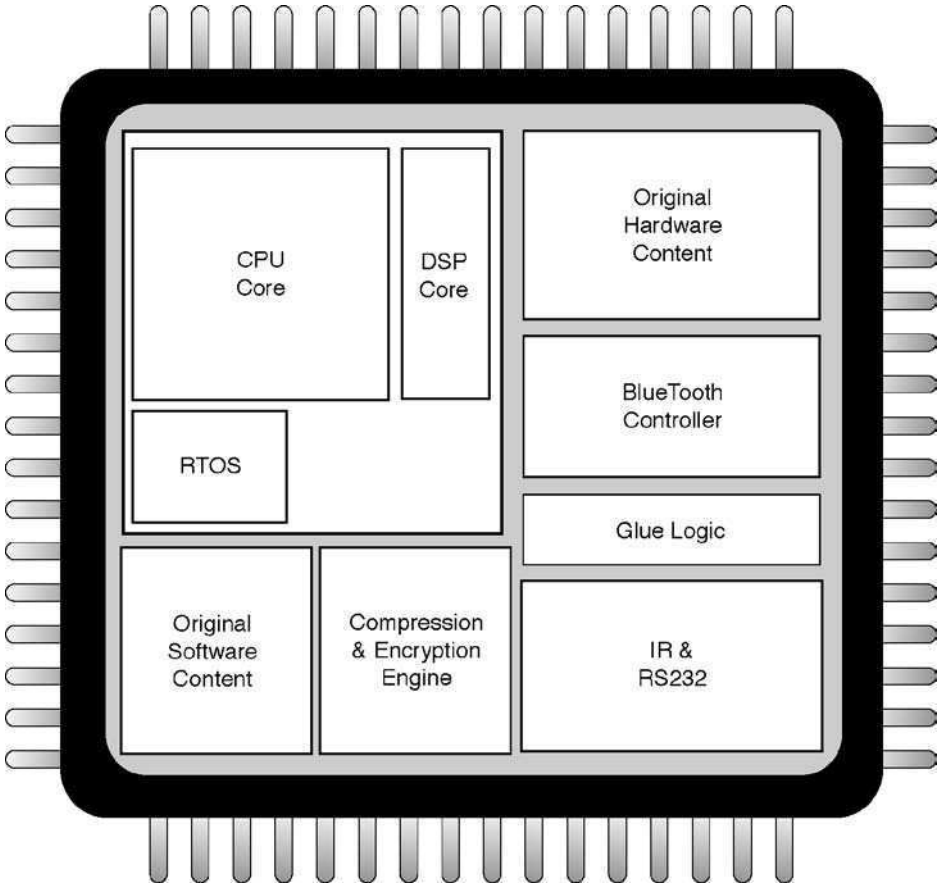



*Figure 9.1: A system on chip (SoC)*

ways to configure the logic for the job in hand. As these became more popular and silicon technology got better and better, they grew—getting bigger and bigger, until the latest devices, where there are literally millions of logic elements and interconnects. These flexible system components have begun to have the potential to become systems in their own right! This is the way that FPGAs are viewed today.

### FPGAs

Early PLDs were "use once." They had fuse or antifuse links that were "burned" in a similar way to a PROM. Modern devices often keep the programming data in SRAM, which is loaded from external flash memory on powerup. With their increasing capacity and, hence, the greater design complexity, various programming methodologies have been developed over the years. Today, almost all FPGAs are designed using hardware design languages (HDLs), which have uncanny similarities to software programming languages. In Europe, the most commonly used HDL is VHDL, which has a syntax

based upon Ada; in the United States, it's Verilog, which has syntax derived from C. There is an increasing move toward a new generation of languages that have an even closer resemblance to C or C++.

### Specialized Logic and Hard Cores

In recent years, a couple of interesting trends have become established. First, the concept of an FPGA has gone beyond a simple array of uniform logic elements. Many vendors now offer parts with various specialized logic components that may be incorporated into a design, yielding a more efficient result. Examples include memory storage, signal processing blocks, and a selection of peripheral device components. From the embedded developer's point of view, the most interesting example of such "specialized logic" is the use of microprocessor cores. Examples include PowerPC processors embedded in Xilinx parts and ARM processors utilized by Altera. These are commonly termed "hard cores," because they are fixed pieces of logic built into the chip.

### Intellectual Property and Soft Cores

The other key trend is the relentless increase in capacity offered by the newest FPGAs. This presents an interesting challenge: the capacity is so great that being able to develop enough logic to make good use of this space is very tough. The simple solution is to license and reuse design elements. Such "intellectual property" (IP) is available from a variety of sources, and many types of devices are available, including USB, PCI, and Ethernet interfaces, for example. Again, from the embedded developer's point of view, the availability of processor cores is of interest. The initial offerings were "clones" of well-known processors, like 8051, Z80, and the x86 family. More recently, processors designed specifically for instantiation in FPGAs have appeared. The best-known examples are Nios from Altera and MicroBlaze from Xilinx. These are both configurable cores and may be further optimized for specific applications. In contrast to the fixed logic of hard cores, these devices, which are realized in the fabric of the FPGA, are termed "soft cores."

### FPGA Systems

So, FPGA systems may comprise a processor—a soft core or a hard core—along with memory and lots of custom logic. But they can go further than that. Vendors are already providing devices with multiple hard cores on a single chip. For example, Xilinx offers a part with four embedded PowerPCs—an incredible amount of processing power in a single device. Of course, the soft cores may be instantiated multiple times because their consumption of FPGA fabric may be quite modest. This provides a lot of flexibility in system design, but what gets really interesting is when one or two hard cores are doing the "serious" work, with a number of soft cores performing ancillary services (see Figure 9.2). Again, there is outstanding flexibility in system design, enabling entirely new applications to be considered.

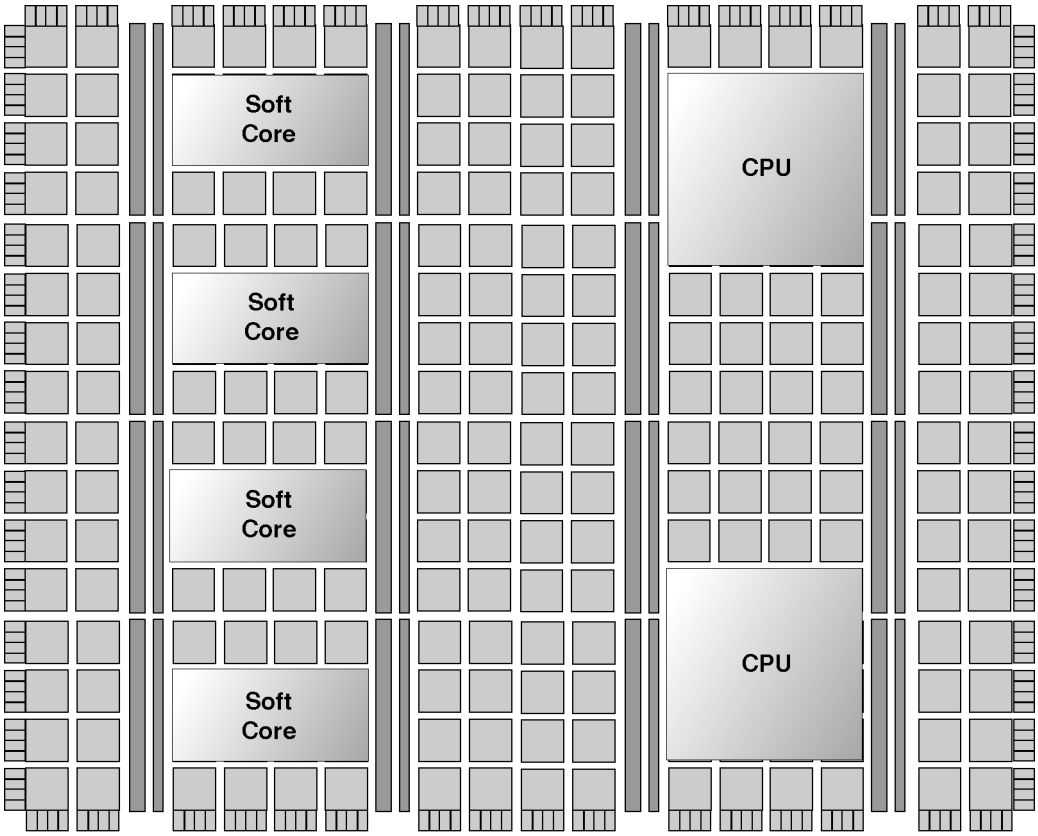



*Figure 9.2: An FPGA system*

### Software for FPGA Cores

With such flexible system design, the big challenge comes with the software. For the most part, the software requirements for the FPGA cores—hard cores and soft cores—are little different from regular microprocessor devices. This gets particularly interesting with a heterogeneous mixture of hard and soft cores.

First off, you need software development tools—compilers and debuggers. The connection to the target is likely to be JTAG, so there may also be a need for some JTAG IP. Second, all but the most trivial applications will need an RTOS. This system must be small and fast, of course, but other desirable characteristics are less obvious. If the same RTOS is available for many processor architectures, it may be employed right across a more complex FPGA-based design. This has a big payback in the deployment of expertise. From a commercial perspective, as many soft cores are royalty free, a similar business model for the RTOS may well be desirable.

## 9.2 FPGA-Based Design Delivers Customized Embedded Solutions

*Bob Garrett of Altera responded to my request for articles about FPGA-optimized processors with a piece that addressed the possibilities of the whole technology, which was the basis for this article. Of course, the capabilities of Altera's Nios II MIPS-based processor were never far from his mind. (CW)*

One of the biggest challenges faced by embedded developers is selecting a processor for their next design. With literally hundreds of off-the-shelf embedded processors available, designers must choose carefully to obtain the right mix of features, performance, and price. Experience indicates that you can usually achieve any two of these goals. Invariably, designers either purchase more processor than they need (to get the right mix of features and performance), or they get less than they had hoped for and must compensate some other way.

Custom systems on chips have been a good alternative for those who can afford the time and cost required for a gate array or cell-based ASIC design. A custom design offers the benefits of increased system integration (i.e., lower cost) and higher performance, but comes with high cost, long design cycles, and significant risk. As device geometries continue to shrink, and ASIC design costs continue to grow, fewer and fewer applications can justify the expense of a full-custom design.

With the advent of low-cost FPGA families, in which you can implement one or more embedded processors, custom embedded solutions are now viable for cost-sensitive applications and are well within the reach of the average embedded developer. Examining the possibilities that this relatively new embedded methodology offers lets you understand the benefits and tradeoffs.

### Embedded Processors in FPGAs

Building a custom device containing an exact set of peripherals, memory interfaces, and processing functions is not hard to imagine: ASIC designers have been doing it for years. Early efforts to create a custom FPGA-based embedded processor device were sometimes successful but were usually a tribute to the designer's technical prowess and weren't economically viable when compared to the discrete alternatives. It wasn't until the late 1990s that FPGAs had enough on-chip memory, programmable logic, and raw performance to create a commercially viable embedded processor. Today, embedded IP functions designed specifically for FPGAs—including CPUs, signal processing engines, peripherals, and standard communications interfaces—are readily available and offer both cost and performance benefits over traditional discrete embedded devices.

FPGA processors can be "hard" or "soft" core. Hard-core processors are implemented within the fixed silicon design on the FPGA, similar to discrete processors. On FPGAs, however, the CPU is surrounded by programmable logic that can be configured to perform other functions (e.g., peripherals, memory interfaces). Hard-core processors typically offer higher CPU performance than soft-core processors, depending on factors

such as processor architecture, clock rate, and process technology. As the name implies, hard-core processor performance and feature sets are fixed and typically offered only as a variation of a particular FPGA. The number and type of hard-core processors within an FPGA is also fixed as a function of that particular FPGA.

Soft-core processors, conversely, are implemented in programmable logic, use on-chip resources such as multipliers and memory, and can be instantiated in almost any FPGA family. The performance and cost of a soft-core processor depend mainly on the FPGA in which it is instantiated, but performance and cost are typically lower than in hard-core implementations. The number of soft-core processors that can be instantiated in a single device is limited only by the device's resources (i.e., its logic and memory). High-density FPGAs, for example, can contain hundreds of soft-core processors. Likewise, the different types of soft-core processor can be implemented: 16- or 32-bit, performance optimized, logic-area optimized, and so on. Designers can choose to migrate their soft-core processor designs to hard-core implementations when moving to high-volume structured ASICs, gate arrays, or cell-based designs.

### Connecting Systems

Processors must communicate with memory, peripherals, coprocessors, and external devices. Connecting several blocks of complex IP functions can pose challenges that, at first, may appear modest but can impact system development time and performance. The list of logic functions required to "glue" the system together includes address decoders, data multiplexers, interrupt controllers, wait-state generators, arbitration logic, bus width sizing, signal timing and level matching, and clock domain crossing logic. These IP functions are well within the capabilities of a good hardware designer, but hand-coding Verilog or VHDL for these (arguably) low-value IP functions seems a waste of good engineering talent. Routine tasks such as these are perfect candidates for computer automation. Modern FPGA system-generation tools handle these chores seamlessly, letting designers focus on more important aspects of their project.

In traditional embedded systems, processors share a system bus with other "master" components that periodically request sole access to the shared bus. This interconnect topology creates a potential bottleneck when two masters need to access the bus at the same time. FPGAs are rich in routing resources, allowing masters to have dedicated multiplexed data paths to slave components.

Changing from master-side to slave-side arbitration allows simultaneous transactions to occur unless two masters try to access the same slave device at the same time (see Figure 9.3). Using this switch-style topology allows all masters to communicate to their dedicated slave devices simultaneously, dramatically boosting overall system throughput.

### Establishing Processor Performance

Establishing processor performance requirements for embedded systems can be challenging, particularly when the application software is still in flux. Industry-standard

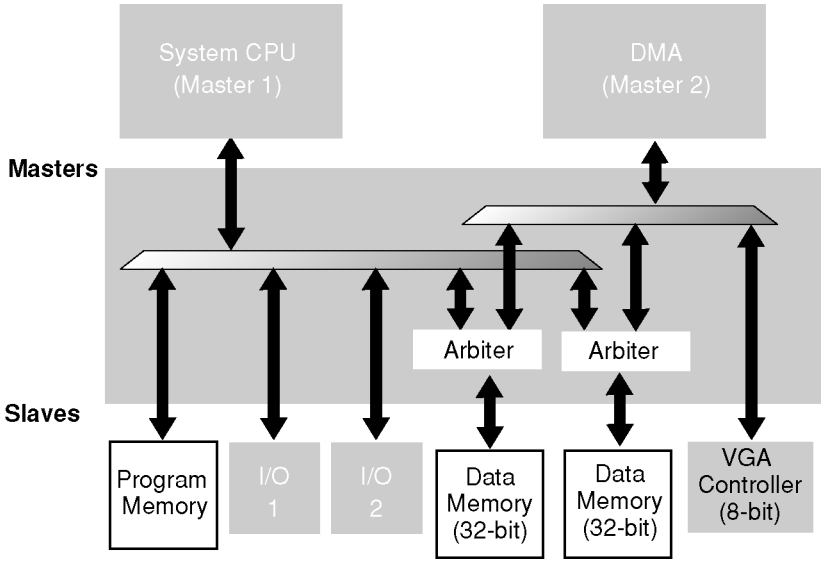



*Figure 9.3: Slave-side arbitration*

benchmarks provide some guidance, but nothing is certain until the software is complete. This tends to make designers cautious about under-calling their performance needs and may result in selecting a higher performance (and higher price) device than necessary. If a designer could accurately predict the performance required, processor selection would be much simpler. Such estimates would consider performance required by time-critical tasks as well as the load created by one or more low-priority tasks.

FPGA-based embedded systems can provide scalable performance, allowing last-minute changes to boost system performance based on customer demands. Compute-intensive algorithms, converted to logic in an FPGA, can run orders of magnitudes faster than the same algorithm run in software by a microprocessor or digital signal processor. More importantly, hardware resources can be applied to performance-hungry algorithms where they are needed most, potentially reducing the need for a high-performance CPU, reducing clock frequency, reducing power consumption, and simplifying the board design.

Performance enhancements available to the developer in FPGA-based designs fall into several classes, which are discussed in the following sections.

### *Extending the Instruction Set*

The ability to extend a processor's instruction set to include application-specific algorithms implemented in hardware is offered by several processor IP vendors and is also available in FPGA implementations. Using the processor's normal load/store operations, data can be fed to a custom logic block that essentially becomes a part of the processor's

arithmetic logic unit (ALU—see Figure 9.4). In some cases, custom instructions can support multicycle operation and access other system resources such as FIFOs and memory buffers. Typical applications of custom instructions include bit manipulation, complex numeric, and logical operations.

While reliant on processor load and store operations, custom instructions can provide significant performance benefits over running the same algorithm using normal ALU resources. For example, a cyclic redundancy check (CRC) of a 64 K buffer executed 27 times faster when implemented as a custom instruction than when run in software (Lara Simsic, "Accelerating Algorithms in Hardware," *Embedded Systems Programming*, February 2004). Performance results will vary from application to application, but in general are much faster than software alone.

How the developer makes use of a custom instruction varies by processor IP vendor. On one end of the spectrum, a new compiler must be generated when adding custom instructions. This custom compiler then infers custom instruction calls based on some application criteria. A more simplistic approach relies on users to call the instruction explicitly in their C code just as they would a subroutine. This approach seems more natural since the software designer will most likely know best when to use a custom instruction.

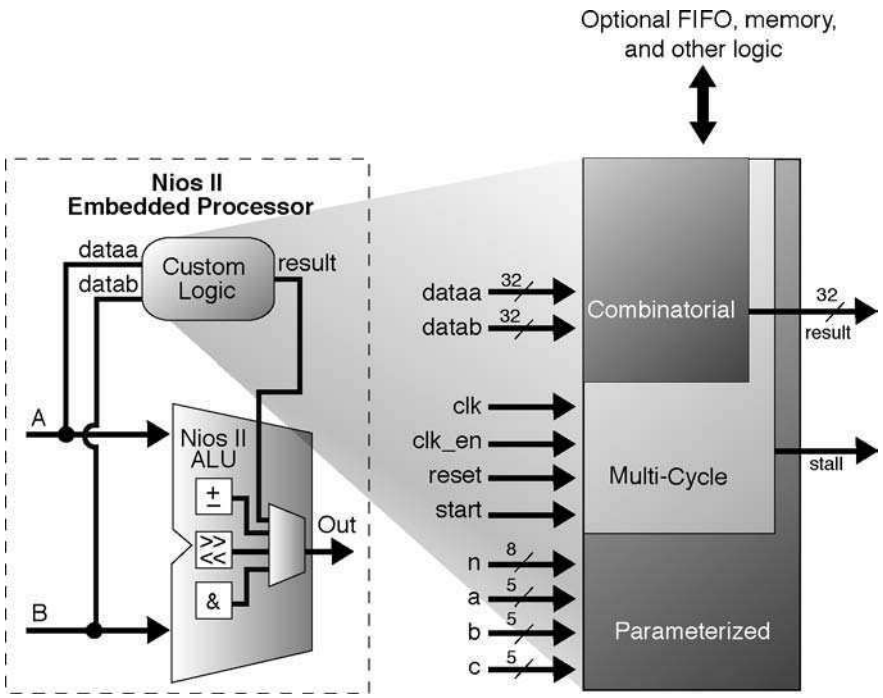



*Figure 9.4: Extending the Nios II processor*

### Adding On-Chip Coprocessors

Another approach to boosting system performance is to add hardware accelerators (i.e., coprocessors). Unlike custom instructions, hardware accelerators operate as independent logic modules that take direction from the embedded CPU and can process entire buffers of data without CPU intervention. The diagram in Figure 9.5 shows the processing module with two half-DMA channels (one to read input data, the other to store the results), and a control interface for the CPU to set-up, start, stop, and poll the unit during operation. This architecture is much better suited to tasks that involve large blocks of data where the process of the CPU loading data and storing the results would become the performance bottleneck.

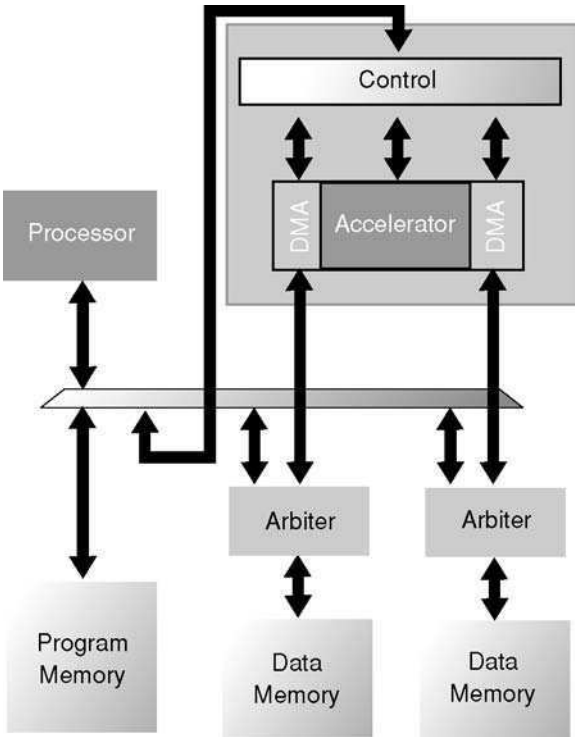



*Figure 9.5: On-chip coprocessor*

Hardware accelerators can achieve several orders of magnitude performance increase over tasks run in software due to their autonomous nature and the fact that the acceleration function is designed in hardware. Using the previous CRC example, the same operation performed by a hardware accelerator processes data 530 times faster than a software-only approach.

### More Heads Are Better Than One

Multiple soft processors can be added, within the FPGA logic resource constraints, as a divide-and-conquer strategy to increase system performance. While the term "multiprocessor" may conjure memories of academic papers on "parallel processing," commercial applications of multiple CPUs in a single device are much more straightforward. Essentially, designers partition the problem the same way they might if they were building a multiprocessor system on a printed circuit board, each assigned to specific task.

### Multiple Independent Processors

The configuration shown in Figure 9.6 is useful when software tasks can be grouped where there are little or no dependencies across groups. Each processor is treated as an individual CPU, assigned to its own set of processing tasks. For example, one processor might perform general system housekeeping such as monitoring cabinet fans, man-machine interface, or the system console, while the others handle communications, signal processing, statistics gathering, or other system tasks.

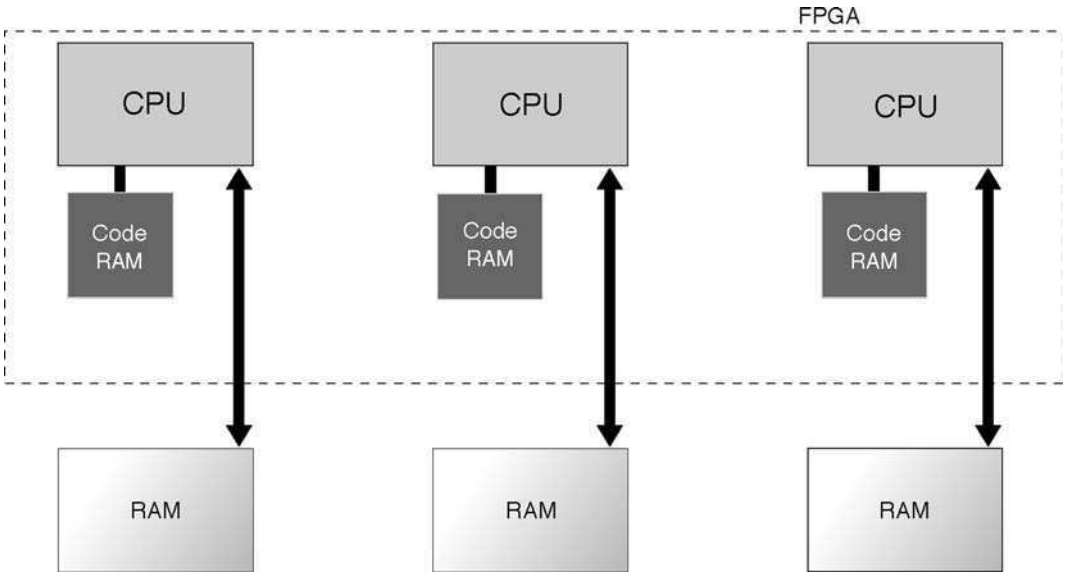



*Figure 9.6: Multiple independent processors*

### *Multichannel Applications*

Multichannel applications, as shown in Figure 9.7, can be scaled to meet system throughput by using multiple processors in a single chip, each dedicated to handling a portion of the overall channel throughput. Each processor may run the exact same code, or some may change algorithms on the fly to adapt to system requirements. In some cases, a master processor is added to handle general housekeeping chores such as system initialization, statistics gathering, and error handling.

### *Serially Linked Processors*

Combining several processors in a chain, as in Figure 9.8, lets system architects treat each as a stage in a larger processing pipeline. Each CPU is responsible for one piece of the overall processing task and would share data memory (arbitrated or dedicated memory interfaces if off-chip, or dual-ported memory if on-chip) to pass results from the output of one stage to the input of the next.

### *Processor Companion Chip*

Discrete processor and DSP chips connected to an FPGA can also benefit from hardware acceleration, peripheral expansion, and interface bridging, regardless of whether a CPU is inside the FPGA. Chip-to-chip interface IP is readily available today to provide external access to peripherals, acceleration logic, and I/O interfaces contained within the FPGA.

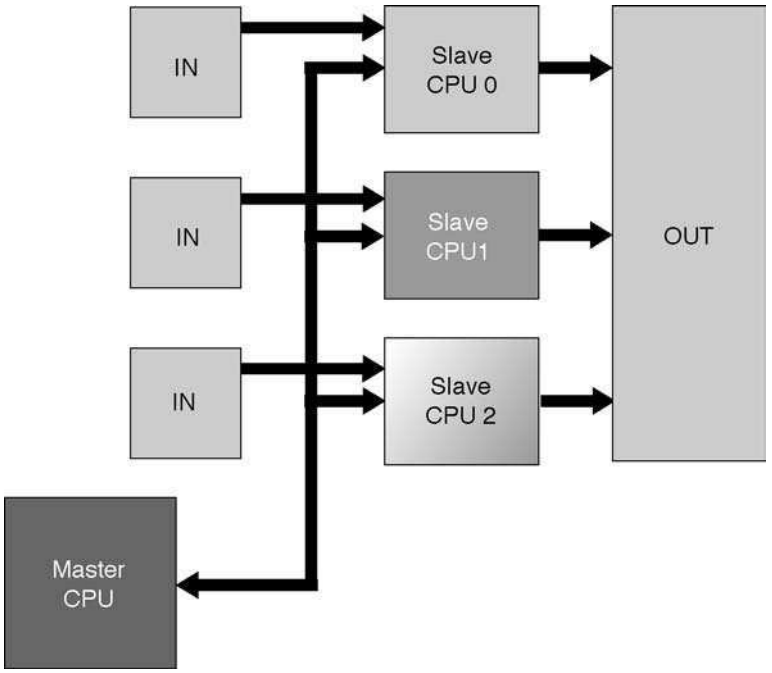



*Figure 9.7: Multichannel application*

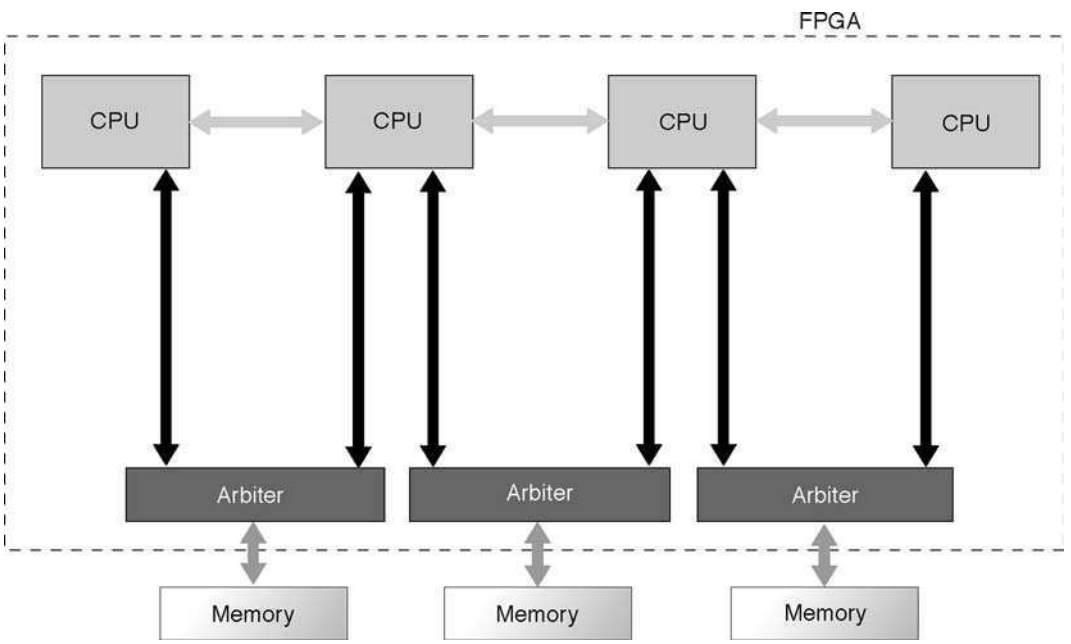



*Figure 9.8: Serially linked processors*

### Customizing to Extend Product Life

Bringing a product to market quickly often results in first-generation versions that ship with fewer features or less-optimal performance than planned. The trade-offs associated with delaying new product releases are usually not attractive. Upgrading system firmware (i.e., flash memory) to fix bugs, add features, and/or improve performance is an established practice. Systems built with programmable logic add the capability to change hardware characteristics as well, since the same flash devices that contain system software can also contain one or more FPGA configuration images. Flash devices containing system configurations are easily rewritten by a processor in the FPGA, after which it can initiate a reconfiguration that updates the system hardware in milliseconds.

Products that are network enabled now have the ability to remotely reconfigure the hardware to add new features, increase performance, and fix bugs over the Internet. While a novelty just a few years ago, this has become a common practice with FPGA-based designs. Fail-safe schemes based on multiple FPGA images stored in flash memory with a default "safe" design that loads in the case of data corruption ensures a working system device. The worst-case scenario, in which the device fails to configure, results in the default design being loaded instead, at which time it can report the failure and continue to operate as before.

### Summary

FPGA-based embedded systems open a powerful set of new options for the embedded designer. No longer are ASIC designers alone in their ability to configure a custom-tailored feature set. Now developers can change the performance characteristics of their embedded system right up to the time the product goes into final test. Now developers can extend product life cycle, getting to market quickly and upgrading both software and hardware features remotely over the Internet. (For more information see: www.altera.com.)

Mainstream application of these techniques has been enabled by a new class of hardware development tools, IP, and FPGA infrastructure. Development and growth in this area by major FPGA vendors continues at a fast pace due to the success of these technologies within the embedded community. It's time to have a look at what these possibilities could mean in your next application.

## 9.3 Xilinx MicroBlaze Soft-Core Processor

*In response to my request for articles about FPGA-optimized processors, David Vornholt of Xilinx kindly provided the following piece. (CW)*

The Xilinx MicroBlaze™ soft processor core (SPC) is a "soft processor," which uses a high-performance 32-bit RISC Harvard-style architecture. Optimized for the Xilinx FPGA fabric, it is referred to as a soft processor because the CPU, the cache memories, and any peripherals used can all be built out of on-chip look up tables (LUTs) and block RAM memory. As a result, MicroBlaze SPCs can be placed in any type of Virtex® series and Spartan-3™ series of devices, with as many as are needed to accomplish the required task. This flexibility also enables easy migration to new device families, giving protection against processor obsolescence plus higher potential future performance as the host FPGA fabric's speed improves. The MicroBlaze SPC is used in many different applications such as network equipment, telecommunication applications, control, and consumer markets. Figure 9.9 shows a typical MicroBlaze SPC implementation with its peripherals.

### MicroBlaze SPC Architecture

The processor has 32 by 32-bit registers, which feature a very short register access time. For cache, either the on-chip block RAM or off-chip memory can be used. The access time to the on-chip block RAM is minimal because there are dedicated routing resources to access them. The MicroBlaze SPC can be customized for virtually any application. Its barrel shifter, divide unit, data cache, instruction cache, and the FSL (Fast Simplex Link) bus system are optional so that performance and required resources can be selectively balanced to minimize the resources used on the FPGA—driving down the cost of implementation. (The processor typically uses 950 logic cells.) The sizes of the caches are configurable from 2 to 64 K. Standard peripherals are provided and are CoreConnect™ compatible. Consequently, they can be integrated into an embedded design very easily. These peripherals are either free, such as the memory controller, UART, interrupt controller, and timer, or are commercial cores such as the Ethernet controller, gigabit Ethernet controller, PCI, and HDLC. All commercial IP cores can be evaluated. For all free cores, the VHDL and the C code (TCP/IP stack) are readable.

The MicroBlaze SPC features a parallel pipeline, divided into three stages: Fetch, Decode, and Execute. All instructions take one clock cycle except for the following:

- Load/store: 2 clock cycles + I/O latency
- Multiply: 3 clock cycles
- Branches: 3 clock cycles; can be 1 clock cycle

The FSL interface can dramatically improve the performance of an application by outsourcing time-critical tasks into hardware on the FPGA fabric. With the FSL interface

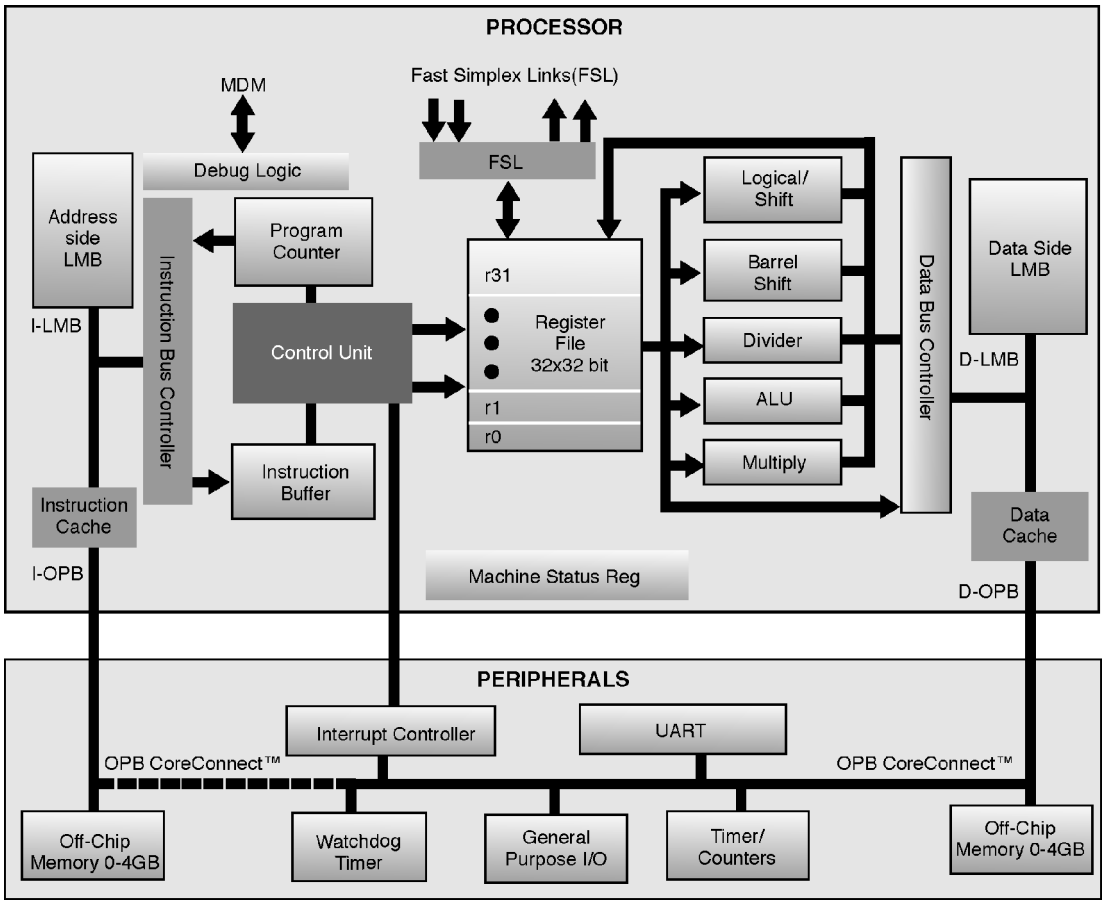



*Figure 9.9: MicroBlaze SPC implementation*

it is possible to have up to 8 inputs and 8 outputs. This allows much greater flexibility and insures that cascaded logic within the customized core doesn't affect or lock the MicroBlaze SPC RISC unit.

The FSL channels are dedicated unidirectional point-to-point data streaming interfaces. The FSL interfaces on MicroBlaze SPC are 32 bits wide. Further, the same FSL channels can be used to transmit or receive either control or data words. A separate bit indicates whether the transmitted (received) word is control or data information. The performance of the FSL interface can reach up to 300 MB/sec. (This throughput depends on the target device selected.). The FSL bus system is ideal for MicroBlaze-to-MicroBlaze or streaming I/O communications.

The main features of the FSL interface are:

- Unidirectional point-to-point communication
- Unshared nonarbitrated communication mechanism
- Control and data communication support
- FIFO-based communication
- Configurable data size
- 600 MHz standalone operation

The FSL bus is driven by one master and drives one slave. FSL peripherals may be created as a master or a slave to the FSL bus. A peripheral connected to the master ports of the FSL bus pushes data and control signals onto the FSL. All peripherals that act as a master to the FSL bus should create a bus interface of the type MASTER for the bus standard FSL in the Microprocessor Peripheral Description (MPD) file. A peripheral connected to the slave ports of the FSL bus reads and pops data and control signals from the FSL. All peripherals that are a slave to the FSL bus should create a bus interface of the type SLAVE for the bus standard FSL in the MPD file. The put and get instructions of MicroBlaze SPC can be used to transfer the contents of a MicroBlaze SPC register onto the FSL bus and vice versa. The FSL bus configuration of MicroBlaze SPC can be used in conjunction with any of the other bus configurations.

### Using MicroBlaze SPC

A reference design for greatly accelerating the performance of a one-dimensional Inverse Direct Cosine Transform (IDCT) is available showing how the implementation of a customized core can be done in software and hardware using the FSL bus for the MicroBlaze SPC.

The MicroBlaze SPC is delivered as part of the Xilinx Platform Studio EDK tool suite, a design environment for using Xilinx processors. See Figure 9.10 for the development flow.

### Other Processors

In addition to the MicroBlaze SPC, Xilinx also offers PicoBlaze, a fully embedded 8-bit microcontroller for the Virtex and Spartan series of FPGAs and CoolRunner-II CPLDs. The PicoBlaze reference design provides a small footprint and easy-to-use solution for building simple, flexible, and cost-effective controller applications.

For those who want to utilize the Virtex series immersed PowerPC™ processor while coding in VHDL or Verilog, the UltraController-II (see Figure 9.11) embedded processor solution is available as a complete reference design to be utilized as a lightweight PowerPC™ microcontroller.

The 32-bit input/32-bit output design created as a simple block, ready to integrate into larger designs, requires only a reset and a clock input. The UltraController-II solution utilizes the available PowerPC processor(s) in the Virtex-II Pro™ and new Virtex-4™

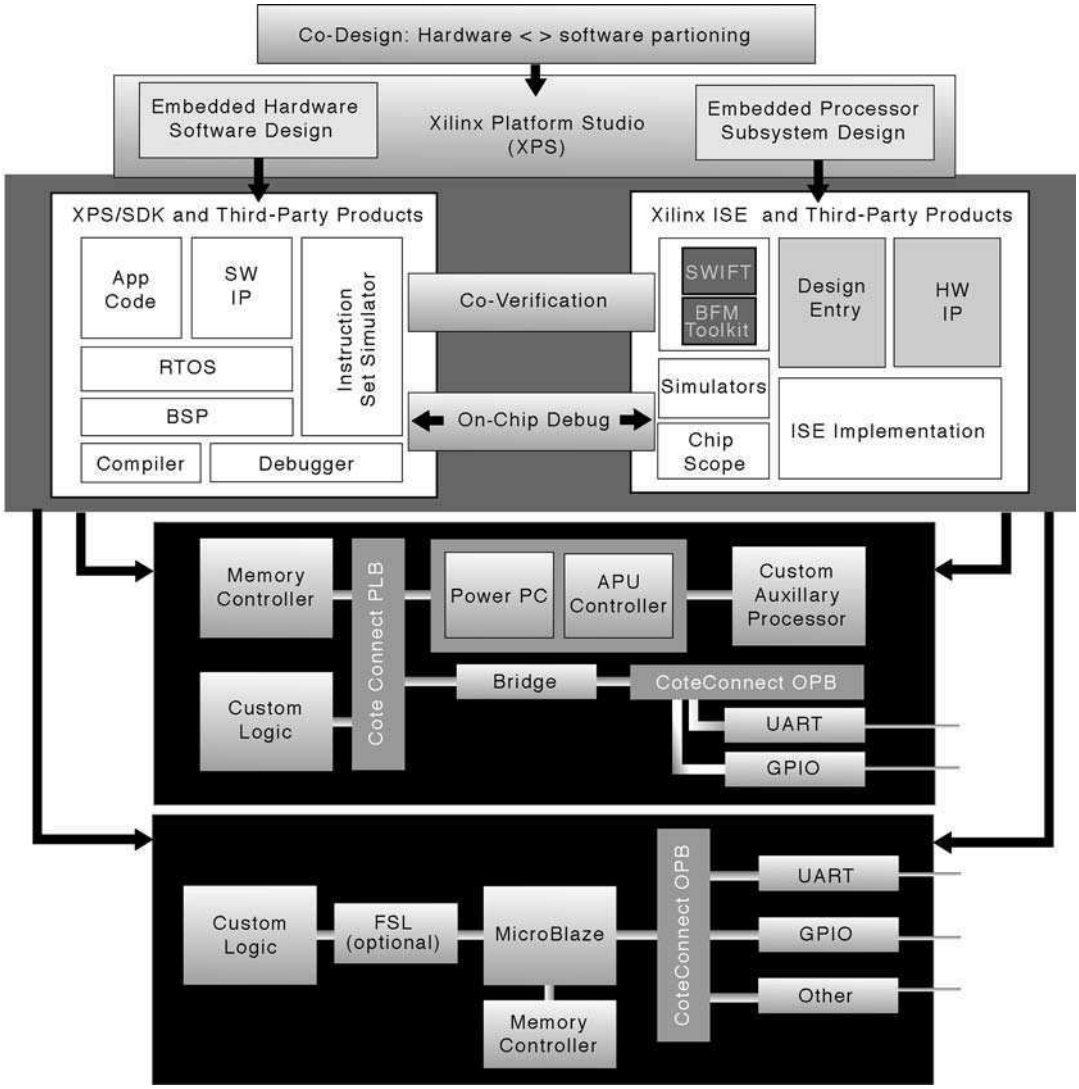



*Figure 9.10: MicroBlaze design flow*

FX device families. This small footprint engine uses minimal FPGA resources by executing code strictly from the integrated PowerPC caches that are already supplied on-chip. The UltraController-II design is available for a variety of applications including logic and data control, device configuration, system monitoring, and simple data manipulation. FPGA logic resources previously used for complex state machines or nonperformance-critical functions can now be off-loaded to the UltraController. Replacing slow

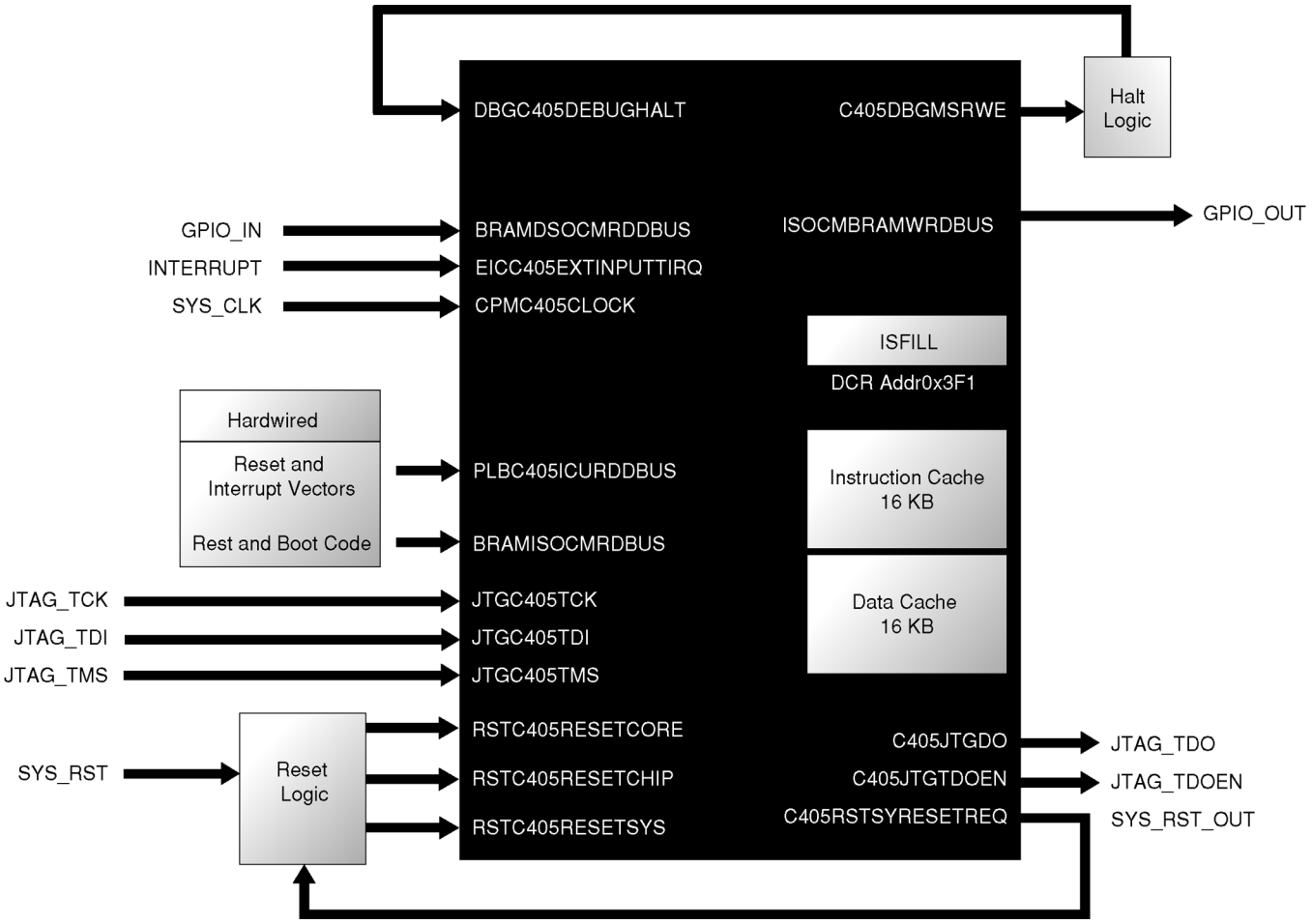



*Figure 9.11: UltraController-II*

logic with code by using the UltraController-II enables migration to a smaller device with good cost savings.

## Conclusions

This new generation of FPGAs has a variety of solutions for hardware-oriented designs as well as traditional software-centric embedded developers, ranging from 8 to 32 bits. For more information see: www.xilinx.com.

## 9.4 Real-Time Operating Systems for FPGA

*I wrote this article in 2003 for an electronic newsletter published by another division of Mentor Graphics, which was usually concerned solely with FPGA development—RTOSs were very foreign to them at that time. For more background on FPGAs, see my article "FPGAs and Processor Cores: The Future of Embedded Systems?" earlier in this chapter. (CW)*

Until a couple of years ago, the idea of mentioning an RTOS and an FPGA in the same sentence would not have occurred to anyone. The average software engineer had little idea what an FPGA was, and FPGA designers could not have cared less about RTOSs. So, what changed? Two things did:

- The major FPGA vendors announced new devices, which incorporated powerful processor cores (ARM or PowerPC) onto programmable logic devices.
- A number of soft-core processors (devices designed to be instantiated into the FPGA fabric) were announced (notably Nios from Altera and MicroBlaze from Xilinx).

Until this time, embedded software engineers had either worked with discrete processor chips on boards with memory and peripheral devices, or they had been involved in system on chip (SoC) designs. Now there was a third context in which they may be developing code: often termed the "field-programmable system on chip" (FPSoC). In this article, we introduce some requirements for an RTOS in this new context.

### Writing Software for FPSoC

Embedded programmers working with board-based designs had very few constraints: plentiful memory, readily available debug access to the processor, standard peripheral devices (that presented few programming challenges), and promptly available hardware. Moving to an SoC was a shock: limited memory, low processor visibility, lots of custom logic (for which drivers needed to be written), and long lead times to available hardware.

FPSoC is an interesting middle ground. Some of the SoC constraints are there, but the technology offers access to hardware much earlier in the design cycle. Furthermore, since the hardware can be changed readily, the bad (but necessary) practice of patching the code to get around hardware design bugs could be eliminated.

### RTOS Requirements

At the core of the software in any modern embedded design is an RTOS, which provides structure and control to the application as a whole. Typically, an RTOS will be licensed from an appropriate vendor.

The unique architecture of FPSoC presents some interesting challenges in selecting an RTOS:

#### *Functionality*

Since the embedded applications are not becoming any less complex, there is no opportunity to sacrifice functionality, and a well-rounded RTOS product is a must.

Furthermore, optional components, like networking protocols, file systems, and graphics, may be required. But they should certainly be *optional*—there is no room for excess memory footprint; see the section "Size" later in this article.

### Speed and Real-Time Response

It will depend somewhat upon the application, but many embedded applications need to be fast and exhibit a real-time response. Broadly, this means that the RTOS must be completely predictable (deterministic) in its behavior.

### Size

On an FPSoC, as on an SoC, memory is at a premium, so the RTOS must utilize it wisely. This is not simply a matter of being small. There must be a means to ensure that no excess memory is consumed by unwanted RTOS functionality. This is addressed in two ways:

- Layered products must be true options. If a file system, for example, is not required for the application, this component should be omitted from the RTOS memory image.
- The RTOS (and layered products) should be scalable. This means that it is provided as a library, from which only the required components are extracted. This is illustrated in Figure 9.12.

### Processor Support

It is very desirable to have an RTOS that is available on a wide range of processor architectures. The benefit of such coverage is that, in a given company, many processor architectures may be in use, either now or over time. There may even be multiple processors in a given design (like a PowerPC and multiple MicroBlaze devices, for example). The advantage of being able to use a single RTOS product line is associated with the engineers' skill set. Once they are familiar with a particular RTOS programming interface, they can reuse that knowledge extensively; there is no more learning curve to ascend.

### Debug Capabilities

It is essential to have some RTOS-aware debug capabilities and most commercial RTOS products will have some such provision. However, this issue is not so black and white. There are many possible types and levels of debug functionality, and this topic could fill an article by itself. Suffice it to say that opportunities to leverage the FPSoC architecture should not be missed.

### Business Model

Applications for FPSoC devices are many and varied. Although this variation is obvious from a technical perspective, it is commercially just as significant. The business model of the RTOS must match the requirements of the FPSoC application. The key is *flexibility*.

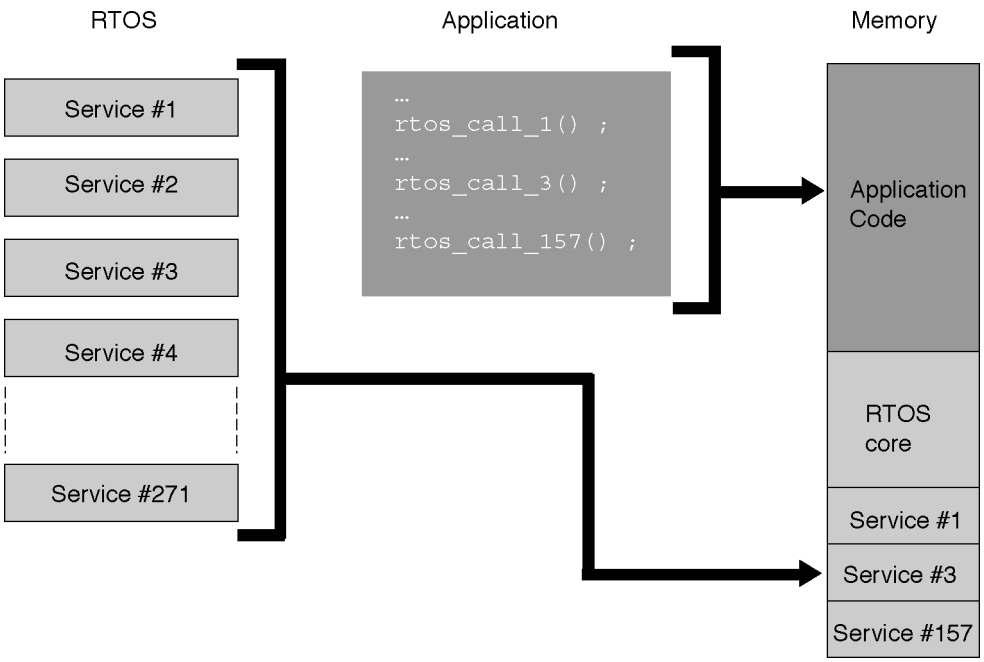



*Figure 9.12: RTOS scalability*

Although some applications are low volume and will find a royalty-based licensing scheme acceptable, FPSoC is increasingly being used in high volumes, where a royalty-free model is more appropriate. In other contexts, different approaches, such as subscription models, make more sense.

There is a curious synergy: the FPGA soft-core devices are mainly offered on a royalty-free basis; a royalty-free RTOS therefore seems a logical fit.

## Conclusions

It is early days for FPSoC, but it is clear that FPGA technology and embedded software, including RTOSs, will be considered together increasingly in the coming years.

# Afterword

## Great Expectations

Colin Walls

*To close, here's a brief story that might make you think. (CW)*

---

Unless it is now 2025 and you have found this book in a remainders bookshop, you are probably a child of the twentieth century. As such, you have perceptions and expectations about the world around you in general and embedded systems in particular. Take a moment to consider the expectations of citizens of the twenty-first century.

Meet Archie.

At the time this photo was taken, he had just celebrated his third birthday, so he is very much a citizen of the twenty-first century. Archie is interested in embedded systems. For example, he loves having his photo taken. Just produce a camera, and he is ready to pose for a snapshot. And, as you can see, he makes a good photograph.

As a citizen of the twenty-first century, however, he has expectations. Having had his picture taken, he wants to see results—*right away*. He expects to immediately see the photograph on the LCD on the back of the camera.

If you were to take his photograph with an old-fashioned film camera and say "It's OK Archie, we'll get the pictures back tomorrow," he would be unimpressed. Unimpressed three year olds are not fun to be with.

So, when you are designing your next embedded system, think about the expectations of your users. And think about Archie.

# Index

*This index is designed to lead you to the article(s) that cover the topic you have looked up. If the page number is **bold**, it indicates that this is a key topic of the article. In most cases, the page references refer you to the first occurrence of a term in a given article.*

# ELSEVIER SCIENCE CD-ROM LICENSE AGREEMENT

PLEASE READ THE FOLLOWING AGREEMENT CAREFULLY BEFORE USING THIS CD-ROM PRODUCT. THIS CD-ROM PRODUCT IS LICENSED UNDER THE TERMS CONTAINED IN THIS CD-ROM LICENSE AGREEMENT ("Agreement"). BY USING THIS CD-ROM PRODUCT, YOU, AN INDIVIDUAL OR ENTITY INCLUDING EMPLOYEES, AGENTS AND REPRESENTATIVES ("You" or "Your"), ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, THAT YOU UNDERSTAND IT, AND THAT YOU AGREE TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS AGREEMENT. ELSEVIER SCIENCE INC. ("Elsevier Science") EXPRESSLY DOES NOT AGREE TO LICENSE THIS CD-ROM PRODUCT TO YOU UNLESS YOU ASSENT TO THIS AGREEMENT. IF YOU DO NOT AGREE WITH ANY OF THE FOLLOWING TERMS, YOU MAY, WITHIN THIRTY (30) DAYS AFTER YOUR RECEIPT OF THIS CD-ROM PRODUCT RETURN THE UNUSED CD-ROM PRODUCT AND ALL ACCOMPANYING DOCUMENTATION TO ELSEVIER SCIENCE FOR A FULL REFUND.

## DEFINITIONS

As used in this Agreement, these terms shall have the following meanings:

"Proprietary Material" means the valuable and proprietary information content of this CD-ROM Product including all indexes and graphic materials and software used to access, index, search and retrieve the information content from this CD-ROM Product developed or licensed by Elsevier Science and/or its affiliates, suppliers and licensors.

"CD-ROM Product" means the copy of the Proprietary Material and any other material delivered on CD-ROM and any other human-readable or machine-readable materials enclosed with this Agreement, including without limitation documentation relating to the same.

## OWNERSHIP

This CD-ROM Product has been supplied by and is proprietary to Elsevier Science and/or its affiliates, suppliers and licensors. The copyright in the CD-ROM Product belongs to Elsevier Science and/or its affiliates, suppliers and licensors and is protected by the national and state copyright, trademark, trade secret and other intellectual property laws of the United States and international treaty provisions, including without limitation the Universal Copyright Convention and the Berne Copyright Convention. You have no ownership rights in this CD-ROM Product. Except as expressly set forth herein, no part of this CD-ROM Product, including without limitation the Proprietary Material, may be modified, copied or distributed in hardcopy or machine-readable form without prior written consent from Elsevier Science. All rights not expressly granted to You herein are expressly reserved. Any other use of this CD-ROM Product by any person or entity is strictly prohibited and a violation of this Agreement.

## SCOPE OF RIGHTS LICENSED (PERMITTED USES)

Elsevier Science is granting to You a limited, non-exclusive, non-transferable license to use this CD-ROM Product in accordance with the terms of this Agreement. You may use or provide access to this CD-ROM Product on a single computer or terminal physically located at Your premises and in a secure network or move this CD-ROM Product to and use it on another single computer or terminal at the same location for personal use only, but under no circumstances may You use or provide access to any part or parts of this CD-ROM Product on more than one computer or terminal simultaneously.

You shall not (a) copy, download, or otherwise reproduce the CD-ROM Product in any medium, including, without limitation, online transmissions, local area networks, wide area networks, intranets, extranets and the Internet, or in any way, in whole or in part, except that You may print or download limited portions of the Proprietary Material that are the results of discrete searches; (b) alter, modify, or adapt the CD-ROM Product, including but not limited to decompiling, disassembling, reverse engineering, or creating derivative works, without the prior written approval of Elsevier Science; (c) sell, license or otherwise distribute to third parties the CD-ROM Product or any part or parts thereof; or (d) alter, remove, obscure or obstruct the display of any copyright, trademark or other proprietary notice on or in the CD-ROM Product or on any printout or download of portions of the Proprietary Materials.

## RESTRICTIONS ON TRANSFER

This License is personal to You, and neither Your rights hereunder nor the tangible embodiments of this CD-ROM Product, including without limitation the Proprietary Material, may be sold, assigned, transferred or sub-licensed to any other person, including without limitation by operation of law, without the prior written consent of Elsevier Science. Any purported sale, assignment, transfer or sublicense without the prior written consent of Elsevier Science will be void and will automatically terminate the License granted hereunder.

TERM

This Agreement will remain in effect until terminated pursuant to the terms of this Agreement. You may terminate this Agreement at any time by removing from Your system and destroying the CD-ROM Product. Unauthorized copying of the CD-ROM Product, including without limitation, the Proprietary Material and documentation, or otherwise failing to comply with the terms and conditions of this Agreement shall result in automatic termination of this license and will make available to Elsevier Science legal remedies. Upon termination of this Agreement, the license granted herein will terminate and You must immediately destroy the CD-ROM Product and accompanying documentation. All provisions relating to proprietary rights shall survive termination of this Agreement.

LIMITED WARRANTY AND LIMITATION OF LIABILITY

NEITHER ELSEVIER SCIENCE NOR ITS LICENSORS REPRESENT OR WARRANT THAT THE INFORMATION CONTAINED IN THE PROPRIETARY MATERIALS IS COMPLETE OR FREE FROM ERROR, AND NEITHER ASSUMES, AND BOTH EXPRESSLY DISCLAIM, ANY LIABILITY TO ANY PERSON FOR ANY LOSS OR DAMAGE CAUSED BY ERRORS OR OMISSIONS IN THE PROPRIETARY MATERIAL, WHETHER SUCH ERRORS OR OMISSIONS RESULT FROM NEGLIGENCE, ACCIDENT, OR ANY OTHER CAUSE. IN ADDITION, NEITHER ELSEVIER SCIENCE NOR ITS LICENSORS MAKE ANY REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE PERFORMANCE OF YOUR NETWORK OR COMPUTER SYSTEM WHEN USED IN CONJUNCTION WITH THE CD-ROM PRODUCT.

If this CD-ROM Product is defective, Elsevier Science will replace it at no charge if the defective CD-ROM Product is returned to Elsevier Science within sixty (60) days (or the greatest period allowable by applicable law) from the date of shipment.

Elsevier Science warrants that the software embodied in this CD-ROM Product will perform in substantial compliance with the documentation supplied in this CD-ROM Product. If You report significant defect in performance in writing to Elsevier Science, and Elsevier Science is not able to correct same within sixty (60) days after its receipt of Your notification, You may return this CD-ROM Product, including all copies and documentation, to Elsevier Science and Elsevier Science will refund Your money.

YOU UNDERSTANT THAT, EXCEPT FOR THE 60-DAY LIMITED WARRANTY RECITED ABOVE, ELSEVIER SCIENCE, ITS AFFILIATES, LICENSORS, SUPPLIERS AND AGENTS, MAKE NO WARRANTIES, EXPRESSED OR IMPLIED, WITH RESPECT TO THE CD-ROM PRODUCT, INCLUDING, WITHOUT LIMITATION THE PROPRIETARY MATERIAL, AND SPECIFICALLY DISCLAIM ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

If the information provided on this CD-ROM contains medical or health sciences information, it is intended for professional use within the medical field. Information about medical treatment or drug dosages is intended strictly for professional use, and because of rapid advances in the medical sciences, independent verification of diagnosis and drug dosages should be made.

IN NO EVENT WILL ELSEVIER SCIENCE, ITS AFFILIATES, LICENSORS, SUPPLIERS OR AGENTS, BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES, ARISING OUT OF YOUR USE OR INABILITY TO USE THE CD-ROM PRODUCT REGARDLESS OF WHETHER SUCH DAMAGES ARE FORESEEABLE OR WHETHER SUCH DAMAGES ARE DEEMED TO RESULT FROM THE FAILURE OR INADEQUACY OF ANY EXCLUSIVE OR OTHER REMEDY.

U.S. GOVERNMENT RESTRICTED RIGHTS

The CD-ROM Product and documentation are provided with restricted rights. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraphs (a) through (d) of the Commercial Computer Restricted Rights clause at FAR 52.22719 or in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.2277013, or at 252.2117015, as applicable. Contractor/Manufacturer is Elsevier Science Inc., 655 Avenue of the Americas, New York, NY 10010-5107 USA.

GOVERNING LAW

This Agreement shall be governed by the laws of the State of New York, USA. In any dispute arising out of this Agreement, you and Elsevier Science each consent to the exclusive personal jurisdiction and venue in the state and federal courts within New York County, New York, USA.