

Real-Time Concepts for Embedded Systems

Qing Li
with **Caroline Yao**

Published by CMP Books
an imprint of CMP Media LLC
Main office: 600 Harrison Street, San Francisco, CA 94107 USA
Tel: 415-947-6615; fax: 415-947-6015
Editorial office: 1601 West 23rd Street, Suite 200, Lawrence, KS 66046 USA
www.cmpbooks.com
email: books@cmp.com

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

Copyright © 2003 by Wind River Systems, Inc., except where noted otherwise. Published by CMP Books, CMP Media LLC. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

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

Technical editors: Robert Ward and Marc Briand
Copyeditor: Catherine Janzen
Layout design & production: Madeleine Reardon Dimond and Michelle O'Neal
Managing editor: Michelle O'Neal
Cover art design: Damien Castaneda

Distributed to the book trade in the U.S. by:

Publishers Group West
Berkeley, CA 94710
1-800-788-3123

Distributed in Canada by:

Jaguar Book Group
100 Armstrong Avenue
Georgetown, Ontario M6K 3E7 Canada
905-877-4483

For individual orders and for information on special discounts for quantity orders, please contact:

CMP Books Distribution Center, 6600 Silacci Way, Gilroy, CA 95020
Tel: 1-800-500-6875 or 408-848-3854; fax: 408-848-5784
email: cmp@rushorder.com; Web: www.cmpbooks.com

Library of Congress Cataloging-in-Publication Data

Li, Qing, 1971-

Real-time concepts for embedded systems / Qing Li ; with Caroline Yao.
p. cm.

Includes bibliographical references and index.

ISBN 1-57820-124-1 (alk. paper)

1. Embedded computer systems. 2. Real-time programming. I. Yao, Caroline. II. Title.

Tk7895.E42L494 2003

004'.33-dc21

2003008483

Printed in the United States of America

03 04 05 06 07 5 4 3 2 1

To my wife, Huaying, and my daughter, Jane, for their love, understanding, and support.

To my parents, Dr. Y. H. and Dr. N. H. Li, and my brother, Dr. Yang Li, for being the exemplification of academic excellence.

ISBN: 1-57820-124-1

About the Authors

Qing Li is currently a senior architect at Wind River systems and has four patents pending in the embedded kernel and networking protocol design areas. His 12+ years in engineering include expertise as a principal engineer designing and developing protocol stacks and embedded applications for the telecommunications and networks arena. Qing is the lead architect of Wind River's embedded IPv6 products and is at the forefront of various IPv6 initiatives. In the past, Qing owned his own company developing commercial software for the telecommunications industry. Additionally, he was one of a four-member Silicon Valley startup that designed and developed proprietary algorithms and applications for embedded biometric devices in the security industry.

Qing holds a Bachelor of Science degree with Specialization in Computing Science from the University of Alberta in Edmonton, Alberta, Canada. Qing has a Masters of Science degree with Distinction in Computer Engineering, with focus in Advanced High Performance Computing from Santa Clara University, Santa Clara, CA, USA. Qing is a member of Association for Computing Machinery and a member of IEEE Computer Society.

Caroline Yao has 15+ years in technology and the commercial software arena with six years in the embedded market. She has expertise ranging from product development, product management, product marketing, business development, and strategic alliances. She is also a co-inventor and co-US patent pending (June 12, 2001) holder for 'System and Method for Providing Cross-Development Application Design Tools and Services Via a Network.'

Caroline holds a Bachelor of Arts in Statistics from the University of California Berkeley.

Foreword

We live in a world today in which software plays a critical part. The most critical software is not running on large systems and PCs. Rather, it runs inside the infrastructure and in the devices that we use every day. Our transportation, communications, and energy systems won't work if the embedded software contained in our cars, phones, routers and power plants crashes.

The design of this invisible, embedded software is crucial to all of us. Yet, there has been a real shortage of good information as to effective design and implementation practices specific to this very different world. Make no mistake, it is indeed different and often more difficult to design embedded software than more traditional programs. Time, and the interaction of multiple tasks in real-time, must be managed. Seemingly esoteric concepts, such as priority inversion, can become concrete in a hurry when they bring a device to its knees. Efficiency—a small memory footprint and the ability to run on lower cost hardware—become key design considerations because they directly affect cost, power usage, size, and battery life. Of course, reliability is paramount when so much is at stake—company and product reputations, critical infrastructure functions, and, some times, even lives.

Mr. Li has done a marvelous job of pulling together the relevant information. He lays out the issues, the decision and design process, and the available tools and methods. The latter part of the book provides valuable insights and practical experiences in understanding application development, common design problems, and solutions. The book will be helpful to anyone embarking on an embedded design project, but will be of particular help to engineers who are experienced in software development but not yet in real-time and embedded software development. It is also a wonderful text or reference volume for academic use.

The quality of the pervasive, invisible software surrounding us will determine much about the world being created today. This book will have a positive effect on that quality and is a welcome addition to the engineering bookshelf.

Jerry Fiddler
Chairman and Co-Founder, Wind River

Acknowledgments

We would like to thank the team at CMP Books and especially Paul Temme, Michelle O'Neal, Marc Briand, Brandy Erzen, and Robert Ward.

We wish to express our thanks to the reviewers Jerry Krasner, Shin Miyakawa, Jun-ichiro Itojun Hagino, and Liliana Britvic for their contributions.

We would like to thank Nauman Arshad for his initial participation on this project.

We would also like to thank Anne-Marie Eileraas, Salvatore LiRosi, Loren Shade, and numerous other individuals at Wind River for their support.

Finally, thanks go to our individual families for their love and support, Huaying and Jane Lee, Maya and William Yao.

Chapter 1: Introduction

Overview

In ways virtually unimaginable just a few decades ago, embedded systems are reshaping the way people live, work, and play. Embedded systems come in an endless variety of types, each exhibiting unique characteristics. For example, most vehicles driven today embed intelligent computer chips that perform value-added tasks, which make the vehicles easier, cleaner, and more fun to drive. Telephone systems rely on multiple integrated hardware and software systems to connect people around the world. Even private homes are being filled with intelligent appliances and integrated systems built around embedded systems, which facilitate and enhance everyday life.

Often referred to as *pervasive* or *ubiquitous* computers, embedded systems represent a class of dedicated computer systems designed for specific purposes. Many of these embedded systems are reliable and predictable. The devices that embed them are convenient, user-friendly, and dependable.

One special class of embedded systems is distinguished from the rest by its requirement to respond to external events in real time. This category is classified as the *real-time embedded system*.

As an introduction to embedded systems and real-time embedded systems, this chapter focuses on:

- § examples of embedded systems,
- § defining embedded systems,
- § defining embedded systems with real-time behavior, and
- § current trends in embedded systems.

1.1 Real Life Examples of Embedded Systems

Even though often nearly invisible, embedded systems are ubiquitous. Embedded systems are present in many industries, including industrial automation, defense, transportation, and aerospace. For example, NASA's Mars Path Finder, Lockheed Martin's missile guidance system, and the Ford automobile all contain numerous embedded systems.

Every day, people throughout the world use embedded systems without even knowing it. In fact, the embedded system's invisibility is its very beauty: users reap the advantages without having to understand the intricacies of the technology.

Remarkably adaptable and versatile, embedded systems can be found at home, at work, and even in recreational devices. Indeed, it is difficult to find a segment of daily life that does not involve embedded systems in some way. Some of the more visible examples of embedded systems are provided in the next sections.

1.1.1 Embedded Systems in the Home Environment

Hidden conveniently within numerous household appliances, embedded systems are found all over the house. Consumers enjoy the effort-saving advanced features and benefits provided by these embedded technologies.

As shown in [Figure 1.1](#) embedded systems in the home assume many forms, including security systems, cable and satellite boxes for televisions, home theater systems, and telephone answering machines. As advances in microprocessors continue to improve the functionality of

ordinary products, embedded systems are helping drive the development of additional home-based innovations.



Figure 1.1: Embedded systems at home.

1.1.2 Embedded Systems in the Work Environment

Embedded systems have also changed the way people conduct business. Perhaps the most significant example is the Internet, which is really just a very large collection of embedded systems that are interconnected using various networking technologies. [Figure 1.2](#) illustrates what a small segment of the Internet might look like.

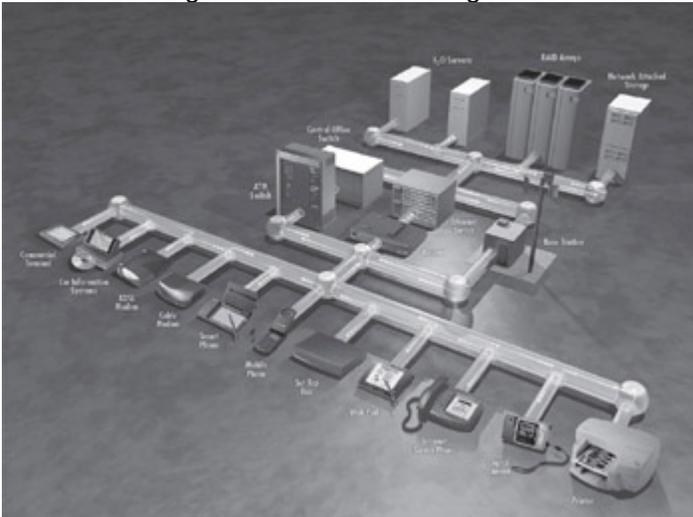


Figure 1.2: Embedded systems at work.

From various individual network end-points (for example, printers, cable modems, and enterprise network routers) to the backbone gigabit switches, embedded technology has helped make use of the Internet necessary to any business model. The network routers and the backbone gigabit switches are examples of real-time embedded systems. Advancements in real-time embedded technology are making Internet connectivity both reliable and responsive, despite the enormous amount of voice and data traffic carried over the network.

1.1.3 Embedded Systems in Leisure Activities

At home, at work, even at play, embedded systems are flourishing. A child's toy unexpectedly springs to life with unabashed liveliness. Automobiles equipped with in-car navigation systems transport people to destinations safely and efficiently. Listening to favorite tunes with anytime-anywhere freedom is readily achievable, thanks to embedded systems buried deep within sophisticated portable music players, as shown in [Figure 1.3](#).



Figure 1.3: Navigation system and portable music player.

Even the portable computing device, called a *web tablet*, shown in [Figure 1.4](#), is an embedded system.



Figure 1.4: A web tablet.

Embedded systems also have teamed with other technologies to deliver benefits to the traditionally low-tech world. GPS technology, for example, uses satellites to pinpoint locations to centimeter-level accuracy, which allows hikers, cyclists, and other outdoor enthusiasts to use GPS handheld devices to enjoy vast spaces without getting lost. Even fishermen use GPS devices to store the locations of their favorite fishing holes.

Embedded systems also have taken traditional radio-controlled airplanes, racecars, and boats to new heights...and speeds. As complex embedded systems in disguise, these devices take command inputs from joysticks and pass them wirelessly to the device's receiver, enabling the model airplane, racecar, or boat to engage in speedy and complex maneuvers. In fact, the introduction of embedded technology has rendered these sports safer and more enjoyable for model owners by virtually eliminating the once-common threat of crashing due to signal interference.

1.1.4 Defining the Embedded System

Some texts define embedded systems as computing systems or devices without a keyboard, display, or mouse. These texts use the “look” characteristic as the differentiating factor by saying, “embedded systems do not look like ordinary personal computers; they look like digital cameras or smart toasters.” These statements are all misleading.

A general definition of *embedded systems* is: embedded systems are computing systems with tightly coupled hardware and software integration, that are designed to perform a dedicated function. The word embedded reflects the fact that these systems are usually an integral part of a

larger system, known as the embedding system. Multiple embedded systems can coexist in an embedding system.

This definition is good but subjective. In the majority of cases, embedded systems are truly embedded, i.e., they are “systems within systems.” They either cannot or do not function on their own. Take, for example, the digital set-top box (DST) found in many home entertainment systems nowadays. The digital audio/video decoding system, called the *A/V decoder*, which is an integral part of the DST, is an embedded system. The A/V decoder accepts a single multimedia stream and produces sound and video frames as output. The signals received from the satellite by the DST contain multiple streams or channels. Therefore, the A/V decoder works in conjunction with the transport stream decoder, which is yet another embedded system. The transport stream decoder de-multiplexes the incoming multimedia streams into separate channels and feeds only the selected channel to the A/V decoder.

In some cases, embedded systems can function as standalone systems. The network router illustrated in [Figure 1.2](#) is a standalone embedded system. It is built using a specialized communication processor, memory, a number of network access interfaces (known as network ports), and special software that implements packet routing algorithms. In other words, the network router is a standalone embedded system that routes packets coming from one port to another, based on a programmed routing algorithm.

The definition also does not necessarily provide answers to some often-asked questions. For example: “Can a personal computer be classified as an embedded system? Why? Can an Apple iBook that is used only as a DVD player be called an embedded system?”

A single comprehensive definition does not exist. Therefore, we need to focus on the characteristics of embedded systems from many different perspectives to gain a real understanding of what embedded systems are and what makes embedded systems special.

1.1.5 Embedded Processor and Application Awareness

The processors found in common personal computers (PC) are general-purpose or universal processors. They are complex in design because these processors provide a full scale of features and a wide spectrum of functionalities. They are designed to be suitable for a variety of applications. The systems using these universal processors are programmed with a multitude of applications. For example, modern processors have a built-in memory management unit (MMU) to provide memory protection and virtual memory for multitasking-capable, general-purpose operating systems. These universal processors have advanced cache logic. Many of these processors have a built-in math co-processor capable of performing fast floating-point operations. These processors provide interfaces to support a variety of external peripheral devices. These processors result in large power consumption, heat production, and size. The complexity means these processors are also expensive to fabricate. In the early days, embedded systems were commonly built using general-purpose processors.

Because of the quantum leap in advancements made in microprocessor technology in recent years, embedded systems are increasingly being built using embedded processors instead of general-purpose processors. These embedded processors are special-purpose processors designed for a specific class of applications. The key is application awareness, i.e., knowing the nature of the applications and meeting the requirement for those applications that it is designed to run.

One class of embedded processors focuses on size, power consumption, and price. Therefore, some embedded processors are limited in functionality, i.e., a processor is good enough for the class of applications for which it was designed but is likely inadequate for other classes of applications. This is one reason why many embedded processors do not have fast CPU speeds. For example, the processor chosen for a personal digital assistant (PDA) device does not have a

floating-point co-processor because floating-point operations are either not needed or software emulation is sufficient. The processor might have a 16-bit addressing architecture instead of 32-bit, due to its limited memory storage capacity. It might have a 200MHz CPU speed because the majority of the applications are interactive and display-intensive, rather than computation-intensive. This class of embedded processors is small because the overall PDA device is slim and fits in the palm of your hand. The limited functionality means reduced power consumption and long-lasting battery life. The smaller size reduces the overall cost of processor fabrication.

On the other hand, another class of embedded processors focuses on performance. These embedded processors are powerful and packed with advanced chip-design technologies, such as advanced pipeline and parallel processing architecture. These processors are designed to satisfy those applications with intensive computing requirements not achievable with general-purpose processors. An emerging class of highly specialized and high-performance embedded processors includes network processors developed for the network equipment and telecommunications industry. Overall, system and application speeds are the main concerns.

Yet another class of embedded processors focuses on all four requirements—performance, size, power consumption, and price. Take, for example, the embedded digital signal processor (DSP) used in cell phones. Real-time voice communication involves digital signal processing and cannot tolerate delays. A DSP has specialized arithmetic units, optimized design in the memory, and addressing and bus architectures with multiprocessing capability that allow the DSP to perform complex calculations extremely fast in real time. A DSP outperforms a general-purpose processor running at the same clock speed many times over comes to digital signal processing. These reasons are why DSPs, instead of general-purpose processors, are chosen for cell phone designs. Even though DSPs are incredibly fast and powerful embedded processors, they are reasonably priced, which keeps the overall prices of cell phones competitive. The battery from which the DSP draws power lasts for hours and hours. A cell phone under \$100 fits in half the palm-size of an average person at the time this book was written.

System-on-a-chip (SoC) processors are especially attractive for embedded systems. The SoC processor is comprised of a CPU core with built-in peripheral modules, such as a programmable general-purpose timer, programmable interrupt controller, DMA controller, and possibly Ethernet interfaces. Such a self-contained design allows these embedded processors to be used to build a variety of embedded applications without needing additional external peripheral devices, again reducing the overall cost and size of the final product.

Sometimes a gray area exists when using processor type to differentiate between embedded and non-embedded systems. It is worth noting that, in large-scale, high-performance embedded systems, the choice between embedded processors and universal microprocessors is a difficult one.

In high-end embedded systems, system performance in a predefined context outweighs power consumption and cost. The choice of a high-end, general purpose processor is as good as the choice of a high-end, specialized embedded processor in some designs. Therefore, using processor type alone to classify embedded systems may result in wrong classifications.

1.1.6 Hardware and Software Co-Design Model

Commonly both the hardware and the software for an embedded system are developed in parallel. Constant design feedback between the two design teams should occur in this development model. The result is that each side can take advantage of what the other can do. The software component can take advantage of special hardware features to gain performance. The hardware component can simplify module design if functionality can be achieved in software that reduces overall hardware complexity and cost. Often design flaws, in both the hardware and software, are uncovered during this close collaboration.

The hardware and software co-design model reemphasizes the fundamental characteristic of embedded systems—they are application-specific. An embedded system is usually built on custom hardware and software. Therefore, using this development model is both permissible and beneficial.

1.1.7 Cross-Platform Development

Another typical characteristic of embedded systems is its method of software development, called *cross-platform development*, for both system and application software. Software for an embedded system is developed on one platform but runs on another. In this context, the *platform* is the combination of hardware (such as particular type of processor), operating system, and software development tools used for further development.

The *host system* is the system on which the embedded software is developed. The *target system* is the embedded system under development.

The main software tool that makes cross-platform development possible is a cross compiler. A *cross compiler* is a compiler that runs on one type of processor architecture but produces object code for a different type of processor architecture. A cross compiler is used because the target system cannot host its own compiler. For example, the DIAB compiler from Wind River Systems is such a cross compiler. The DIAB compiler runs on the Microsoft Windows operating system (OS) on the IA-32 architecture and runs on various UNIX operating systems, such as the Solaris OS on the SPARC architecture. The compiler can produce object code for numerous processor types, such as Motorola's 68000, MIPS, and ARM. We discuss more cross-development tools in [Chapter 2](#).

1.1.8 Software Storage and Upgradeability

Code for embedded systems (such as the real-time embedded operating system, the system software, and the application software) is commonly stored in ROM and NVRAM memory devices. In [Chapter 3](#), we discuss the embedded system booting process and the steps involved in extracting code from these storage devices. Upgrading an embedded system can mean building new PROM, deploying special equipment and/or a special method to reprogram the EPROM, or reprogramming the flash memory.

The choice of software storage device has an impact on development. The process to reprogram an EPROM when small changes are made in the software can be tedious and time-consuming, and this occurrence is common during development. Removing an EPROM device from its socket can damage the EPROM; worse yet, the system itself can be damaged if careful handling is not exercised.

The choice of the storage device can also have an impact on the overall cost of maintenance. Although PROM and EPROM devices are inexpensive, the cost can add up if a large volume of shipped systems is in the field. Upgrading an embedded system in these cases means shipping replacement PROM and EPROM chips. The embedded system can be upgraded without the need for chip replacement and can be upgraded dynamically over a network if flash memory or EEPROM is used as the code storage device (see the following sidebar).

Armed with the information presented in the previous sections, we can now attempt to answer the questions raised earlier. A personal computer is not an embedded system because it is built using a general-purpose processor and is built independently from the software that runs on it. The software applications developed for personal computers, which run operating systems such as FreeBSD or Windows, are developed natively (as opposed to cross-developed) on those operating systems. For the same reasons, an Apple iBook used only as a DVD player is used like an embedded system but is not an embedded system.

Read Only Memory (ROM)

With non-volatile content and without the need for an external power source.

- § **Mask Programmed ROM**—the memory content is programmed during the manufacturing process. Once programmed, the content cannot be changed. It cannot be reprogrammed.
- § **Field Programmable ROM (PROM)**—the memory content can be custom-programmed one time. The memory content cannot change once programmed.
- § **Erasable Programmable ROM (EPROM)**—an EPROM device can be custom-programmed, erased, and reprogrammed as often as required within its lifetime (hundreds or even thousands of times). The memory content is non-volatile once programmed. Traditional EPROM devices are erased by exposure to ultraviolet (UV) light. An EPROM device must be removed from its housing unit first. It is then reprogrammed using a special hardware device called an EPROM programmer.
- § **Electrically Erasable Programmable ROM (EEPROM or E2PROM)**—modern EPROM devices are erased electrically and are thus called EEPROM. One important difference between an EPROM and an EEPROM device is that with the EEPROM device, memory content of a single byte can be selectively erased and reprogrammed. Therefore, with an EEPROM device, incremental changes can be made. Another difference is the EEPROM can be reprogrammed without a special programmer and can stay in the device while being reprogrammed. The versatility of byte-level programmability of the EEPROM comes at a price, however, as programming an EEPROM device is a slow process.
- § **Flash Memory**—the flash memory is a variation of EEPROM, which allows for block-level (e.g., 512-byte) programmability that is much faster than EEPROM.

Random Access Memory (RAM)

Also called Read/Write Memory, requires external power to maintain memory content. The term random access refers to the ability to access any memory cell directly. RAM is much faster than ROM. Two types of RAM that are of interest:

- § **Dynamic RAM (DRAM)**—DRAM is a RAM device that requires periodic refreshing to retain its content.
- § **Static RAM (SRAM)**—SRAM is a RAM device that retains its content as long as power is supplied by an external power source. SRAM does not require periodic refreshing and it is faster than DRAM.
- § **Non-Volatile RAM (NVRAM)**—NVRAM is a special type of SRAM that has backup battery power so it can retain its content after the main system power is shut off. Another variation of NVRAM combines SRAM and EEPROM so that its content is written into the EEPROM when power is shut off and is read back from the EEPROM when power is restored.

1.2 Real-Time Embedded Systems

In the simplest form, real-time systems can be defined as those systems that respond to external events in a timely fashion, as shown in [Figure 1.5](#). The response time is guaranteed. We revisit this definition after presenting some examples of real-time systems.

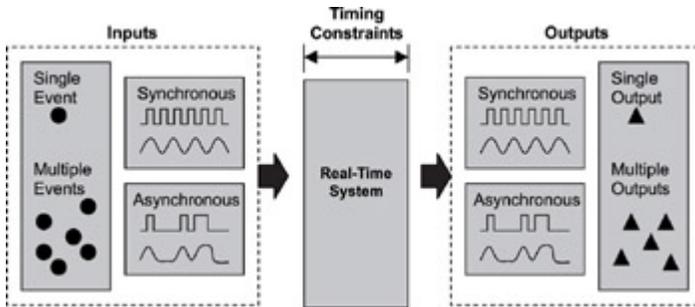


Figure 1.5: A simple view of real-time systems.

External events can have synchronous or asynchronous characteristics. Responding to external events includes recognizing when an event occurs, performing the required processing as a result of the event, and outputting the necessary results within a given time constraint. Timing constraints include finish time, or both start time and finish time.

A good way to understand the relationship between real-time systems and embedded systems is to view them as two intersecting circles, as shown in [Figure 1.6](#). It can be seen that not all embedded systems exhibit real-time behaviors nor are all real-time systems embedded. However, the two systems are not mutually exclusive, and the area in which they overlap creates the combination of systems known as *real-time embedded systems*.

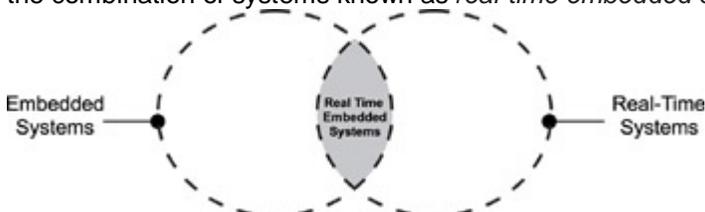


Figure 1.6: Real-time embedded systems.

Knowing this fact and because we have covered the various aspects of embedded systems in the previous sections, we can now focus our attention on real-time systems.

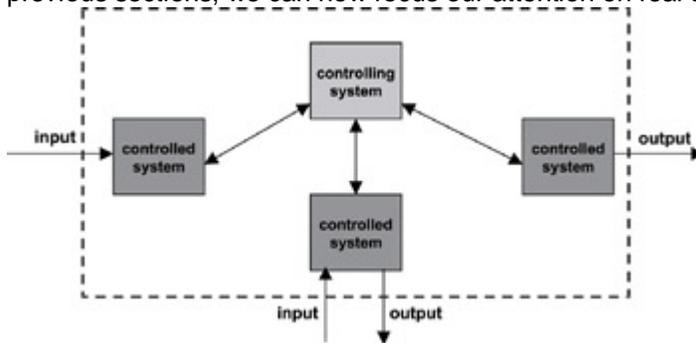


Figure 1.7: Structure of real-time systems.

1.2.1 Real-Time Systems

The environment of the real-time system creates the external events. These events are received by one or more components of the real-time system. The response of the real-time system is then injected into its environment through one or more of its components. Decomposition of the real-time system, as shown in [Figure 1.5](#), leads to the general structure of real-time systems.

The structure of a real-time system, as shown in [Figure 1.7](#), is a controlling system and at least one controlled system. The controlling system interacts with the controlled system in various ways. First, the interaction can be *periodic*, in which communication is initiated from the

controlling system to the controlled system. In this case, the communication is predictable and occurs at predefined intervals. Second, the interaction can be *aperiodic*, in which communication is initiated from the controlled system to the controlling system. In this case, the communication is unpredictable and is determined by the random occurrences of external events in the environment of the controlled system. Finally, the communication can be a combination of both types. The controlling system must process and respond to the events and information generated by the controlled system in a guaranteed time frame.

Imagine a real-time weapons defense system whose role is to protect a naval destroyer by shooting down incoming missiles. The idea is to shred an incoming missile into pieces with bullets before it reaches the ship. The weapons system is comprised of a radar system, a command-and-decision (C&D) system, and weapons firing control system. The controlling system is the C&D system, whereas the controlled systems are the radar system and the weapons firing control system.

§ The radar system scans and searches for potential targets. Coordinates of a potential target are sent to the C&D system periodically with high frequency after the target is acquired.

§ The C&D system must first determine the threat level by threat classification and evaluation, based on the target information provided by the radar system. If a threat is imminent, the C&D system must, at a minimum, calculate the speed and flight path or trajectory, as well as estimate the impact location. Because a missile tends to drift off its flight path with the degree of drift dependent on the precision of its guidance system, the C&D system calculates an area (a box) around the flight path.

§ The C&D system then activates the weapons firing control system closest to the anticipated impact location and guides the weapons system to fire continuously within the moving area or box until the target is destroyed. The weapons firing control system is comprised of large-caliber, multi-barrel, high-muzzle velocity, high-power machine guns.

In this weapons defense system example, the communication between the radar system and the C&D system is aperiodic, because the occurrence of a potential target is unpredictable and the potential target can appear at any time. The communication between the C&D system and the weapons firing control system is, however, periodic because the C&D system feeds the firing coordinates into the weapons control system periodically (with an extremely high frequency). Initial firing coordinates are based on a pre-computed flight path but are updated in real-time according to the actual location of the incoming missile.

Consider another example of a real-time system—the cruise missile guidance system. A cruise missile flies at subsonic speed. It can travel at about 10 meters above water, 30 meters above flat ground, and 100 meters above mountain terrains. A modern cruise missile can hit a target within a 50-meter range. All these capabilities are due to the high-precision, real-time guidance system built into the nose of a cruise missile. In a simplified view, the guidance system is comprised of the radar system (both forward-looking and look-down radars), the navigation system, and the divert-and-altitude-control system. The navigation system contains digital maps covering the missile flight path. The forward-looking radar scans and maps out the approaching terrains. This information is fed to the navigation system in real time. The navigation system must then recalculate flight coordinates to avoid terrain obstacles. The new coordinates are immediately fed to the divert-and-altitude-control system to adjust the flight path. The look-down radar periodically scans the ground terrain along its flight path. The scanned data is compared with the estimated section of the pre-recorded maps. Corrective adjustments are made to the flight coordinates and sent to the divert-and-altitude-control system if data comparison indicates that the missile has drifted off the intended flight path.

In this example, the controlling system is the navigation system. The controlled systems are the radar system and the divert-and-altitude-control system. We can observe both periodic and aperiodic communications in this example. The communication between the radars and the

navigation system is aperiodic. The communication between the navigation system and the diver-and-altitude-control system is periodic.

Let us consider one more example of a real-time system—a DVD player. The DVD player must decode both the video and the audio streams from the disc simultaneously. While a movie is being played, the viewer can activate the on-screen display using a remote control. On-screen display is a user menu that allows the user to change parameters, such as the audio output format and language options. The DVD player is the controlling system, and the remote control is the controlled system. In this case, the remote control is viewed as a sensor because it feeds events, such as pause and language selection, into the DVD player.

1.2.2 Characteristics of Real-Time Systems

The C&D system in the weapons defense system must calculate the anticipated flight path of the incoming missile quickly and guide the firing system to shoot the missile down before it reaches the destroyer. Assume T_1 is the time the missile takes to reach the ship and is a function of the missile's distance and velocity. Assume T_2 is the time the C&D system takes to activate the weapons firing control system and includes transmitting the firing coordinates plus the firing delay. The difference between T_1 and T_2 is how long the computation may take. The missile would reach its intended target if the C&D system took too long in computing the flight path. The missile would still reach its target if the computation produced by the C&D system was inaccurate. The navigation system in the cruise missile must respond to the changing terrain fast enough so that it can re-compute coordinates and guide the altitude control system to a new flight path. The missile might collide with a mountain if the navigation system cannot compute new flight coordinates fast enough, or if the new coordinates do not steer the missile out of the collision course.

Therefore, we can extract two essential characteristics of real-time systems from the examples given earlier. These characteristics are that real-time systems must produce correct computational results, called *logical or functional correctness*, and that these computations must conclude within a predefined period, called *timing correctness*.

Real-time systems are defined as those systems in which the overall correctness of the system depends on both the functional correctness and the timing correctness. The timing correctness is at least as important as the functional correctness.

It is important to note that we said the timing correctness is at least as important as the functional correctness. In some real-time systems, functional correctness is sometimes sacrificed for timing correctness. We address this point shortly after we introduce the classifications of real-time systems.

Similar to embedded systems, real-time systems also have substantial knowledge of the environment of the controlled system and the applications running on it. This reason is one why many real-time systems are said to be deterministic, because in those real-time systems, the response time to a detected event is bounded. The action (or actions) taken in response to an event is known a priori. A deterministic real-time system implies that each component of the system must have a deterministic behavior that contributes to the overall determinism of the system. As can be seen, a deterministic real-time system can be less adaptable to the changing environment. The lack of adaptability can result in a less robust system. The levels of determinism and of robustness must be balanced. The method of balancing between the two is system- and application-specific. This discussion, however, is beyond the scope of this book. Consult the reference material for additional coverage on this topic.

1.2.3 Hard and Soft Real-Time Systems

In the [previous section](#), we said computation must complete before reaching a given deadline. In other words, real-time systems have timing constraints and are deadline-driven. Real-time systems can be classified, therefore, as either hard real-time systems or soft real-time systems.

What differentiates hard real-time systems and soft real-time systems are the degree of tolerance of missed deadlines, usefulness of computed results after missed deadlines, and severity of the penalty incurred for failing to meet deadlines.

For hard real-time systems, the level of tolerance for a missed deadline is extremely small or zero tolerance. The computed results after the missed deadline are likely useless for many of these systems. The penalty incurred for a missed deadline is catastrophe. For soft real-time systems, however, the level of tolerance is non-zero. The computed results after the missed deadline have a rate of depreciation. The usefulness of the results does not reach zero immediately passing the deadline, as in the case of many hard real-time systems. The physical impact of a missed deadline is non-catastrophic.

A *hard real-time system* is a real-time system that must meet its deadlines with a near-zero degree of flexibility. The deadlines must be met, or catastrophes occur. The cost of such catastrophe is extremely high and can involve human lives. The computation results obtained after the deadline have either a zero-level of usefulness or have a high rate of depreciation as time moves further from the missed deadline before the system produces a response.

A *soft real-time system* is a real-time system that must meet its deadlines but with a degree of flexibility. The deadlines can contain varying levels of tolerance, average timing deadlines, and even statistical distribution of response times with different degrees of acceptability. In a soft real-time system, a missed deadline does not result in system failure, but costs can rise in proportion to the delay, depending on the application.

Penalty is an important aspect of hard real-time systems for several reasons.

§ What is meant by 'must meet the deadline'?

§ It means something catastrophic occurs if the deadline is not met. It is the penalty that sets the requirement.

§ Missing the deadline means a system failure, and no recovery is possible other than a reset, so the deadline must be met. Is this a hard real-time system?

That depends. If a system failure means the system must be reset but no cost is associated with the failure, the deadline is not a hard deadline, and the system is not a hard real-time system. On the other hand, if a cost is associated, either in human lives or financial penalty such as a \$50 million lawsuit, the deadline is a hard deadline, and it is a hard real-time system. It is the penalty that makes this determination.

§ What defines the deadline for a hard real-time system?

§ It is the penalty. For a hard real-time system, the deadline is a deterministic value, and, for a soft real-time system, the value can be estimation.

One thing worth noting is that the length of the deadline does not make a real-time system hard or soft, but it is the requirement for meeting it within that time.

The weapons defense and the missile guidance systems are hard real-time systems. Using the missile guidance system for an example, if the navigation system cannot compute the new coordinates in response to approaching mountain terrain before or at the deadline, not enough distance is left for the missile to change altitude. This system has zero tolerance for a missed deadline. The new coordinates obtained after the deadline are no longer useful because at subsonic speed the distance is too short for the altitude control system to navigate the missile into the new flight path in time. The penalty is a catastrophic event in which the missile collides with

the mountain. Similarly, the weapons defense system is also a zero-tolerance system. The missed deadline results in the missile sinking the destroyer, and human lives potentially being lost. Again, the penalty incurred is catastrophic.

On the other hand, the DVD player is a soft real-time system. The DVD player decodes the video and the audio streams while responding to user commands in real time. The user might send a series of commands to the DVD player rapidly causing the decoder to miss its deadline or deadlines. The result or penalty is momentary but visible video distortion or audible audio distortion. The DVD player has a high level of tolerance because it continues to function. The decoded data obtained after the deadline is still useful.

Timing correctness is critical to most hard real-time systems. Therefore, hard real-time systems make every effort possible in predicting if a pending deadline might be missed. Returning to the weapons defense system, let us discuss how a hard real-time system takes corrective actions when it anticipates a deadline might be missed. In the weapons defense system example, the C&D system calculates a firing box around the projected missile flight path. The missile must be destroyed a certain distance away from the ship or the shrapnel can still cause damage. If the C&D system anticipates a missed deadline (for example, if by the time the precise firing coordinates are computed, the missile would have flown past the safe zone), the C&D system must take corrective action immediately. The C&D system enlarges the firing box and computes imprecise firing coordinates by methods of estimation instead of computing for precise values. The C&D system then activates additional weapons firing systems to compensate for this imprecision. The result is that additional guns are brought online to cover the larger firing box. The idea is that it is better to waste bullets than sink a destroyer.

This example shows why sometimes functional correctness might be sacrificed for timing correctness for many real-time systems.

Because one or a few missed deadlines do not have a detrimental impact on the operations of soft real-time systems, a soft real-time system might not need to predict if a pending deadline might be missed. Instead, the soft real-time system can begin a recovery process after a missed deadline is detected.

For example, using the real-time DVD player, after a missed deadline is detected, the decoders in the DVD player use the computed results obtained after the deadline and use the data to make a decision on what future video frames and audio data must be discarded to re-synchronize the two streams. In other words, the decoders find ways to catch up.

So far, we have focused on meeting the deadline or the finish time of some work or job, e.g., a computation. At times, meeting the start time of the job is just as important. The lack of required resources for the job, such as CPU or memory, can prevent a job from starting and can lead to missing the job completion deadline. Ultimately this problem becomes a resource-scheduling problem. The scheduling algorithms of a real-time system must schedule system resources so that jobs created in response to both periodic and aperiodic events can obtain the resources at the appropriate time. This process affords each job the ability to meet its specific timing constraints. This topic is addressed in detail in [Chapter 14](#).

1.3 The Future of Embedded Systems

Until the early 1990s, embedded systems were generally simple, autonomous devices with long product lifecycles. In recent years, however, the embedded industry has experienced dramatic transformation, as reported by the Gartner Group, an independent research and advisory firm, as well as by other sources:

- § Product market windows now dictate feverish six- to nine-month turnaround cycles.
- § Globalization is redefining market opportunities and expanding application space.

- § Connectivity is now a requirement rather than a bonus in both wired and emerging wireless technologies.
- § Electronics-based products are more complex.
- § Interconnecting embedded systems are yielding new applications that are dependent on networking infrastructures.
- § The processing power of microprocessors is increasing at a rate predicted by Moore's Law, which states that the number of transistors per integrated circuit doubles every 18 months.

If past trends give any indication of the future, then as technology evolves, embedded software will continue to proliferate into new applications and lead to smarter classes of products. With an ever-expanding marketplace fortified by growing consumer demand for devices that can virtually run themselves as well as the seemingly limitless opportunities created by the Internet, embedded systems will continue to reshape the world for years to come.

1.4 Points to Remember

- § An embedded system is built for a specific application. As such, the hardware and software components are highly integrated, and the development model is the hardware and software co-design model.
- § Embedded systems are generally built using embedded processors.
- § An embedded processor is a specialized processor, such as a DSP, that is cheaper to design and produce, can have built-in integrated devices, is limited in functionality, produces low heat, consumes low power, and does not necessarily have the fastest clock speed but meets the requirements of the specific applications for which it is designed.
- § Real-time systems are characterized by the fact that timing correctness is just as important as functional or logical correctness.
- § The severity of the penalty incurred for not satisfying timing constraints differentiates hard real-time systems from soft real-time systems.
- § Real-time systems have a significant amount of application awareness similar to embedded systems.
- § Real-time embedded systems are those embedded system with real-time behaviors.

Chapter 2: Basics Of Developing For Embedded Systems

2.1 Introduction

[Chapter 1](#) states that one characteristic of embedded systems is the cross-platform development methodology. The primary components in the development environment are the host system, the target embedded system, and potentially many connectivity solutions available between the host and the target embedded system, as shown in [Figure 2.1](#).

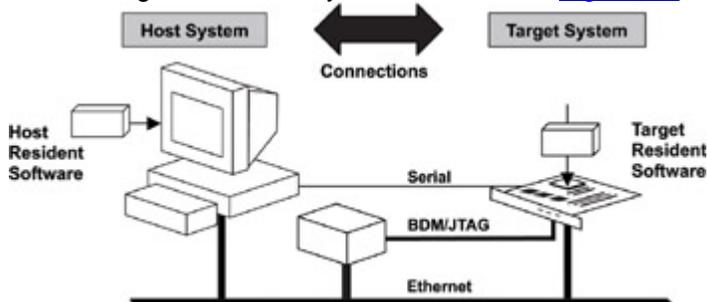


Figure 2.1: Typical cross-platform development environment.

The essential development tools offered by the host system are the cross compiler, linker, and source-level debugger. The target embedded system might offer a dynamic loader, a link loader, a monitor, and a debug agent. A set of connections might be available between the host and the target system. These connections are used for downloading program images from the host system to the target system. These connections can also be used for transmitting debugger information between the host debugger and the target debug agent.

Programs including the system software, the real-time operating system (RTOS), the kernel, and the application code must be developed first, compiled into object code, and linked together into an executable image. Programmers writing applications that execute in the same environment as used for development, called *native development*, do not need to be concerned with how an executable image is loaded into memory and how execution control is transferred to the application. Embedded developers doing cross-platform development, however, are required to understand the target system fully, how to store the program image on the target embedded system, how and where to load the program image during runtime, and how to develop and debug the system iteratively. Each of these aspects can impact how the code is developed, compiled, and most importantly linked.

The areas of focus in this chapter are

- § the ELF object file format,
- § the linker and linker command file, and
- § mapping the executable image onto the target embedded system.

This chapter does not provide full coverage on each tool, such as the compiler and the linker, nor does this chapter fully describe a specific object file format. Instead, this chapter focuses on providing in-depth coverage on the aspects of each tool and the object file format that are most relevant to embedded system development. The goal is to offer the embedded developer practical insights on how the components relate to one another. Knowing the big picture allows an embedded developer to put it all together and ask the specific questions if and when necessary.

2.2 Overview of Linkers and the Linking Process

Figure 2.2 illustrates how different tools take various input files and generate appropriate output files to ultimately be used in building an executable image.

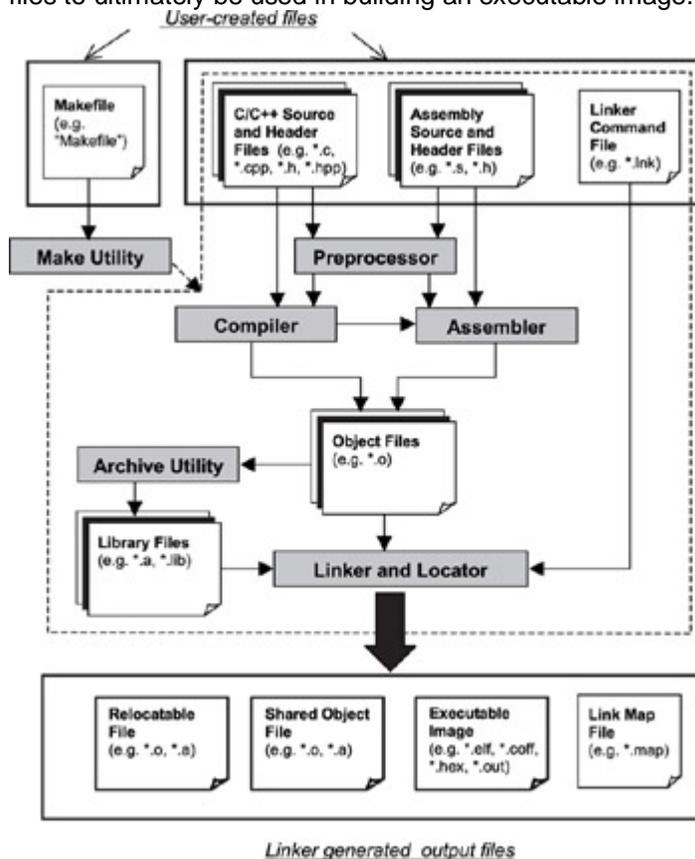


Figure 2.2: Creating an image file for the target system.

The developer writes the program in the C/C++ source files and header files. Some parts of the program can be written in assembly language and are produced in the corresponding assembly source files. The developer creates a `makefile` for the `make` utility to facilitate an environment that can easily track the file modifications and invoke the compiler and the assembler to rebuild the source files when necessary. From these source files, the compiler and the assembler produce object files that contain both machine binary code and program data. The archive utility concatenates a collection of object files to form a library. The linker takes these object files as input and produces either an executable image or an object file that can be used for additional linking with other object files. The linker command file instructs the linker on how to combine the object files and where to place the binary code and data in the target embedded system.

The main function of the linker is to combine multiple object files into a larger relocatable object file, a shared object file, or a final executable image. In a typical program, a section of code in one source file can reference variables defined in another source file. A function in one source file can call a function in another source file. The global variables and non-static functions are commonly referred to as *global symbols*. In source files, these symbols have various names, for example, a global variable called `foo_bar` or a global function called `func_a`. In the final executable binary image, a symbol refers to an address location in memory. The content of this memory location is either data for variables or executable code for functions.

The compiler creates a symbol table containing the symbol name to address mappings as part of the object file it produces. When creating relocatable output, the compiler generates the address

that, for each symbol, is relative to the file being compiled. Consequently, these addresses are generated with respect to offset 0. The symbol table contains the global symbols defined in the file being compiled, as well as the external symbols referenced in the file that the linker needs to resolve. The linking process performed by the linker involves symbol resolution and symbol relocation.

Symbol resolution is the process in which the linker goes through each object file and determines, for the object file, in which (other) object file or files the external symbols are defined. Sometimes the linker must process the list of object files multiple times while trying to resolve all of the external symbols. When external symbols are defined in a static library, the linker copies the object files from the library and writes them into the final image.

Symbol relocation is the process in which the linker maps a symbol reference to its definition. The linker modifies the machine code of the linked object files so that code references to the symbols reflect the actual addresses assigned to these symbols. For many symbols, the relative offsets change after multiple object files are merged. Symbol relocation requires code modification because the linker adjusts the machine code referencing these symbols to reflect their finalized addresses. The relocation table tells the linker where in the program code to apply the relocation action. Each entry in the relocation table contains a reference to the symbol table. Using this reference, the linker can retrieve the actual address of the symbol and apply it to the program location as specified by the relocation entry. It is possible for the relocation table to contain both the address of the symbol and the information on the relocation entry. In this case, there is no reference between the relocation table and the symbol table.

[Figure 2.3](#) illustrates these two concepts in a simplified view and serves as an example for the following discussions.

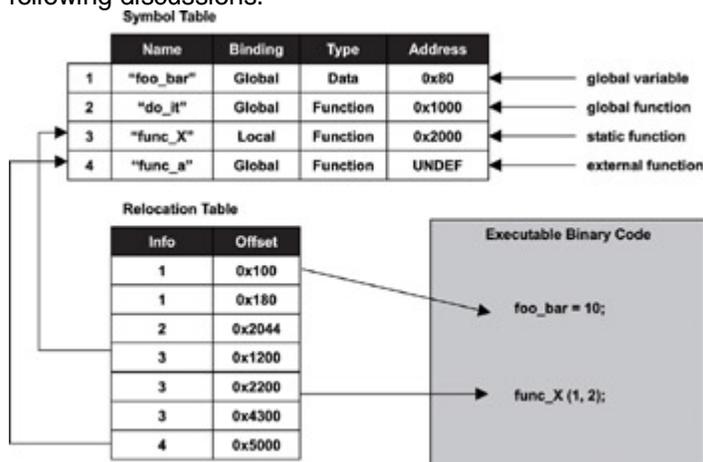


Figure 2.3: Relationship between the symbol table and the relocation table.

For an executable image, all external symbols must be resolved so that each symbol has an absolute memory address because an executable image is ready for execution. The exception to this rule is that those symbols defined in shared libraries may still contain relative addresses, which are resolved at runtime (dynamic linking).

A relocatable object file may contain unresolved external symbols. Similar to a library, a linker-reproduced relocatable object file is a concatenation of multiple object files with one main difference—the file is partially resolved and is used for further linking with other object files to create an executable image or a shared object file. A shared object file has dual purposes. It can be used to link with other shared object files or relocatable object modules, or it can be used as an executable image with dynamic linking.

2.3 Executable and Linking Format

Typically an object file contains

- § general information about the object file, such as file size, binary code and data size, and source file name from which it was created,
- § machine-architecture-specific binary instructions and data
- § symbol table and the symbol relocation table, and
- § debug information, which the debugger uses.

The manner in which this information is organized in the object file is the *object file format*. The idea behind a standard object file format is to allow development tools which might be produced by different vendors—such as a compiler, assembler, linker, and debugger that conform to the well-defined standard—to interoperate with each other.

This interoperability means a developer can choose a compiler from vendor A to produce object code used to form a final executable image by a linker from vendor B. This concept gives the end developer great flexibility in choice for development tools because the developer can select a tool based on its functional strength rather than its vendor.

Two common object file formats are the common object file format (COFF) and the executable and linking format (ELF). These file formats are incompatible with each other; therefore, be sure to select the tools, including the debugger, that recognize the format chosen for development.

We focus our discussion on ELF because it supersedes COFF. Understanding the object file format allows the embedded developer to map an executable image into the target embedded system for static storage, as well as for runtime loading and execution. To do so, we need to discuss the specifics of ELF, as well as how it relates to the linker.

Using the ELF object file format, the compiler organizes the compiled program into various system-defined, as well as user-defined, content groupings called *sections*. The program's binary instructions, binary data, symbol table, relocation table, and debug information are organized and contained in various sections. Each section has a type. Content is placed into a section if the section type matches the type of the content being stored.

A section also contains important information such as the load address and the run address. The concept of load address versus run address is important because the run address and the load address can be different in embedded systems. This knowledge can also be helpful in understanding embedded system loader and link loader concepts introduced in [Chapter 3](#).

[Chapter 1](#) discusses the idea that embedded systems typically have some form of ROM for non-volatile storage and that the software for an embedded system can be stored in ROM. Modifiable data must reside in RAM. Programs that require fast execution speed also execute out of RAM. Commonly therefore, a small program in ROM, called a *loader*, copies the initialized variables into RAM, transfers the program code into RAM, and begins program execution out of RAM. This physical ROM storage address is referred to as the section's *load address*. The section's *run address* refers to the location where the section is at the time of execution. For example, if a section is copied into RAM for execution, the section's run address refers to an address in RAM, which is the destination address of the loader copy operation. The linker uses the program's run address for symbol resolutions.

The ELF file format has two different interpretations, as shown in [Figure 2.4](#). The linker interprets the file as a linkable module described by the section header table, while the loader interprets the file as an executable module described by the program header table.

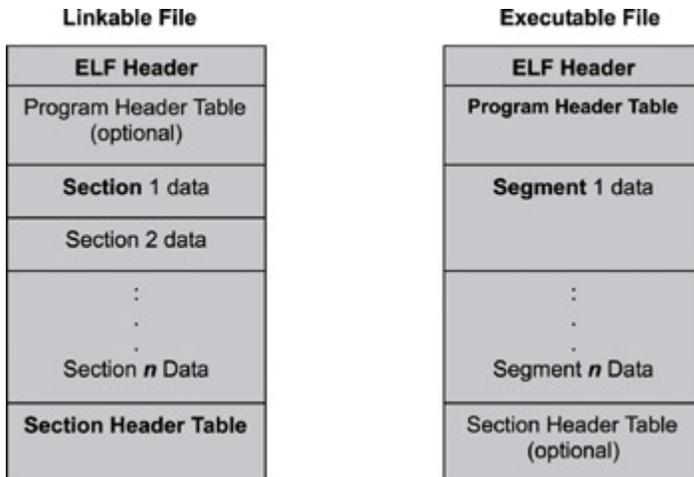


Figure 2.4: Executable and linking format.

Listing 2.1 shows both the section header and the program header, as represented in C programming structures. We describe the relevant fields during the course of this discussion.

Listing 2.1: Section header and program header.

Section header	Program header
<pre>typedef struct { Elf32_Word sh_name; Elf32_Word sh_type; Elf32_Word sh_flags; Elf32_Addr sh_addr; Elf32_Off sh_offset; Elf32_Word sh_size; Elf32_Word sh_link; Elf32_Word sh_info; Elf32_Word sh_addralign; Elf32_Word sh_entsize; } Elf32_Shdr;</pre>	<pre>typedef struct { Elf32_Word p_type; Elf32_Off p_offset; Elf32_Addr p_vaddr; Elf32_Addr p_paddr; Elf32_Word p_filesz; Elf32_Word p_memsz; Elf32_Word p_flags; Elf32_Word p_align; } Elf32_Phdr;</pre>

A *section header table* is an array of section header structures describing the sections of an object file. A *program header table* is an array of program header structures describing a loadable segment of an image that allows the loader to prepare the image for execution. Program headers are applied only to executable images and shared object files.

One of the fields in the section header structure is `sh_type`, which specifies the type of a section. Table 2.1 lists some section types.

NULL	Inactive header without a section.
PROGBITS	Code or initialized data.

SYMTAB	Symbol table for static linking.
STRTAB	String table.
RELA/REL	Relocation entries.
HASH	Run-time symbol hash table.
DYNAMIC	Information used for dynamic linking.
NOBITS	Uninitialized data.
DYNSYM	Symbol table for dynamic linking.

The `sh_flags` field in the section header specifies the attribute of a section. [Table 2.2](#) lists some of these attributes.

Table 2.2: Section attributes.

WRITE	Section contains writeable data.
ALLOC	Section contains allocated data.
EXECINSTR	Section contains executable instructions.

Some common system-created default sections with predefined names for the `PROGBITS` are `.text`, `.sdata`, `.data`, `.sbss`, and `.bss`. Program code and constant data are contained in the `.text` section. This section is read-only because code and constant data are not expected to change during the lifetime of the program execution. The `.sbss` and `.bss` sections contain uninitialized data. The `.sbss` section stores *small data*, which is the data such as variables with sizes that fit into a specific size. This size limit is architecture-dependent. The result is that the compiler and the assembler can generate smaller and more efficient code to access these data items. The `.sdata` and `.data` sections contain initialized data items. The small data concept described for `.sbss` applies to `.sdata`. A `.text` section with executable code has the `EXECINSTR` attribute. The `.sdata` and `.data` sections have the `WRITE` attribute. The `.sbss` and `.bss` sections have both the `WRITE` and the `ALLOC` attributes.

Other common system-defined sections are `.symtab` containing the symbol table, `.strtab` containing the string table for the program symbols, `.shstrtab` containing the string table for the section names, and `.relaname` containing the relocation information for the section named *name*. We have discussed the role of the symbol table (`SYMTAB`) previously. In [Figure 2.3](#), the symbol name is shown as part of the symbol table. In practice, each entry in the symbol table contains a reference to the string table (`STRTAB`) where the character representation of the name is stored.

The developer can define custom sections by invoking the linker command `.section`. For example, where the source files states

```
.section my_section
```

the linker creates a new section called `my_section`. The reasons for creating custom named sections are explained shortly.

The `sh_addr` is the address where the program section should reside in the target memory. The `p_paddr` is the address where the program segment should reside in the target memory. The `sh_addr` and the `p_paddr` fields refer to the load addresses. The loader uses the load address field from the section header as the starting address for the image transfer from non-volatile memory to RAM.

For many embedded applications, the run address is the same as the load address. These embedded applications are directly downloaded into the target system memory for immediate execution without the need for any code or data transfer from one memory type or location to another. This practice is common during the development phase. We revisit this topic in [Chapter 3](#), which covers the topic of image transfer from the host system to the target system.

2.4 Mapping Executable Images into Target Embedded Systems

After multiple source files (C/C++ and assembly files) have been compiled and assembled into ELF object files, the linker must combine these object files and merge the sections from the different object files into program segments. This process creates a single executable image for the target embedded system. The embedded developer uses linker commands (called *linker directives*) to control how the linker combines the sections and allocates the segments into the target system. The linker directives are kept in the *linker command file*. The ultimate goal of creating a linker command file is for the embedded developer to map the executable image into the target system accurately and efficiently.

2.4.1 Linker Command File

The format of the linker command file, as well as the linker directives, vary from linker to linker. It is best to consult the programmer's reference manual from the vendor for specific linker commands, syntaxes, and extensions. Some common directives, however, are found among the majority of the available linkers used for building embedded applications. Two of the more common directives supported by most linkers are `MEMORY` and `SECTION`.

The `MEMORY` directive can be used to describe the target system's *memory map*. The memory map lists the different types of memory (such as RAM, ROM, and flash) that are present on the target system, along with the ranges of addresses that can be accessed for storing and running an executable image. An embedded developer needs to be familiar with the addressable physical memory on a target system before creating a linker command file. One of the best ways to do this process, other than having direct access to the hardware engineering team that built the target system, is to look at the target system's *schematics*, as shown in [Figure 2.5](#), and the hardware documentation. Typically, the hardware documentation describes the target system's memory map.

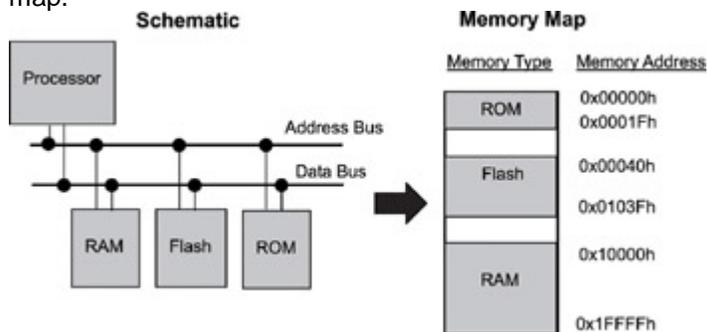


Figure 2.5: Simplified schematic and memory map for a target system.

The linker combines input sections having the same name into a single output section with that name by default. The developer-created, custom-named sections appear in the object file as independent sections. Sometimes developers might want to change this default linker behavior of only coalescing sections with the same name. The embedded developer might also need to instruct the linker on where to map the sections, in other words, what addresses should the linker use when performing symbol resolutions. The embedded developer can use the `SECTION` directive to achieve these goals.

The `MEMORY` directive defines the types of physical memory present on the target system and the address range occupied by each physical memory block, as specified in the following generalized syntax

```
MEMORY {  
    area-name : org = start-address, len = number-of-bytes  
    ...  
}
```

In the example shown in [Figure 2.5](#), three physical blocks of memory are present:

- § a ROM chip mapped to address space location 0, with 32 bytes,
- § some flash memory mapped to address space location 0x40, with 4,096 bytes, and
- § a block of RAM that starts at origin 0x10000, with 65,536 bytes.

Translating this memory map into the `MEMORY` directive is shown in [Listing 2.2](#). The named areas are ROM, FLASH, and RAM.

Listing 2.2: Memory map.

```
MEMORY {  
    ROM: origin = 0x0000h, length = 0x0020h  
    FLASH: origin = 0x0040h, length = 0x1000h  
    RAM: origin = 0x1000h, length = 0x10000h  
}
```

The `SECTION` directive tells the linker which input sections are to be combined into which output section, which output sections are to be grouped together and allocated in contiguous memory, and where to place each section, as well as other information. A general notation of the `SECTION` command is shown in [Listing 2.3](#).

Listing 2.3: SECTION command.

```
SECTION {  
    output-section-name : { contents } > area-name  
    ...  
    GROUP {  
        [ALIGN(expression)]  
        section-definition  
        ...  
    } > area-name  
}
```

The example shown in [Figure 2.6](#) contains three default sections (`.text`, `.data`, and `.bss`), as well as two developer-specified sections (`loader` and `my_section`), contained in two object files generated by a compiler or assembler (`file1.o` and `file2.o`). Translating this example into the `MEMORY` directive is shown in [Listing 2.4](#).

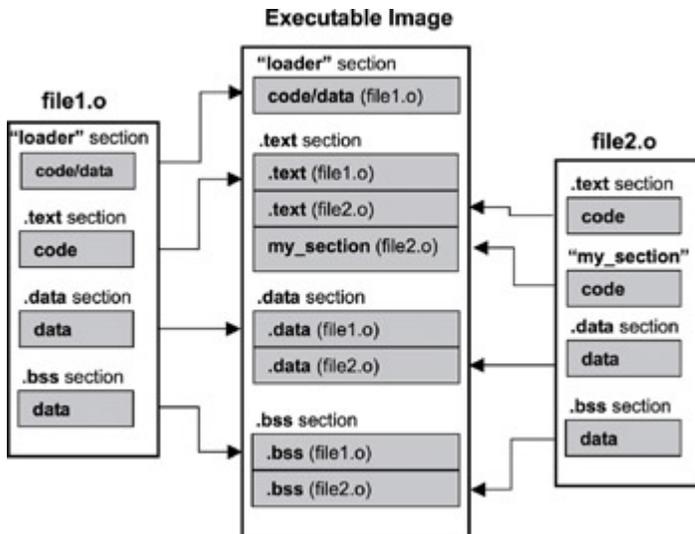


Figure 2.6: Combining input sections into an executable image.

Listing 2.4: Example code.

```
SECTION {
    .text :
    {
        my_section
        *(.text)
    }
    loader : > FLASH
    GROUP ALIGN (4) :
    {
        .text,
        .data : {}
        .bss : {}
    } >RAM
}
```

The `SECTION` command in the linker command file instructs the linker to combine the input section named `my_section` and the default `.text` sections from all object files into the final output `.text` section. The loader section is placed into flash memory. The sections `.text`, `.data`, and `.bss` are grouped together and allocated in contiguous physical RAM memory aligned on the 4-byte boundary, as shown in [Figure 2.7](#).

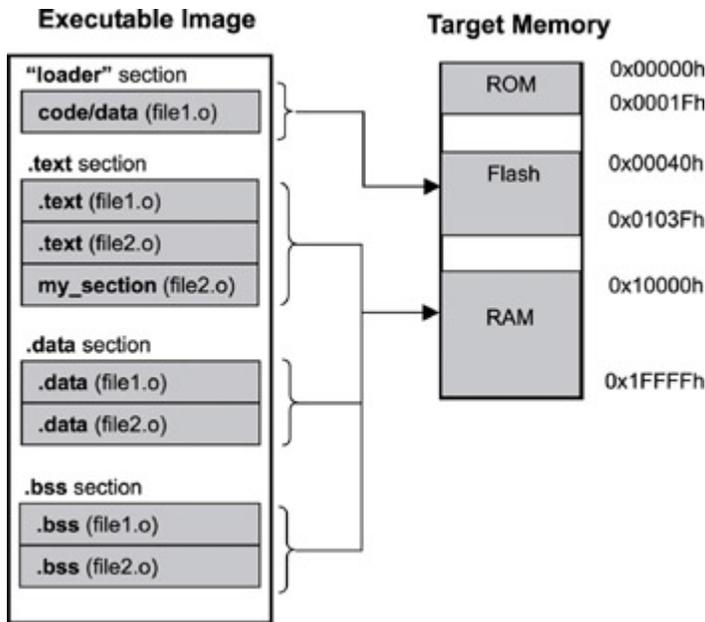


Figure 2.7: Mapping an executable image into the target system.

Tips on section allocation include the following:

- § allocate sections according to size to fully use available memory, and
- § examine the nature of the underlying physical memory, the attributes, and the purpose of a section to determine which physical memory is best suited for allocation.

2.4.2 Mapping Executable Images

Various reasons exist why an embedded developer might want to define custom sections, as well as to map these sections into different target memory areas as shown in the last example. The following sections list some of these reasons.

Module Upgradeability

[Chapter 1](#) discusses the storage options and upgradability of software on embedded systems. Software can be easily upgraded when stored in non-volatile memory devices, such as flash devices. It is possible to upgrade the software dynamically while the system is still running. Upgrading the software can involve downloading the new program image over either a serial line or a network and then re-programming the flash memory. The loader in the example could be such an application. The initial version of the loader might be capable of transferring an image from ROM to RAM. A newer version of the loader might be capable of transferring an image from the host over the serial connection to RAM. Therefore, the loader code and data section would be created in a custom loader section. The entire section then would be programmed into the flash memory for easy upgradeability in the future.

Memory Size Limitation

The target system usually has different types of physical memory, but each is limited in size. At times, it is impossible to fit all of the code and data into one type of memory, for example, the SDRAM. Because SDRAM has faster access time than DRAM, it is always desirable to map code and data into it. The available physical SDRAM might not be large enough to fit everything, but plenty of DRAM is available in the system. Therefore, the strategy is to divide the program into multiple sections and have some sections allocated into the SDRAM, while the rest is mapped

into the DRAM. For example, an often-used function along with a frequently searched lookup table might be mapped to the SDRAM. The remaining code and data is allocated into the DRAM.

Data Protection

Programs usually have various types of constants, such as integer constants and string constants. Sometimes these constants are kept in ROM to avoid accidental modification. In this case, these constants are part of a special data section, which is allocated into ROM.

2.4.3 Example in Practice

Consider an example system containing 256 bytes of ROM, 16KB of flash memory, and two blocks of RAM. RAMB0 is 128KB of SDRAM, and RAMB1 is 2MB of DRAM. An embedded application with a number of sections, as listed in [Table 2.3](#), needs to be mapped into this target system.

Sections	Size	Attribute¹	Description
_loader	10KB	RD	Contains the loader code
_wflash	2KB	RD	Contains the flash memory programmer
.rodata	128 bytes	RD	Contains non-volatile default initialization parameters and data, such as copyright information
.sbss	10KB	R/W	Contains uninitialized data less than 64KB (e.g., global variables)
.sdata	2KB	R/W	Contains initialized data less than 64KB
.bss	128KB	R/W	Contains uninitialized data larger than 64KB
.data	512KB	R/W	Contains initialized data larger than 64KB
_monitor	54KB	RD	Contains the monitor code
.text	512KB	RD	Contains other program code

1. RD = read only; R/W = readable and writeable

One possible allocation is shown in [Listing 2.5](#); it considers why an embedded engineer might want greater section allocation control.

Listing 2.5: Possible section allocation.

```
MEMORY {
    ROM: origin = 0x00000h, length = 0x000100h
    FLASH: origin = 0x00110h, length = 0x004000h
    RAMB0: origin = 0x05000h, length = 0x020000h
    RAMB1: origin = 0x25000h, length = 0x200000h
}
```

```

SECTION {
    .rodata : > ROM
    _loader : > FLASH
    _wflash : > FLASH
    _monitor : > RAMB0
    .sbss (ALIGN 4) : > RAMB0
    .sdata (ALIGN 4) : > RAMB0
    .text : > RAMB1
    .bss (ALIGN 4) : > RAMB1
    .data (ALIGN 4) : > RAMB1
}

```

This program allocation is shown in [Figure 2.8 \(page 34\)](#). The section allocation strategies applied include the following:

- § The `.rodata` section contains system initialization parameters. Most likely these default values never change; therefore, allocate this section to ROM.
- § The loader program is usually part of the system program that executes at startup. The `_loader` and the `_wflash` sections are allocated into flash memory because the loader code can be updated with new versions that understand more object formats. You need the flash memory programmer for this purpose, which can also be updated. Therefore, section `_wflash` is allocated into the flash memory as well.
- § The embedded programmer interacts with the monitor program to probe system execution states and help debug application code; therefore, it should be responsive to user commands. SDRAM is faster than DRAM, with shorter access time. Therefore, section `_monitor` is allocated into RAMB0.
- § RAMB0 still has space left to accommodate both sections `.sbss` and `.sdata`. The allocation strategy for these two sections is to use the leftover fast memory fully.
- § The remaining sections (`.text`, `.bss`, and `.data`) are allocated into RAMB1, which is the only memory that can accommodate all of these large sections.

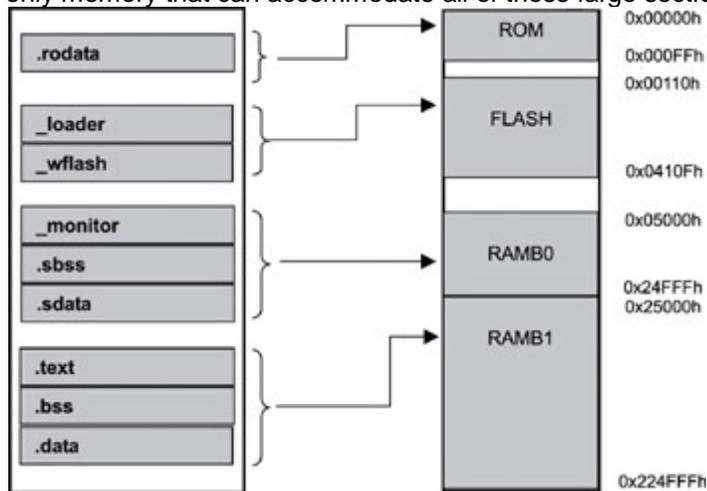


Figure 2.8: Mapping an executable image into the target system.

2.5 Points to Remember

Some points to remember include the following:

- § The linker performs symbol resolution and symbol relocation.
- § An embedded programmer must understand the exact memory layout of the target system towards which development is aimed.
- § An executable target image is comprised of multiple program sections.
- § The programmer can describe the physical memory, such as its size and its mapping address, to the linker using the linker command file. The programmer can also instruct the linker on combining input sections into output sections and placing the output program sections using the linker command file.
- § Each program section can reside in different types of physical memory, based on how the section is used. Program code (or `.text` section) can stay in ROM, flash, and RAM during execution. Program data (or `.data` section) must stay in RAM during execution.