REAL-TIME SOFTWARE

Robert L. Glass

73.8722 /
G549

REAL-TIME SOFTWARE

ROBERT L. GLASS

ASSISTANT PROFESSOR
SOFTWARE ENGINEERING PROGRAM
SEATTLE UNIVERSITY

5506759

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data
Main entry under title:

Real-time software.

Collection of papers on computer software.
Bibliography: p. 0000
Includes index.
1. Real-time data processing. 2. Electronic digital computers--Programming I. Glass, Robert L., 1932-
QA76.54.R425 1984 001.64'42 83-3156
ISBN 0-13-767103-2

Editorial/production supervision
and interior design: Karen Skrable
Manufacturing buyer: Anthony Caruso
Cover design: Edsal Enterprises

D287/13

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-767103-2

PRENTICE-HALL INTERNATIONAL, INC., LONDON
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, SYDNEY
EDITORA PRENTICE-HALL DO BRASIL, LTDA., RIO DE JANEIRO
PRENTICE-HALL CANADA INC., TORONTO
PRENTICE-HALL OF INDIA PRIVATE LIMITED, NEW DELHI
PRENTICE-HALL OF JAPAN, INC., TOKYO
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., SINGAPORE
WHITEHALL BOOKS LIMITED, WELLINGTON, NEW ZEALAND

PREFACE AND SUMMARY

Let me take you into the Software Development Laboratory of a major real-time project.

Over there, in the dominant position in the room, is the "target" minicomputer. It is the computer that will eventually control the real-time system for which the software is being built. (The software, of course, will control the computer that controls the system.) The computer is called a target because it does not contain the facilities—the compilers, assemblers, and linkers—that generate its code. A "host" computer, which is located in another part of the building and is a mainframe, does that job.

Sitting in front of the target computer console are a couple of programmers. They look perplexed. The computer's lights are not blinking; it has apparently stopped, and the programmers are attempting to determine why. From their conversation, we can tell that one programmer is trying to debug an application program; the other programmer evidently wrote the executive program that is being used by the application program, and there seems to be a difference of opinion as to who is to blame for the halt.

All around the programmers, occupying the rest of the Software Development Laboratory, are pieces of equipment. The equipment is a replication of devices that will actually talk to the target computer when the completed system is working. Cables connect the equipment to the target.

The Laboratory is not pretty. Instead, it looks . . . well . . . functional. Only when the Laboratory is transformed, after checkout, into an operational system will prettiness become a goal.

This is the world of the real-time software engineer. *Engineer* is the key word here, where it implies a certain amount of pragmatism and a dogged determination to make things work.

The programmers at the console have long since removed their coats, and one wearing a tie has loosened it. A stale cup of coffee sits on a table near the console, and an ashtray full of half-smoked cigarettes is next to it.

Here, in the Software Development Lab, is the grittiness—and the joy—of creating software for a real-time system.

The programmers are beginning to raise their voices. Apparently the fault is a complex one, and it really isn't obvious whose code went wrong. Let's tiptoe out.

The scene we have just viewed is characteristic of an environment that is becoming more and more pervasive. In this book we will see how real-time systems and real-time software are becoming commonplace. Although they service an amazing diversity of applications, the development process and the overall structure of real-time systems are surprisingly similar. The Lab is one of those similarities, as are the engineering emphasis and the team approach of the programmers. They will resolve their problem in the next ten minutes and move on to another.

The purpose of this book is to explore the world of the real-time programmer. We may not see things quite as graphically as stepping into that Laboratory—but there are many intangibles in the real-time software world, and most of us find them at least as interesting as viewing and touching the Laboratory hardware.

This book has five main chapters. Chapter 1 gives us an overview of real-time software. We take a look at some examples of real-time systems, where the problems lie in these systems, and what real-time software is all about.

In chapter 2 we move on to applications of real-time software. This chapter illustrates just how pervasive the world of real-time software actually is and explores some of the techniques of creating that software.

Chapter 3 is the real "how-to" chapter. There we dissect the process of software construction into some constituent phases and analyze how to perform each one. We look at a mix of practice and theory, perhaps best symbolized by a paper on the generation of a requirements specification that details the application of still fluid theory to an actual real-time project. We will despair at some of the obsolete practices in the field—there is a paper on "patching," for example—and yet cheer for the amazing success of real-timers who get these complex systems to work. They do so with a reliability track record that looks better for software, says another paper, than for other disciplines.

In chapter 4 we deal with some specifics. There is a collection of tools that a real-timer needs, and chapter 4 discusses not only what those tools are but what they should be like. The executive is one of them, and the language processor is another. A hard look is taken at some key aspects of those and other tools.

Finally, in chapter 5, we talk about languages for real-time software. Here the book places a heavy emphasis on pragmatism; some theoretic language advances are discussed, but the bulk of the material deals with the background and use of the more (potentially) mainstream languages, including a paper on Ada.

Before we move on into the material we have introduced, we must perform a few housekeeping functions. Think of this as the initialization routine for your real-time software system—it may not be very interesting, but nothing that follows will work well without it!

First of all, this book is a collection of papers. The papers were written by several real-time software experts. The papers were chosen to cover what your editor believes are the most important areas of real-time software and have been grouped into a readable structure.

Second, although these papers are a mix of tutorial and experiential works, the emphasis here is on the experienced real-time software practitioner and what will be useful to him or her. Certainly if you have had no previous exposure to software at all, and perhaps if you have had no previous exposure to real-time software, a tutorial text such as *Introduction to Real-Time Software Design,* written by S. T. Allworth (1981), might be a better starting place. Topics such as the relative efficiency of compiler- and assembler-generated codes, which are covered later in this book, may have limited value to the novice but are of critical value to the veteran.

Finally, no technical book is complete without a definition of its subject-matter terminology. For this book that means defining the term *real-time software.* This can be a surprisingly difficult task—some authors announce the decision to provide a definition, then cleverly write several pages but never provide one! However, an initialization routine cannot vacillate. Here we go:

> *Real-time software is that software that controls a computer that controls a real-time system. A real-time system is one that provides services or control to an on-going physical process.*

That done, let us proceed to the material at hand.

ACKNOWLEDGMENT AND DEDICATION

Real-time software is a "doing" more than a "hypothesizing" kind of business. Because of that, and because of the well-known reluctance of "doers" to write about what they have done, I owe a deep debt of gratitude to the authors of the papers that form the substance of this book. Those papers—and this book—are the creation of doers who cared enough to share what they have done.

This book is dedicated to all the "doers" of real-time software.

ROBERT L. GLASS

CONTENTS

1 AN OVERVIEW OF REAL-TIME SOFTWARE

The digital computer is becoming ever more present in the daily lives of all of us. Computers allow our watches to play games as well as tell time, optimize the gas mileage of our latest-generation cars, and sequence our appliances. Soon we may even have computers controlling the heating and lighting patterns in our homes.

All these computing interactions—be they helpful or intrusive—are examples of real-time computing. The computer is controlling something which interacts with reality on a timeable basis. In fact, timing is the essence of the interaction. Our game-playing wristwatch must respond within a few seconds or we will shake it to see if it is still working. Our car's optimized carburetor had better snap to instant attention when we depress the gas pedal. An unresponsive-real-time system may be worse than no system at all.

The computer in a real-time system may not be too different from the computer in a nonreal-time system. What is *really* "real" in a real-time system is the software. Here, in the intangible recesses of the computer's memory, is where the input is ingested, the decisions made, and the output sent—all in whatever quick-as-a-wink timing the real-time system requires.

Efficiency is always a goal in computer software, but in real-time software efficiency reigns supreme. Real-time systems often use marginal computer hardware—a little small, a little slow—to cut the cost of the total system. In those circumstances the real-time software person has to shoehorn too much software into the too-small machine in such a way that the system interacts with its external reality acceptably fast.

However, it is not simply efficiency that differentiates real-time software from other software. The papers which follow give an in-depth analysis of what characterizes this subject.

These papers take the form of a James Martin sandwich—two papers by Martin describe (1) what kinds of systems make up the spectrum of real-time systems, and (2) what difficulties characterize real-time software. In between, as a solid, meaty filling for the sandwich, is a survey paper on process control (that is, real-time) software by Janos Gertler and Jan Sedlak.

The Martin papers are taken from his book *Programming Real-Time*

Systems. In 1965 this book was *the* authoritative work on real-time software. In a field where most information becomes obsolete in well under ten years, what is amazing is that Martin's material is still clearly written, accurate, and insightful by today's standards. Only his example applications begin to feel a little dated.

In contrast with Martin's light writing style is the extremely thorough work of Gertler and Sedlak. Dealing on a level ranging from methodology to standardization, these two Europeans (Gertler is from Hungary and Sedlak from Czechoslovakia) have carefully sculpted a master overview of a difficult subject.

6. *The complexity of the Supervisory Programs.* This is the Control, or Executive, Program which schedules the work, organizes input and output operations, and so on. On small systems it can be a fairly simple program and may involve only a small addition to the standard program packages provided by the computer manufacturers. On large and complex systems it can be a very sophisticated and intricate group of programs. . . . Its complexity is determined by:
   (a) The complexity of the equipment.
   (b) The degree of multiprogramming, that is, the simultaneous processing of transactions.
   (c) The complexity of the priority structure. On some systems all transactions have the same priority, but on others there are differences in priorities between different messages or different functions.

1-1-1 THE COMPLEXITY OF THE EQUIPMENT

The simplest form of a real-time system might have one device, such as a typewriter, which can send a message to the computer. The computer interrupts its processing, handles the message, perhaps sends a reply, and then continues its processing. The computer may have a random-access file attached, and the typewriter may update or interrogate this. Slightly more complicated would be a system with several terminals or typewriters. These may be attached to a buffer or they may be all on one communication line (Figure 1-1-1). Only one

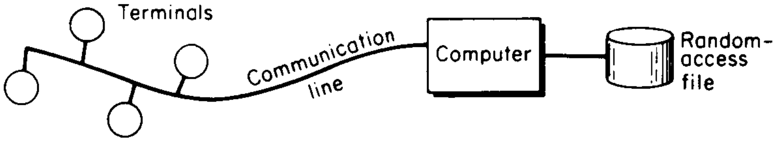


Figure 1-1-1 A system with one communication line.

terminal can send a message at one time. The next step up in complexity would be to have more than one communications line (Figure 1-1-2).

With two or more lines, message handling may or may not overlap in the computer. This depends upon the size and throughput of the system. Systems with a high throughput will process messages in parallel.

With some makes of equipment the communication line is able to go directly into the computer. The computer assembles the bits and characters from the line and compiles the messages under program control. With other systems the communication lines do not go into the computer but into a separate programmed *Multiplexor* or Line Control Computer. This is in effect a small special-purpose computer for controlling communication line input and

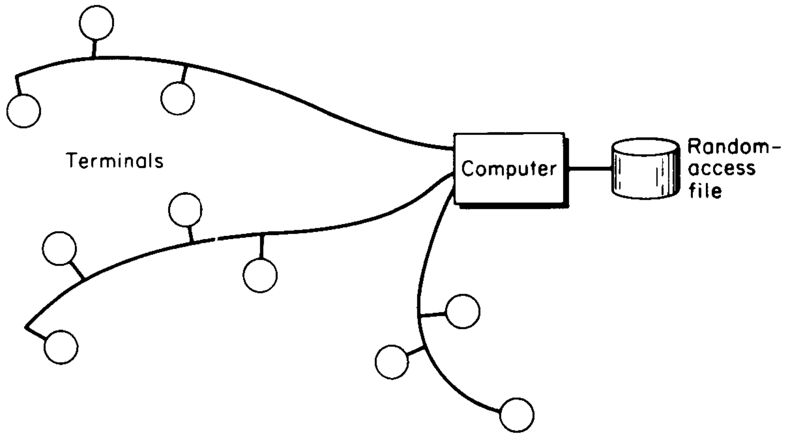


Figure 1-1-2 A system with several communication lines.

output. To it can be attached any general-purpose computer which reads data from it and sends data to it in the same way that it would to any input/output unit (Figure 1-1-3).

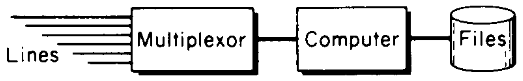


Figure 1-1-3 A system with a separate line control computer.

In some systems more than one computer has been used because one computer is not big enough or fast enough. In a two-computer system, one may handle the input and output to the Line Control device and files, while the other does the processing. A good reason for this could be that the processing computer is doing nonreal-time work, but is interrupted occasionally by the other which has assembled some real-time messages ready for processing (Figure 1-1-4).

Computer B in Figure 1-1-4 may be a much more powerful machine than A. Computer A may handle some simple real-time transactions itself, such as requests for interrogation of the files. When a transaction requires more complex processing it interrupts computer B. This prepares a reply for A to send and then continues its other work. Computer A may queue the messages so that it does not interrupt computer B very often. Computer B may have a program store on a disk file or drum so that it can load itself with the necessary real-time programs.

Because of the nature of the real-time work a very high degree of reliability may be needed in the system. This may be achieved by duplicating the components of the system. If one computer has a breakdown time of .2 percent, two

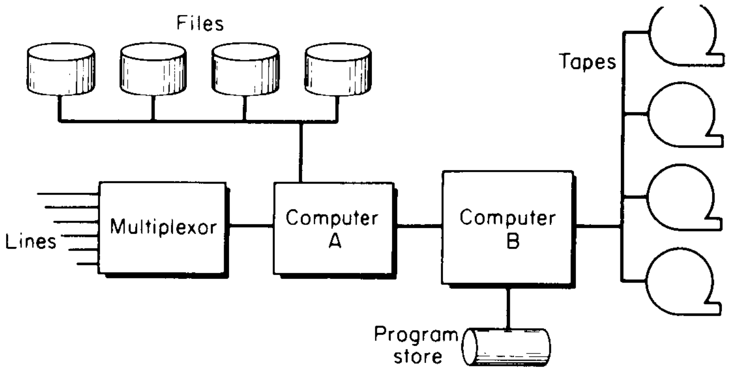


Figure 1-1-4 A multicomputer system.

similar machines backing each other up will have a breakdown time of approximately 0.04 percent. "Duplexing" in this manner increases the equipment complexity, especially if switchover on failure is to be automatic. A duplexed version of the system in Figure 1-1-3 is shown in Figure 1-1-5. Many systems of this type have been installed.

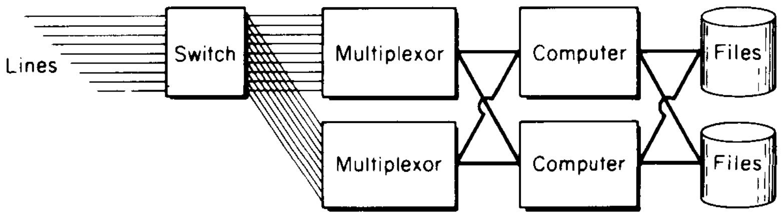
It will be seen here that if a Line Control Unit, computer, or file fails, the system can be switched so that the duplicate takes over.

The cost of this may not be as prohibitive as it seems at first sight if the standby computer can be doing other work while standing by. The complexity of the Supervisory Programs, however, is increased. Every time a file is updated, for example, it is necessary to update both files. If a file breaks down and is later returned to use, it must be quickly updated with all that it has missed, and this updating must not interfere with current work using the files.

Triplexing the equipment would, of course, give even higher reliability but would be even more expensive. On certain special systems, however, very high reliability is essential, whatever the cost. On the American [space] shots, for example, the monitoring computer [may even] be quadruplexed.

The configurations illustrated above are the types in common usage. It is possible to have systems that are much more complex than these, often with more than two interconnected computers. In systems where the interval between message arrivals is short and the file access time long, one computer can-

Figure 1-1-5 A duplexed system.



not cope without multiprogramming, that is, handling two or more transactions at once. In this case, it has been suggested that several small computers should be used instead and the work split between them. The programming techniques described . . . relate to systems such as those above. However, the principles and conclusions that emerge would be applicable to any configuration, not only to these more common ones.

1-1-2 THE RESPONSE TIME

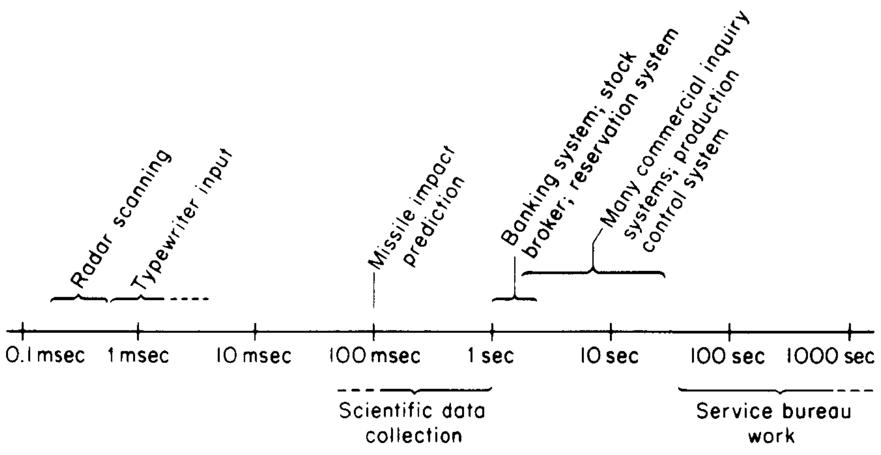
The response time, in this discussion, is the total time a transaction remains in the computer system, that is, from the time at which it is completely received to the time at which a reply starts to be transmitted, or, if there is no reply, the time at which processing is completed.

In a simple case, then, the response time is the time the computer takes to interrupt what it was doing and to process the transaction. There may, however, be certain delays involved in the response time. First, there may be several transactions contending for the computer's time, so that the transaction may have to wait in various queues, like a customer going to the Motor Vehicle Licensing Office at a peak period. Second, the computer may, in some applications, be doing another job and the transaction will have to wait until it is convenient to process it.

In some types of systems a high-speed response time is necessary because of the nature of the work. The computer has to be programmed to react quickly. In others it does not matter—a response time of twenty seconds may be adequate.

The range of response times in some existing applications is shown in Figure 1-1-6. A bank teller or an airline reservation clerk may desire a response

Figure 1-1-6 Examples of required response times.



time of three seconds or less, so as to give customers or telephone enquirers the best possible service. A warehouseman making stock inquiries may be content with a reply in twenty seconds. For controlling a petroleum plant five minutes may be adequate, and for sending instructions to the shop floor of a factory perhaps a half-hour is soon enough. Some scientific control and data logging applications require . . . much shorter response times than these. Examples are data logging on a jet engine or rocket motor test bed, scanning radar readouts or tracking a missile to predict its impact point. The interval between events may be only a few milliseconds, and the response must be programmed not to exceed this brief period. A clock or similar device in a computer is used to prevent the computer from being tied up on one transaction so that it cannot provide this response time when required.

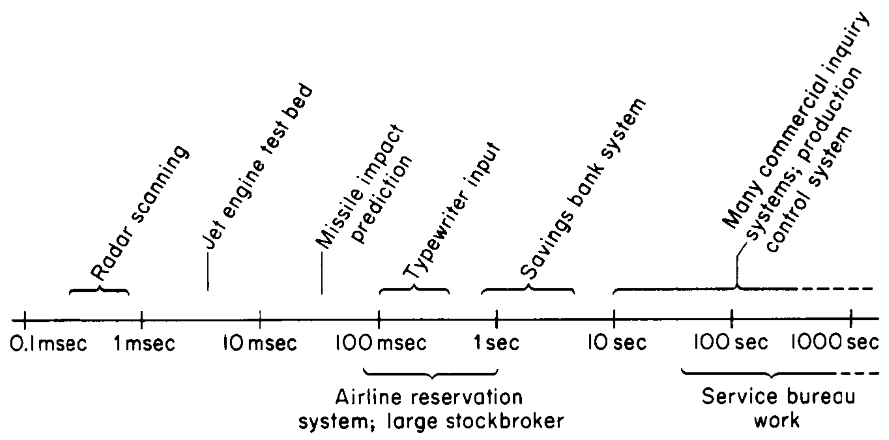
1-1-3 THE INTERVAL BETWEEN EVENTS

The interval between the arrival of transactions at the computer may be random and determined by external events such as a clerk pressing a key; or it may be cyclical and governed by a clock or scanning device in the computer.

As with the response time, it may vary from a fraction of a millisecond to a half-hour or more. The range of interarrival times is shown in Figure 1-1-7 for some existing applications.

An airline reservation system with a thousand terminals may have transactions pouring into it from all over a country at a rate that will be as high as twenty per second at peak periods. On the other hand, some inquiry systems may [have only] an occasional inquiry now and then. A savings bank with a steady stream of customers into each of its branches at lunchtime might average about one transaction every two seconds. In a European bank with a large

Figure 1-1-7 Examples of intervals between message arrivals.



number of branches this could be much higher. A typist keying characters into an on-line terminal may send them at a rate of five to ten per second.

When considering transaction rates for random systems like these, it is necessary to examine the times of maximum traffic because the system must be built to handle these peaks. Indeed, it must in some way cater to the very rare circumstance of all the terminal operators pressing their buttons at the same instant. It will not attempt to process a flood of messages of this magnitude at once, but, on the other hand, if a momentary transaction peak reaches the computer, none of these messages must be lost.

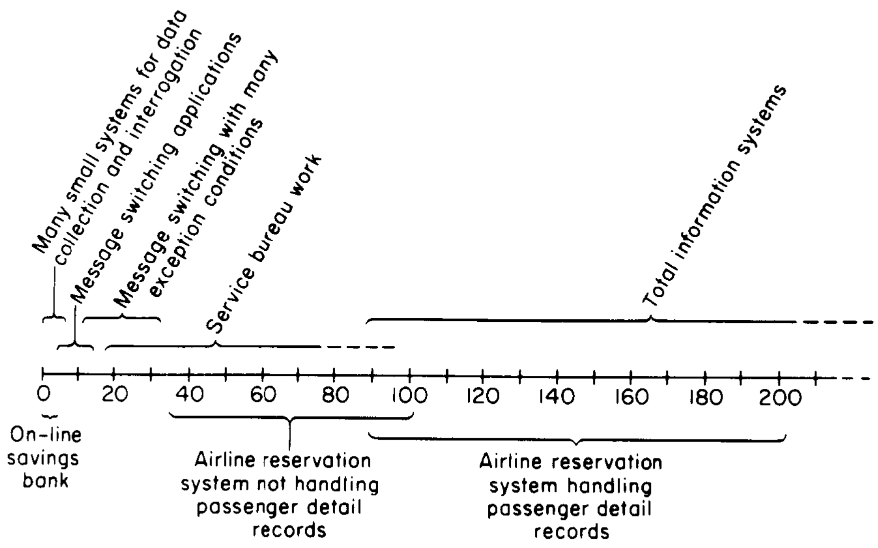
In radar scanning or data logging applications the inputs are scanned with a fixed cycle time. This will probably be of the same order as the response time quoted above.

1-1-4 THE NUMBER OF INSTRUCTIONS IN THE APPLICATION PROGRAMS

The variation in the number of instructions in the real-time Application Programs gives a good indication of the range of complexity of these systems. Some small systems have less than a thousand instructions but the big ones exceed 200,000. Figure 1-1-8 illustrates this spread.

These figures are for the real-time programs only. In other words, this mass of coding is in the system ready for use at any one time. The programs must all fit together like cogs in a machine. Many nonreal-time systems have a

Figure 1-1-8 Number of instructions for on-line work.


Many small systems for data collection and interrogation
Message switching applications
Message switching with many exception conditions
Service bureau work
Total information systems
On-line savings bank
Airline reservation system not handling passenger detail records
Airline reservation system handling passenger detail records