Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

Mine Design

Examples Using Simulation

John R. Sturgul

Published by the
Society for Mining, Metallurgy, and Exploration, Inc.

Society for Mining, Metallurgy, and Exploration, Inc. (SME)
8307 Shaffer Parkway
Littleton, CO, USA 80127
(303) 973-9550
www.smenet.org

SME advances the worldwide minerals community through information exchange and professional development. With more than 16,000 members in 50 countries, SME is the world's largest professional association of mineral professionals.

Copyright © 2000 Society for Mining, Metallurgy, and Exploration, Inc.

Student GPSS/H and Proof Animation Demo Viewer are copyrighted material of, and distributed with permission of, Wolverine Software (www.wolverinesoftware.com).

All Rights Reserved. Printed in the United States of America

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

ISBN 0-87335-181-9

Library of Congress Cataloging-in-Publication Data

Sturgul, John R.
Mine design : examples using simulation / John R. Sturgul.
p. cm.
Includes index.
ISBN 0-87335-181-9 (pbk.)
1. Mining engineering–Computer simulation. 2. GPSS/H (Computer program language) I. Title.

TN153.S83 1999
622'.01'13–dc21

99-048032

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

Preface

Computers, especially the personal computer or PC, are now a standard tool of the mining engineer. The popularity of the personal computer has made it possible for the mining engineer to have immediate access to his or her own computer. Along with this increase in computers, there are now a multitude of software packages available. A casual reading of any of the popular computer periodicals reveals what may seem like a staggering number of such packages. In addition, there has been a burgeoning of computer languages that are not traditionally taught in mining programs. A problem now facing the mining engineer is how to use these computers, the software, and the languages to best advantage.

This book is intended to show how a particular computer language, GPSS (General Purpose Simulation System), can be applied to a wide variety of problems that arise in everyday situations in mining. This book also is intended to be used as a supplement to normal instruction in various mining engineering classes, including those on mine design, mine equipment selection, or computer applications in mining. There are numerous exercises at the end of most chapters, and all have the answers given at the end of the book. It is not assumed in this book that the engineer has a working knowledge of any computer languages as the programs all run by keying in a simple list of instructions.

This book is divided into three parts. Part I is a review of simulation in mining and a discussion of why the GPSS/H simulation language (a version of GPSS) was selected. It is interesting to note that mining engineers were eager to embrace the computer as soon as it was made generally available for use in mining applications, including mining simulation. Part II is an introduction to the GPSS/H simulation language and is divided into 28 chapters. In order to obtain the most benefit from the mining examples that are given in Part III, a

knowledge of GPSS/H as presented in Part II is essential. The student version of GPSS/H can be used to learn the language. A bonus from studying GPSS/H: As one learns more about the language, one automatically learns how to do simulation studies.

If one were to look first at the computer programs that are used for the examples for Part III, they would seem strange. However, as familiarity with the GPSS/H language grows, so does one's appreciation of the power of GPSS/H, the speed at which it can model systems, and the compactness of the programs. All of the program blocks (i.e., lines of code) used in the 30 examples from Part III are covered in Chapters 5 to 31 of Part II. Chapter 32 includes a few additional GPSS/H blocks that might be useful in other mining applications.

Part III is contained on the CD included with this book. It encompasses 30 examples from a wide variety of mining operations, some large, some small, some from surface mines, some from underground mines, some foreign, and some domestic. The suggested way to approach each of these examples is to read the explanation of the example and run the program–via the student version of GPSS/H included on the CD–for the example by using the sample data given with it. The program to simulate each of the examples is interactive; the user is prompted to input the data used in the simulation. The animation can then be run to view the mining operation on the screen in "cartoon" fashion. There are also exercises to go with the examples that should further increase one's knowledge of mine system simulation; the answer to each of these exercises is given.

It may seem strange to introduce the mining engineer to a new computer language, especially one that is not taught in traditional mining programs. Most (if not all) mining engineers are exposed to a problem-solving language such as Fortran or BASIC (and, in some cases, Pascal or C) early in their education. It will be a pleasant surprise to learn that programs for simulation problems that take thousands of lines of computer code in Fortran can be written, in many cases, in GPSS/H with 100 (or fewer) lines of code. For example, consider the basic queueing model of a single server (a shovel loading trucks in a mine), infinite population (there are numerous trucks in the mine), random arrival (the trucks arrive at no set time), and random service (the loading times are variable). This elementary model is discussed in every book on queueing theory. If the various arrival and service times are given by the exponential distribution, this is known as the M/M/1 model. This problem is modeled in only 7 (!) lines of code in GPSS/H. All of the examples in this book are solved in *less than 125 lines of code* (that's the limit of the student version of GPSS/H). Some of the examples represent quite complicated mining situations.

The practicing mining engineer might simply thumb through this book to learn of the wide variety of problems that can quickly and easily be solved by using GPSS/H. Programs in the GPSS/H language are given for each problem discussed, and they can kept in a file and run with no programming involved. The person experienced with the language, however, might choose to modify one or more of the programs to suit a particular problem. An even more

important point is that, on the CD, animations are given for each example so that the viewer can "see" the results. This ease of program access and visualization of results sets this book apart from other mining books. Not only does one solve a problem in an example, but one "views" the answer.

Since GPSS was first introduced by IBM in 1962, several versions have been in use. The recent versions are quite different from the original GPSS. The one that will be used in this book is GPSS/H, which is designed for the personal computer. Included with this book is a limited (student) version that will allow the user to run all of the programs in the book.

All of the examples have been animated through the use of the software known as PROOF. These animations are "postsimulation" in that the simulation is run first and an output file is created to be used with a layout file. The normal way to do an animation is to first create the GPSS/H program to do the simulation and then convert its results to create an output file that serves as input to PROOF, which runs the animation. The programs to create these animation files generally exceed the limits of the student version of GPSS/H and therefore are not included on the CD. It is beyond the scope of this book to explain animation programs; the interested reader is directed to software providers such as Wolverine Software Corporation, 2111 Eisenhower Avenue, Suite 404, Alexandria, VA 22314-4679, Phone: (703) 535-6760, Fax: (703) 535-6763 (mail@wolverinesoftware.com). Full directions on running the animations are given in Appendix A. In some cases, there will be more than one animation for a single example. These animations are "demo" versions. This means that they have the full capabilities of the regular animations, but changes that the viewer wishes to make cannot be saved.

Appendix A also gives instructions on the running of GPSS/H programs, and a README.DOC file has been provided on the CD. The programs in Part III can be run without any knowledge of the GPSS/H language, but, for maximum benefit, it is hoped that the reader will study the language first. For detailed instruction in the language itself, such as the way the GPSS/H processor works in moving the transactions through the system, the books by Tom Schriber are recommended highly. Both are referenced on the next page. The 1974 book covers the GPSS language more thoroughly than the later one. The 1991 textbook specifically deals with using GPSS/H on PCs and is more concerned with an introduction to discrete system simulation. The GPSS version discussed in 1974 is designed for punched cards and mainframe computers.

For the person who already is proficient in programming with a version of GPSS for the mainframe, the differences between GPSS/H and the mainframe version can be learned from reading Schriber's 1991 textbook. Be forewarned, however, that the differences are many! The main features of GPSS/H used in the examples in this book include a floating-point clock, the ability to include DO loops, referencing of functions via parentheses, the use of symbols such as "=," "AND," "OR," etc., in logic statements, and, of course, the customized reports.

REFERENCES

Schriber, T.J., 1974. *Simulation Using GPSS.* New York: John Wiley and Sons. This reference is now out of print but available from Krieger Publishing Company, P.O. Box 9542, Melbourne, Florida 32902. This is the book referred to simply as "Big Red" or the "Red Book."

Schriber, T.J., 1991. *An Introduction to Simulation Using GPSS/H.* New York: John Wiley and Sons. This reference includes the student version of GPSS/H.

John R. Sturgul
Moscow, Idaho

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

Contents

PREFACE v

ACKNOWLEDGMENTS ix

PART I — GPSS/H AND SIMULATION IN MINING 1

INTRODUCTION 3

A BIT ABOUT GPSS/H 7

REVIEW OF SIMULATION IN MINING 13

REVIEW OF SIMULATION MODELS 19

PART II — THE GPSS/H SIMULATION LANGUAGE 33

SAMPLE GPSS/H PROGRAMS 35

THE GENERATE AND TERMINATE BLOCKS AND THE START STATEMENT 51

THE TRANSFER BLOCK 65

THE PUTSTRING AND PUTPIC STATEMENTS AND WRITING TO FILES 77

THE ADVANCE BLOCK 87

QUEUE AND DEPART BLOCKS 93

SEIZE AND RELEASE BLOCKS 99

ENTER AND LEAVE BLOCKS 111

THE CLEAR, RESET, AND RMULT STATEMENTS 127

FUNCTIONS 133

MORE ON STANDARD NUMERICAL ATTRIBUTES: ARITHMETIC IN GPSS/H 151

THE TEST BLOCK 157

GPSS/H SUPPLIED FUNCTIONS 167

PARAMETERS, THE LOOP BLOCK, AND THE EQU STATEMENT 179

TABLES IN GPSS/H 197

SAVEVALUES 207

LOGIC SWITCHES AND GATES 217

OTHER FORMS OF THE TRANSFER BLOCK 227

AMPERVARIABLES; DO LOOPS; GETLIST STATEMENTS; IF, GOTO, HERE, AND LET STATEMENTS 241

THE SELECT AND COUNT BLOCKS 255

MATRICES 267

VARIABLES, EXPRESSIONS, AND THE PRINT BLOCK 283

BOOLEAN VARIABLES 289

THE BUFFER BLOCK 301

THE SPLIT BLOCK 307

ASSEMBLY SETS AND THE ASSEMBLE, GATHER, AND MATCH BLOCKS 321

MACROS 335

OTHER GPSS/H BLOCKS 341

PART III EXAMPLES ON THE CD 349

APPENDIX A: RUNNING THE PROGRAMS AND ANIMATIONS 351

APPENDIX B: SNAS USED IN BOOK 357

APPENDIX C: HISTORICAL GPSS/H FORMAT 361

INDEX 363

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

PART I

GPSS/H and Simulation in Mining

CHAPTER 1

Introduction

GPSS (General Purpose Simulation System) is an extremely versatile computer programming language. Originally developed in 1961 to solve very simple simulation problems, it has evolved until, in its present state, it can solve a wide range of problems. Many of these problems are of direct concern to the mining engineer. Benefits of using GPSS include the following:

1. It is easy to follow the logic of the programs.
2. The programs are short. Hence, they are quickly written.
3. Execution times are short.
4. Versions are available for PCs.
5. The programs are easy to change as modifications in the model are required.
6. It is a dynamic language in that it constantly is being modified and improved.
7. It has a proven track record for simulating a wide variety of mining problems.
8. The last benefit is possibly the most important–it allows problems of a practical nature to be solved quickly and easily.

GPSS/H–the version of GPSS used in this book–is not, however, an easy language to master. Like all practical computer tools, it takes time and effort to learn all of its many facilities. However, just as a person can use a common spreadsheet without knowing all of its many features, so can one use the GPSS/H programs in this book without being a GPSS/H expert.

This book will introduce the mining engineer or student to some of the many possible applications of GPSS/H. The approach is to present different mining problems for which the solution can be obtained by using GPSS/H. If a person attempted to use a traditional engineering language such as Fortran to model these problems, the programming effort would be on the order of 100 times as

great as that required when using GPSS. In most, if not all cases, it is doubtful if most mining engineers would have the programming skills or patience to write the Fortran code. In fact, it is doubtful if many actual mining operations could even be simulated through the use of traditional engineering languages. However, when using GPSS/H, this is not the case. Now the mining engineer has time to ensure that the computer model does indeed represent the real-life situation, by making sure that the data used are correct and by making refinements to the model as needed. The actual programming to solve the problem should be a minor part of the whole problem-solving process.

Part II of this book is an introduction to the GPSS/H language. A knowledge of this is important for any mining engineer who wants to understand how the models in Part III were constructed and who eventually wants to construct his or her own models. The language is presented with numerous examples that the reader should run. The solutions to all of the exercises at the end of each chapter in Part II are given at the end of the book. There is only one way to learn a computer language, and that way is to run numerous examples. Each time an exercise is successfully run, the student of GPSS/H will gain something. GPSS/H is fairly easy to debug and the compiler will attempt to determine the place and type of any error. Part II was written by using notes from numerous short courses on the subject. However, what is presented in Part II is in much more detail than what can be taught in a four-day short course. Also, there are many more exercises.

Each mining example in Part III is followed by the GPSS/H program for its solution. No fluency with GPSS/H is needed to begin using the book other than a knowledge of how to run the programs on the computer and how to interpret the results. Naturally, to obtain the most benefit from this book, one should be familiar with how GPSS/H works. Thus, a few remarks about the language follow.

GPSS is a programming language that was introduced originally to assist in solving problems having to do with manufacturing. All the versions of GPSS are used to build computer models to represent systems that can be characterized as consisting of discrete events. A discrete event is one in which, during a small increment of time, the state variables change only a countable number of times. Examples of discrete events include a truck being loaded, a shovel breaking down, a quantity of ore arriving at the end of an ore pass, a ship arriving in a port, etc. Since mining operations can be viewed as consisting of a large number of such separate events, the GPSS language is a natural for use here. In fact, even for situations in which more than one thing happens at the same time (truck A is loaded at, say, simulated time 10023, and truck B finishes dumping its load at the same time), GPSS can still be used.

GPSS in all its versions is a nonprocedural language (Fortran, BASIC, Pascal, etc., are procedural languages). This distinction means, briefly, that GPSS is designed for a special purpose and tries to anticipate what the programmer is doing. For example, one need not use any commands for generating output:

GPSS assumes that the programmer is interested in the results of a simulation, and so automatically it keeps track of data relevant to simulation studies and presents the results at the end of the program.

The first few examples in this book are short, and the programs are easy to follow by beginning GPSS/H programmers. However, the rest of the examples are not given in any particular order. In some cases, exercises are given for the interested reader to work in order to extend the solution. Some of the examples were selected from actual mining situations with few, if any, changes to the data. In a few cases, the examples were selected from ones that appeared in professional papers. These were problems that authors had indicated were solved by some form of computer simulation, although not necessarily GPSS/H. These problems are now resolved by using GPSS/H with minor changes to the problem or data as were required. Some of the examples illustrate the great versatility and power of GPSS/H; others were chosen to illustrate how some feature of GPSS/H can be used to solve a particular mining problem. A discussion of the particular GPSS/H features used to model each problem is kept separate from the problem statement and solution. Thus, a person who is not familiar with the language can omit this section in each chapter.

It is hoped that, after the engineer or student has studied and run the programs that are presented here, she or he will have obtained both a degree of fluency in constructing simulation models and an appreciation of their purpose. Since GPSS/H is a language and not a package, each time the problem changes, a new program has to be written or a previous one modified. However, by studying and running the examples here, the general form of the programs will be learned. The best way to use this book is to run each of the examples and then pose the "What if?" questions that come to mind. The programs can be suitably changed to answer most of these questions; often only a few changes to the program are required. By following this approach, one can also learn a great deal about the power of the GPSS/H language.

For the engineer who has not used a programming language for some time, there is no reason to be concerned about being able to learn GPSS/H. The language is quite different from most other languages (it does resemble a few other simulation languages but is nothing like Fortran). In fact, knowledge of other computer languages can be a bit of a hindrance at times. It is assumed in this book that the engineer has a working knowledge of Fortran, but this assumption has been made primarily to ease the pointing out of the differences in the languages. The benefits gained from learning GPSS/H will be worth the time and effort needed to learn how to use it.

Since this book is designed as an introduction to the types of problems that can be solved by simulation models, the engineer may wonder how long it will take to actually build a simulation model of a large, working mine. The answer is that, by the end of three days of intensive instruction, he or she should be able to have a degree of confidence in understanding GPSS/H programs for simple mining problems. One learns that, in going from many simple mining

situations to more complicated ones, the GPSS/H program involved is not hard to write—only more lines of code are needed. Programs become longer but not more difficult. This cannot be said for Fortran.

Mastering GPSS/H is not simple. For this reason, an introduction to the language is provided in Part II. However, there is no quick and easy way to learn *any* computer programming language, and GPSS/H is no different. In order to really be able to apply GPSS/H to complicated mining problems, one must be prepared to spend many long and hard sessions at the computer. But once a person makes enough programming errors and then learns why the program did the wrong thing, the next program becomes just that much easier to write. The skills gained in learning GPSS/H will enable the mining engineer to solve problems that were beyond his or her reach previously.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 2 A Bit About GPSS/H

This chapter is intended to answer questions about the GPSS/H language and is presented in a question and answer format.

WHY USE THE GPSS/H LANGUAGE?

All of the examples in this book are solved by using the GPSS/H simulation language. Since this computer language is generally not the first one that mining engineers have been trained to program in, it is appropriate to learn why GPSS/H was selected. In fact, there are multiple versions of GPSS available. Of all the GPSS versions available, however, GPSS/H is by far the most advanced. It was designed specifically for the personal computer. It contains features that the other versions of GPSS do not have. Switching from one version to another is not at all difficult, but the user should be aware that many of the features of GPSS/H will not work on other versions.

WHAT IS GPSS/H?

GPSS/H (the version of the General Purpose Simulation System developed by James O. Henriksen, which explains the H) is both a computer language and a computer program. It was designed for studying systems represented by a series of discrete events. GPSS/H is a low-level, nonprocedural language.

WHAT IS A DISCRETE SYSTEM?

A discrete system is one in which only a countable number of events can occur at one time. These discrete events might be trucks being loaded, ships entering a harbor, people entering a bank, cars traveling on a road, parts moving on a conveyor belt, etc.

WHERE DID GPSS/H COME FROM?

GPSS originally was developed by Geoffrey Gordon for IBM in the early 1960s, so it has been around for quite some time. However, it is a dynamic language in that new versions such as GPSS/H are introduced every three or four years. It is now a multivendor language, and various versions are available. It is widely used on both mainframes and PCs. By 1972 there were at least 10 versions of GPSS available, and many of these have survived in one form or other. Greenberg (1972) presented details about these early versions of GPSS and traced their histories.

WHAT ABOUT MODERN VERSIONS OF GPSS?

Since GPSS has been around for quite some time, it is natural that people who were introduced to it in its early stages may not be aware of how it has changed. An excellent summary of its development is given by Schriber (1988). Schriber has listed the common, modern versions of GPSS (GPSS/H, GPSS V, GPSS/PC, GPSSR/PC, and GPSS/VX) and where to obtain relevant information regarding each. The latest versions of GPSS/H for the PC include animation display so that the simulation can be viewed in "cartoon" fashion. In fact, it is possible to find certain functions performed by GPSS/H embedded in other languages such as GPSS-Fortran, APL-Fortran, and PL/1-GPSS. Schriber's paper gives references for these.

Henriksen (1983, and subsequent company updates) has dispelled some of the myths that have grown up surrounding GPSS/H.

1. GPSS/H is an inherently slow simulation language. This is no longer the case. In fact, comparisons of GPSS/H with other simulation languages by Abed et al. (1985a, 1985b) show that GPSS/H is many times faster than other languages such as SLAM and SIMSCRIPT.
2. To make sophisticated simulations with GPSS/H, one must eventually revert to other computer languages. This is rarely the case anymore. In fact, none of the examples presented in this book reverts to any other computer language.
3. Learning GPSS/H is trivial. Any computer language takes time and practice to master, and GPSS/H is no exception. Most industrial short courses last four to five days by which time the participants have a sound introduction to the language. It is normally taught on the university level as a full semester course.
4. Modeling difficulties more often are due to shortcomings in the computer language used than the modeler's level of expertise. This is certainly not the case with modern versions of GPSS/H. Gordon (1978) and Henricksen (1983) gave examples of people who blamed GPSS for their lack of expertise in solving simulation problems when the real fault lay with their own lack of programming expertise.

WHAT IS A NONPROCEDURAL LANGUAGE?

A nonprocedural language is one that anticipates what the programmer is attempting to do and allows the computer code to be very short. Often the programming code for a nonprocedural language appears very similar to the problem it has been designed to solve. For example, an array of data may be sorted by using a procedural language in one of several ways. One way is to find the smallest (or largest) element, place this at the front of the array, and then sort through the remaining elements for the next smaller (or largest), find it, and place it second in the list, etc., until the array is sorted. This method involves numerous comparisons of data and, thus, many lines of code. A nonprocedural language that is used for handling databases that commonly must be sorted may have a single command–namely, SORT–to do this task. In another example, simulation studies often involve queues. To model a queue in GPSS/H to gather certain statistics, the single line of code (known as a block) might be

```
          QUEUE     DUMP
```

No code for output is needed: GPSS/H automatically will gather relevant statistics and output them when the program is finished.

People seeing a GPSS/H program for the first time tend to remark "Is that all there is to it?" In the study of queueing theory, one soon encounters a system having a single server for people arriving at random times from an infinite population. One case of this is known as the M/M/1 (the first M, which comes from Markov, indicates that the arrival rates are Poisson, the second M for exponential server, and the 1 indicates a single server). This system is modeled in GPSS/H by using only 7 (!) programming blocks. The equivalent Fortran program would take many hundreds of lines of code. In addition, to make changes in a GPSS/H program to answer the "What if?" questions often takes only a few lines of code. If a system is being studied with room for only 8 trucks, the relevant line of code may be

```
TRUCKS    STORAGE   8
```

To study the same system but with room for 9 trucks may involve changing only the above line to

```
TRUCKS    STORAGE   9
```

IS GPSS/H HARD TO LEARN?

GPSS/H is not any more difficult to learn than any other programming language. Most people find it easier to learn than traditional engineering languages such as Fortran, BASIC, or Pascal. After about 30 or 40 hours of instruction, most engineers find that they can proceed on their own with writing practical simulation programs. Since it is a very popular language, numerous short courses are held throughout the world.

WILL A KNOWLEDGE OF OTHER LANGUAGES HELP TO LEARN GPSS/H?

The logic behind GPSS/H is so different that knowledge of other procedural languages may be a hindrance. Of course, knowledge of any other simulation language is a different matter.

WHAT ABOUT USING OTHER SIMULATION LANGUAGES?

Other simulation languages exist that are quite good for solving simulation problems relating to mining. Some of these are SIMAN (ARENA), SIMCRIPT II.5, and SLAM. The solutions obtained by investigators using these languages may well be as accurate as those obtained by using GPSS/H.

WHY CHOOSE GPSS/H?

GPSS/H was selected as the language for this book for the following reasons:

1. It is multivendor so it is continually being upgraded.
2. It is widely available.
3. It is written in machine language and, therefore, is inherently very fast.
4. It can solve a wide variety of problems rapidly and accurately. These problems come from many sectors, such as manufacturing, engineering, business, and science.
5. It has withstood the test of time, having been introduced by IBM in 1961. Other simulation languages have fallen by the wayside.
6. It has proved to be extremely versatile for modeling mining and mining-related operations. These include both surface and underground operations as well as material flow through a smelter, mill, and refinery.
7. It is easily coupled with PROOF for making animations.

WILL GPSS/H REPLACE LANGUAGES SUCH AS PASCAL, BASIC, OR C?

No. There are a large number of problems that should and always will be solved through the use of traditional computer languages. Furthermore, there is the possibility of having packages available based on traditional languages to solve mining problems. Learning GPSS/H enhances a person's computer skills rather than replacing any. In this regard it can be looked upon as adding a computer skill such as word processing or learning how to construct a spreadsheet. Knowledge of these techniques does not replace programming skills using traditional procedural languages.

HOW ABOUT A COMPARISON BETWEEN FORTRAN AND GPSS/H?

Here is a rough comparison. The actual values will depend on the particular problem. Consider the problem of writing the computer code to simulate the operation of a large openpit copper mine such as the Panguna Mine on the island of Bougainville, in the South Pacific.

	GPSS/H	Fortran
Time to write program	few days	many months
Execution time	<1 minute CPU	3–4 hours CPU (386)
Ease of changing program	trivial	up to a week
Lines of computer code	300–400	20,000–50,000
User friendly?	yes	no
Graphical output	a few lines of code	many additional lines of code
Animation?	available	not standard

BUT AREN'T FORTRAN PACKAGES ON THE MARKET?

Yes, but these tend to be very expensive (some around $50,000), hard to change, and not user friendly. Also they take a great deal of CPU time to run. These tend to solve only a severely restricted class of problems. In addition, animation is not available.

IS GPSS/H WIDELY USED IN OTHER SPECIALTIES?

Actually, GPSS/H is arguably the most widely used computer simulation language for discrete system simulation. It is used throughout the manufacturing industry.

WHY IS GPSS/H NOT USED MORE BY MINING ENGINEERS?

There are several reasons. Mining engineers do not normally obtain an early exposure to it since their training in computer languages is confined to Fortran or Pascal. Graduate students often learn GPSS/H for their individual research projects but not for teaching purposes. Some engineers may have heard of an early version of it and not be aware of the tremendous advances that have been made. However, it is now common to have instruction in static modeling for mine design. These models are the computer-aided mine design packages that assist the mining engineer to display and study the mine via computer terminals. The next logical progression is to use the computer to study dynamic simulation models of the operating mine. Thus, the use of simulation models should increase as mining engineers continue to develop skills in computer applications in mining.

REFERENCES

Abed, S.Y., Barta, T.A., and McRoberts, K.L. 1985a. A Qualitative Comparison of Three Simulation Languages: GPSS/H, SLAM, SIMSCRIPT. *Computers & Industrial Engineering,* no. 9:35–43.

Abed, S.Y., Barta, T.A., and McRoberts, K.L. 1985b. A Quantitative Comparison of Three Simulation Languages: GPSS/H, SLAM, SIMSCRIPT. *Computers & Industrial Engineering,* no. 9:45–66.

Gordon, G. 1978. The Development of the General Purpose Simulation System (GPSS). *In* Proceeding of the ACM SIGPLAN History of Programming Languages Conference. New York, Association for Computing Machinery, *SIGPLAN Notices,* 13(8):183–198.

Greenberg, S. 1972. *GPSS Primer.* New York: Wiley-Interscience.

Henriksen, J.O. 1983. State-of-the-art GPSS. *In* Proceedings of the 1983 Summer Computer Simulation Conference. San Diego, The Society for Computer Simulation, 918–913. (Wolverine Software has published updates of this article. The updates are available from Wolverine Software Corporation, 7617 Little River Turnpike, Suite 900, Annandale, Virginia 22003-2603; 703-750-3920 [telephone]; 703-642-9634 [fax]; mail@wolverinesoftware.com.)

Schriber, T. 1988. Perspectives on Simulation Using GPSS. *In* Proceedings of the 1988 Winter Simulation Conference. Edited by M. Abrams. San Diego, The Society for Computer Simulation.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 3

Review of Simulation in Mining

Mining engineers have been interested in using computers to build simulation models of mining operations ever since the computer was introduced and accepted into their industry. This chapter, modified from Sturgul (1996), gives a brief history of the development of such models in the mining industry.

SOURCES

Excellent sources of references for models developed in the 1960s and 1970s when computers were first being used by the mining engineer are the proceedings from the APCOM (Application of Computers and Operations Research in Mining) conferences. The first of these was held in 1961 at the University of Arizona in Tucson, Arizona, USA. Since then, there have been over 20 such conferences held in many different countries, and similar conferences are now held in Canada and Australia. Conferences such as the International Symposium on Mine Planning and Equipment Selection, which are held throughout the world, also provide a valuable source of references.

EARLY SIMULATIONS OF QUEUEING PROBLEMS

Many of the models encountered in mining are examples of queueing theory, such as trucks arriving at a loader or a dump. A knowledge of the concepts of queueing theory is essential for anyone interested in building and understanding simulation models. There are textbooks devoted to just this subject, and it is covered in any modern treatment of operations research.

Koenigsberg (1958) published one of the first, if not the first, papers on queueing theory applied to mining. This was, in fact, the first published result of a mathematical solution to a general cyclic queueing theory problem. Koenigsberg solved the problem of determining the production for a set number of crews working at the faces for several underground coal mines. In one case, there were 5 crews (cutting, drilling, blasting, loading, and bolting) and 5 faces. The order of working was set, and only one crew could work at one

face at a time. When a crew finished at one face, it went to the next face and, if no crew was working on it, started its job. If another crew was at the face, the arriving crew had to wait in a queue. The crews did not leave the system and cycled from face to face (hence, the name "cyclic queues"). The distribution of the range of times to do each operation was found to be exponential. Koenigsberg was able to obtain an exact mathematical solution and compare his results with actual mines in the state of Illinois, USA. Koenigsberg's problem is presented as Example 16 in Part III with several variations added. An excellent summary and review of cyclic queues is given by Koenigsberg (1982). Cyclic queueing theory has been used in many industries such as shipping, manufacturing, and electronics.

EARLY SIMULATIONS OF MINING OPERATIONS

Rist (1961) published the first work of a computer simulation of a mining operation. This paper was presented at the first APCOM as well as published in a trade journal. Rist's problem was taken from an actual underground molybdenum mine where his model was used to determine the optimum number of trains to have on a haulage level. Loaded trains had to queue at a portal and wait until the single track was clear as well as wait until the crusher area was free (only two trains could be at or waiting at the crusher). Other restrictions were imposed on the system to make it as lifelike as possible. Rist used a now obsolete version of one of the first attempts at a simulation language. A modification of his model is given as Example 18 in Part III. Several years after Rist's work, Harvey (1964), also at an APCOM meeting, elaborated on Rist's model and specifically mentioned that the GPSS language was used in the simulation. This was the first reported application of a modern simulation language to a mining problem.

During the 1960s other investigators were building computer simulation models of mining operations. However, the computer language used was primarily Fortran. Although this choice was logical since mining engineers were taught this language and it was (and is) universally used, it was not the most efficient language to use. Programming was much slower when using punched cards, and considerable time and effort were spent in writing and debugging the programs. Few mining engineers were interested in learning a second or third language.

One particular problem in mining that concerned engineers was to construct models of conveyor belts, especially for underground coal mines. Sanford (1965) appears to have been the first to undertake this problem; by using Fortran, he simulated a conveyor belt system for his masters' thesis at Pennsylvania State University. In 1968, a large Fortran program, BELTSIM, was developed at the Virginia Polytechnic Institute and State University (described by Bucklen et al. 1969). Talbot (1977) gave an account of an application of BELTSIM to an Australian mine. For a coal mine, Juckett (1969) designed a belt system that could handle up to 25 belts and 12 loading points; the programming language was PL/1. BETHBELT-1 was written for the Bethlehem Steel Corporation by Newhart (1977), who used GASP V, which is similar to GPSS in that it is a language designed for discrete-event

simulation. Hancock and Lyons (1987) described a package known as SIMBELT4 and reviewed the work done by the National Coal Board in England. SIMBELT4 was designed to predict the rate of flow at various points in a belt system and to estimate the ore in each storage bunker.

Suboleski and Lucas (1969) discussed a Fortran program (SIMULATOR1) that would simulate room-and-pillar mining operations. O'Neil and Manula (1967) used a simulation model for materials handling in an openpit mine, and Manula and Venkataraman (1967) were able to simulate truck haulage in an openpit. Waring and Calder (1965) discussed a simulation package developed for a particular mine in Canada. Madge (1964) was able to simulate truck movement in an openpit mining operation, also in Canada. The outline of a large package capable of simulating a wide variety of operations was described by Manula and Rivell (1974).

One of the best early examples of a computer simulation model for a working mine was by Cross and Williamson (1969). This model had to do with a working openpit copper mine in the Southwest of the United States. The mine was a truck-and-shovel operation with 5 shovels loading trucks that dumped at the ore crusher, the waste pile, or a leach area. Initially, the trucks were dedicated to a particular shovel. A simulation model was then constructed to determine if a dispatcher could be used to route the trucks to different shovels so as to minimize queueing time. The model assumed that all of the times were deterministic and was able to indicate that a dispatcher would indeed improve operations. The program, in Fortran, consisted of several thousand lines of code and took considerable time to write. Part III of this book contains several examples, such as 1, 2, 3, 6, and 12, that are variations of this problem.

A more complicated model based on the paper by Cross and Williamson (1968) and using stochastic times for the various operations was published by Sturgul and Yi (1987) to demonstrate the power of the GPSS language. Their program consisted of fewer than 100 lines of code and, what is more important, took less than one hour to write.

In a benchmark paper, Bauer and Calder (1973) pointed out the advantages of using GPSS for openpit operations, especially to simulate load-haul-dump circuits. Steiker (1982) reviewed various simulation models and used GPSS for the simulation of an underground train-haulage system. Wilson (1985) gave an interesting background to the decision to use GPSS for the simulation of ore transport by surface rail for platinum mines in Africa. Basically, a local university reviewed the ore transport problem and suggested that the mining engineers involved learn and use GPSS. Students hired during the summer were given the responsibility of gathering the relevant statistics while the model was written and refined.

PERSONAL COMPUTERS AND MINING SIMULATIONS

With the advent of personal computers and on-screen editing capabilities for rapid writing and debugging of programs, the mining engineer has been given new tools to assist him or her in the use of computers. Other languages

such as Pascal and C are being taught and used by engineers. The time required to learn new languages has been reduced substantially. Many engineers now are exposed to special-purpose languages, such as PROLOG for artificial intelligence or perhaps SQL for use to query a database. A nonprocedural language similar to GPSS is SIMAN. Mutmansky and Mwasinga (1988) presented an overview of this language. SIMAN was used by Tan and Ramani (1988) to study belt networks. One of the strong features of SIMAN is that it can be coupled with CINEMA to provide on-screen animation. In fact, animation is now a feature of nearly all simulation software. Each of the examples presented in Part III has a corresponding animation to view.

The examples in Part III have been simulated with the GPSS/H simulation language. The version used is Student GPSS/H, which is included on the CD-ROM with the book. This software has proven itself ideal for mining examples many times over. The GPSS/H language has been used for a wide variety of mining applications, some of which have been described by Sturgul and Harrison (1987a, 1987b). How the personal computer and GPSS/H can be used for mining applications has been discussed by Sturgul and Singhal (1988) and Sturgul (1987a). Other examples are given by Harrison and Sturgul (1988). Some of the examples in these papers deal with studying how a dispatching system would work in a surface coal mine, determining the height of the wall for a tailings dam for a mine located in the tropics, determining the optimum size of a storage bunker for an underground mine, and analyzing the equipment requirements for a medium-sized open pit iron mine. An example of how to use simulation to determine the optimum location of in-pit crushers is given by Sturgul (1987b).

Some of the examples given in this book represent many of the problems previously published by other researchers, often with changes to the data or variations to the problem. Since the GPSS/H language is so powerful, these variations generally were made to make the problems more complicated. Many examples come from or have been inspired by actual working mines, predominantly in Australia, the United States, and South America.

REFERENCES

Bauer, A., and Calder, P. 1973. Planning Open Pit Mining Operations Using Simulation. In *APCOM 1973*. Johannesburg: South African Institute of Mining and Metallurgy, 273–298.

Bucklen, E.P., Suboleski, S.C., Prelaz, L.J., and Lucas, J.R. 1969. Computer Applications in Underground Mining Systems. Pittsburgh: U.S. Department of the Interior Research and Development Report 37.

Cross, B., and Williamson, G. 1969. Digital Simulation of an Open-Pit Truck Haulage System. In *APCOM*. New York: Society of Mining Engineers of the American Institute of Mining, Metallurgical, and Petroleum Engineers.

Hancock, W.S., and Lyons, D.K.G. 1987. Operational Research in the Planning of Underground Transport. In *APCOM 18*. London: Institute of Mining and Metallurgy.

Harrison, J., and Sturgul, J. 1988. GPSS Computer Simulation of Equipment Requirements for the Iron Duke Mine. Melbourne: Australasian Institute of Mining and Metallurgy, Second Large Open Pit Mining Conference, April.

Harvey, P.R. 1964. Analysis of Production Capabilities. In *APCOM 1964. Colorado School of Mines Quarterly.*

Juckett, J.E. 1969. A Coal Mine Belt System Design Simulation. In *Proceedings, 3rd Conference on the Application of Simulation.* Los Angeles.

Koenigsberg, E. 1958. Cyclic Queues. *Operational Research Quarterly,* 9(1[March]):22–35.

Koenigberg, E. 1982. Twenty Five Years of Cyclic Queues and Closed Queue Networks, A Review. *Journal of the Operational Research Society,* 33:605–619.

Madge, D.N. 1964. Simulation of Truck Movement in an Open Pit Mining Operation. *Canadian Operational Research Society,* 32–41.

Manula, C., and Rivell, R. 1974. A Master Design Simulator. In *12th APCOM. Colorado School of Mines Quarterly.*

Manula, C., and Venkataraman, N. 1967. Open Pit Haulage Simulation. College Station, Pennsylvania State University, Internal Departmental Report.

Mutmansky, J.M., and Mwasinga, P.R. 1988. An Analysis of SIMAN as a General Purpose Simulator for Mining Systems. *International Journal of Surface Mining,* 1–6.

Newhart, D.D. 1977. BETHBELT-1, A Belt Haulage Simulator for Coal Mine Planning. Bethlehem, Pennsylvania: Bethlehem Steel Corporation, Research Department, Research Report File 1720-2.

O'Neil, T.J., and Manula, C.B. 1967. Computer Simulation of Materials Handling in Open Pit Mining. *Transactions, American Institute of Mining Engineers,* 238:137–146.

Rist, K. 1961. The Solution of a Transportation Problem by Use of a Monte Carlo Technique. In *APCOM 1.* Tucson: University of Arizona, March.

Sanford, R.L. 1965. Stochastic Simulation of a Belt Conveyor System. In *APCOM 1965.* Tucson: University of Arizona, March, D1–D18.

Steiker, A.B. 1982. Simulation of an Underground Haulage System. In *APCOM 17.* Golden: Colorado School of Mines, 599–613.

Sturgul, J.R. 1987a. Simulating Mining Engineering Problems Using the GPSS Computer Language. *Australasian Institute of Mining and Metallurgy Bulletin,* 292(4[June]):75–78.

Sturgul, J.R. 1987b. How to Determine the Optimum Location of In-Pit Moveable Crushers. *International Journal of Mining and Geological Engineering,* 5(2).

Sturgul, J.R. 1996. History of Simulation in Mining: 1961–1995. In *Proceedings, First Internet Symposium on Mine Simulation via the Internet,* Athens, Greece, December. Edited by J.R. Sturgul and G.N. Panagiotou. Rotterdam, Balkema.

Sturgul, J.R., and Harrison, J. 1987a. Using a Special Computer Language for Simulation of Coal Mines: *The Coal Journal,* no. 18:21-28.

Sturgul, J.R., and Harrison, J. 1987b. Simulation Models for Surface Mines: *International Journal of Surface Mining,* no. 1:187-189.

Sturgul, J.R., and Singhal, R. 1988. Using the Personal Computer to Simulate Mining Operations. In *Symposium on Computer Applications in the Mineral Industry.* Edited by J.-L. Collins and R. Singhal. Holland: Rotterdam: Balkema, 439-442.

Sturgul, J.R., and Ren Yi. 1987. Building Simulation Models of Surface Coal Mines Using the GPSS Computer Language. *The Coal Journal,* no. 15:11-17.

Suboleski, S.C., and Lucas, J.R. 1969. Simulation of Room and Pillar Face Mining System. In *1969 APCOM, Salt Lake City.* New York: Society of Mining Engineers of the American Institute of Mining, Metallurgical, and Petroleum Engineers, 373-384.

Talbot, K. 1977. Simulation of Conveyor Belt Networks in Coal Mines. In *APCOM 15, Brisbane, Australia.* Melbourne: Australasian Institute of Mining and Metallurgy.

Tan, S., and Ramani, R.V. 1988. Continuous Materials Handling Simulation: An Application to Belt Networks in Mining Operations. Paper presented at Society of Mining Engineers of the American Institute of Mining, Metallurgical, and Petroleum Engineers meeting, Phoenix, Arizona, January.

Waring, R.H., and Calder, P.N. 1965. The Carol Mining Simulator, In *APCOM 1965.* Tucson, Arizona: University of Arizona, Tucson, March, KK1-KK2.

Wilson, J.W. 1985. Simulation of Ore Transport on Surface Rail at Impala Platinum, ltd. In *APCOM 19.* London, Institute of Mining and Metallurgy, 411-418.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 4

Review of Simulation Models

SIMULATION MODELS

The GPSS/H computer programming language is a special language that is used primarily to simulate discrete systems. A discrete system is one in which, at any given instant in time, a countable number of things can take place. Nearly all of the problems one encounters in the study of queueing theory can be represented by discrete systems. Some examples include the following:

1. People entering a barbershop with a single barber. If the barber is busy, people wait in chairs in the waiting area until it is their turn.
2. People entering a bank with multiple tellers. The customers may either form individual queues at each teller or wait in a single queue (known as a "quickline").
3. Trucks working at a construction site where a single shovel loads each truck. The trucks travel to a dump area where they dump and then return to the shovel. This is an example of a cyclic queue. The elements of the system–in this case, the trucks–do not leave the system.
4. Ships entering a harbor with multiple berths. The ships need to be towed into a berth with 1 or more tugboats.
5. Telephone calls arriving at a central switchboard where they need to be routed to the correct extension.
6. Television sets on a conveyor belt arriving at an inspection station. If a set fails inspection, it may be sent back for adjustment, or, in the worst case, it is discarded.

A complete treatment of simulation theory is beyond the scope of this book. However, an understanding of how simulation models are constructed and what they tell us is not too difficult.

Most modeling involves some queueing such as happens when all the tellers are busy in the bank, all the pumps in a filling station are being used, or all the checkout counters at the grocery store are in use.

Consider a bank with customers arriving and tellers giving service. All the possible events that take place in the bank are discrete events or can be considered as being such. Events include customers arriving, customers joining a queue if all the tellers are busy, customers going to a teller who is free, and customers leaving the bank when finished. Perhaps some of the customers will leave the queue if the wait is too long, go to 1 or more other destinations, and return later. GPSS/H is excellent for simulating systems that have this type of queueing. As we shall see, it is very easy to model a great variety of very complicated systems by using GPSS/H.

WHAT WILL BE MODELED

The models we shall be studying might represent the bank working over a period of many months, an assembly plant that manufactures television sets, a barbershop where customers can obtains haircuts, shampoos, and manicures, or even a person doing his or her Saturday morning shopping. In some cases, the model may be only a small part of a large system, such as the tool crib in a large factory.

Simulation models will not "solve" any problem directly but provide information about how the system is working and then how it will work with certain selected parameters changed. Suppose a company has its own fleet of cars for its salespersons to use. If the cars need any service, whether of a routine nature or major repairs, this is done by 1 of 2 mechanics. The company is concerned that the mechanics are not able to keep up with the repairs and wonders whether it would be worthwhile to hire another mechanic. Before the simulation model can be constructed, the company must define the problem to be solved in greater detail than has been given here. The following information is also needed:

1. The company needs complete records of all service for each type of car. This includes the frequency of service and the distribution of times for the particular service.
2. The company needs to know what it would cost to hire a third mechanic as well as the cost in lost sales when a car is not available.
3. Cars that require routine maintenance or very minor repairs are given preference for service over those that are in for major repairs. This practice means that the cars needing less time-consuming repairs are put in the front of the queue of cars that are waiting for service.
4. When the sales manager brings his or her car in for service, it is given special status; this car is immediately worked on. Thus, even if both mechanics are busy, one will abandon the car being worked on and start the repairs on the manager's car.

The information obtained from the simulation model would include the following:

1. The model will show how the system currently works. Obviously, the computer model has to accurately reflect the system as it is working before any reliability can be associated with the results from the changed model.
2. The model will show how the system works with selected changes, such as the car repair facility with 3 mechanics.

Modeling Ways to Increase Profit

Consider a system consisting of a 2-person barbershop with customers arriving to have their hair cut. The shop operates 8 hours per day. There are 2 chairs for haircuts and 4 chairs for the customers to wait in if both barbers are busy. Thus, the system only can hold 6 customers. If a 7th customer arrives and finds the shop full, he always will leave. Both barbers are identical so it can be assumed that they work at the same speed and the customers have no preference for either barber. They are served on a first-come, first-served basis. However, customers do not like to be kept waiting too long, and if a customer finds he has been waiting too long, he will leave the barbershop. The barbers know this behavior; therefore, the time required to give haircuts is a function of how many people are waiting, i.e., as more people are waiting, the 2 barbers will give haircuts faster. The customers do not arrive at regular intervals, and the haircuts are not given at the same time from one day to the next. Both arrival rates and haircut rates have known statistical distributions. Figure 4.1 illustrates the situation of the barbershop.

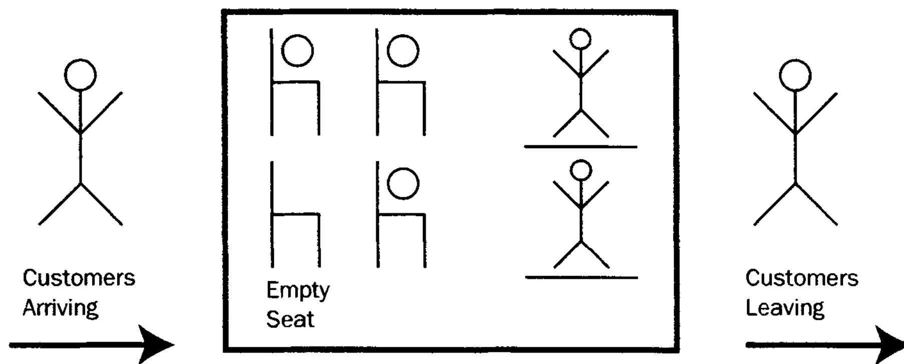


FIGURE 4.1 Two barbers and four chairs for waiting customers. Both barbers are busy, and three of the chairs are occupied.

The owner of the barbershop would like to study the shop to see if it would be profitable to add another barber or simply add another chair for customers to wait. Perhaps it would be possible to purchase new equipment so that the barbers can work even faster. Would the extra haircuts justify the expense of this equipment? GPSS/H will assist in building a model to determine the most profitable step to take. The model can make 2 kinds of predictions:

1. The model will be only as good as the input data and the assumptions given above. The model is verified by checking to see whether it predicts that the current system works as outlined above.
2. Once the model accurately represents the barbershop as it currently works, the model can be modified to predict how the barbershop will work under different conditions.

Item 2 is where GPSS/H is so handy. As will be shown, changes in GPSS/H programs often are made by changing only a few lines of code. *The fact that*

GPSS/H programs can be so easily changed to answer "What if?" type questions makes it an ideal language to use for simulation studies.

Once the modeler is satisfied that the original model is correct, the simulation can be changed, but this time with the system having 3 barbers. Alternatively, the model can be run for 5 seats for waiting customers. Finally, the model can be run for different combinations of speeds for the barbers to cut hair.

By using the cost data for the various combinations of barbers, lost customers, profit per haircut, etc., the modeler then can determine the economics of the system and make the correct choice.

A SIMPLE SIMULATION MODEL

The following example will illustrate the situation of a simulation model with constant arrival rates and constant service rates. Suppose a tool crib has 1 attendant to serve a large group of machinists. These machinists come for 1 tool at a uniform rate of 1 machinist every 5 minutes. It takes exactly 6 minutes to obtain the tool. Machinists earn $8 per hour, and the tool-crib attendant earns $6 per hour. The factory works an 8-hour shift but stops for a 1-hour lunch break. The crib is closed for lunch and at the end of the 8-hour shift. In order to simplify the calculations, if a machinist is waiting for a tool either at the lunch break or the end of the day, he or she will wait and be served. The tool-crib operator does not receive extra pay for working overtime. Should the company hire another tool-crib attendant?

Solution

This is a very simple situation and one that would rarely be encountered in practice. Even so, it will prove instructive to learn what simulation models can tell us. The problem will be solved first for 1 tool-crib attendant, then 2, and then 3.

The machinists arrive every 5 minutes, so there will be 12 per hour arriving. In a 4-hour time period, 48 will arrive. The first arrives 5 minutes after the tool crib opens and does not experience any wait. The second person arrives 5 minutes later and experiences a 1-minute wait until the attendant is free. Similarly, the third person has a 2-minute wait, etc., up to the 48th person, who has a 47-minute wait. In a 4-hour period there will, thus, be a total waiting time for the machinists of $1 + 2 + 3 + \ldots + 47$ or 1128 minutes at $8 per hour. This represents a loss of $(1128/60) \times \$8$ or $150.40. For the two 4-hour periods in a day, this represents a loss of $300.80. If 2 (or more) tool-crib attendants are working, there never will be a wait for a free attendant (only the 6-minute wait for the tool). The following table summarizes these results as well as the case of 3 attendants.

	Number of Attendants		
	1	2	3
Number of machinists who arrive	48	48	48
Total time (minutes) waiting for free attendant	1128	0	0
Cost of lost time per 4 hours	$150.40	0	0
Pay to tool-crib attendant per 4 hours	$24.00	$48.00	$72.00
Total cost per 8 hours	$324.80	$96.00	$144.00

Clearly, the result of this simple model indicates that it would be advantageous to hire 1 additional attendant. Nevertheless, the result may or may not be useful to a company, depending on the accuracy of the originally stated conditions. These may need to be reevaluated:

1. An arrival rate of 1 machinist every 5 minutes was specified. In practice, the arrival rate will be random. There may be an average rate of so many machinists per hour, but, in general, the arrival time for a particular machinist will be random.
2. A constant service rate of the tool-crib attendants was specified. Here, too, in practice, the service rate normally will be random.
3. The tool crib closed every 4 hours, and anyone waiting for service when the 4 hours was up immediately left. In practice, the machinists about to be served still will obtain service.
4. When there is one attendant, the length of the queue tended to grow to an eventual size of 8 every 4 hours. This is not realistic. If the queue is too long, an arriving machinist will tend to leave and come back later when the line is shorter.
5. Each arriving machinist wanted only 1 tool. In practice, the number of tools needed may be 2, 3, or more.

It will be shown that, through the use of the GPSS/H language, a model easily can be constructed to include all of the above possible changes to the original assumptions. Problems, such as the one involving the optimal number of tool-crib attendants, are solved quickly and easily with GPSS/H.

REVIEW OF QUEUEING THEORY

The example of the queueing problem for the tool crib has an exact solution. There are very few such solutions available, especially for problems involving cyclic queues for a finite population. *Cyclic queues* are those in which the system under study has elements that do not leave, such as trucks working in a quarry. Here the trucks are loaded, haul, dump, and return to the loader.

Whenever there is a finite population, as soon as one element is doing a particular thing, the statistical distribution governing rates will change. Thus, if a

company has a fleet of 10 trucks to be studied, if 2 are being serviced, the probability of another coming for service is no longer the same as when all 10 were up and running. In general, in order to study complex systems in which queueing takes place, it is necessary to build computer simulation models. First, however, it might be instructive to review a few basic concepts from queueing theory. These have to do with the possible arrival distribution, service distributions, number of servers (and whether they operate in "series" or in "parallel"), the population size, and the queue discipline. There are 5 possibilities to consider:

1. Population—finite or infinite
2. Arrival time distribution—any statistical distribution including one that changes during the simulation
3. Service time distribution—any statistical distribution including one that changes during the simulation
4. Service facilities—single; multiple; equal or unequal service rates, for example
5. Queue discipline—first in, first out; last in, first out; priority placement in the queue; random placement, etc.

It may come as a surprise to the person who has not formally studied queueing theory, but it is not possible to obtain exact solutions to all of the situations described in this chapter (although a lot of very fine mathematicians have tried). However, several problems do have solutions, and these can be found in textbooks on operations research or queueing theory. As one learns how to construct simulation models, it is instructive to compare the results from the simulation model with what one expects to obtain from an exact solution.

SIMULATION VS. MATHEMATICAL SOLUTION

To illustrate a comparison of a simulation model with one that has an exact solution, consider the case of a convenience store where customers arrive on the average of 24 per hour. The arrivals follow the Poisson statistical distribution. The single clerk in the store can handle 1 customer on the average of every 2 minutes. The distribution for this service is exponential. Service is first-come, first-served. The customers do not mind waiting if there is a queue. It is desired to simulate the store for 50 days or 10 weeks of operation, where a day is 8 hours in duration and the store operates continuously. Compare the results of a simulation with those obtained by an exact solution.

Solution

The problem will be recognized as a standard one that is discussed in any text on queueing theory (e.g., Phillips et al. 1976). The exact mathematical solution is available, and equations can be found for determining the probability of the clerk being idle; the probability of any number of customers being in the store; the expected number of customers in the store; the average time for a customer to wait in the queue, to be in the store, etc.

Even though an exact solution exists to this problem, a computer program was written in GPSS/H to illustrate the way the language is used to solve such problems. The simulation model used the Monte Carlo technique, which employs a random-number generator to simulate both arrival times and service times.

First, a basic time unit needs to be selected. This normally is taken as the smallest time given by the statement of the problem. For the example here, a basic time unit of 1 minute is selected. Thus, the customers will arrive on the average of every 2.5 time units. The clerk can handle a customer every 2 time units. The simulation then is done for times of 8, 50, 100, 200, 400, etc., hours. These times have to be converted to minutes since the basic time unit is a minute.

The simulation starts at simulated time $t = 0$ and runs until the program reaches a point in simulated time that the programmer thinks will represent enough real time to yield correct results. Since the exact solution assumes steady-state conditions from beginning to end of the simulation, it is run for 4 simulated hours (240 time units) and then stopped. All relevant statistics, except for the number of customers in the convenience store, are discarded. Then, the simulation is restarted and run for the desired simulated time. The following lists a selected portion of the output from this simulation program—that for the simulated time of 400 hours:

Customers served	9605
Percent time clerk is busy	80.1%
Average number of customers in store	4.122
Average customer time in store	10.2 minutes

It should be noted that GPSS/H generally outputs proportions as .801 by default. However, as done here, it is possible to customize the output to show the values as percentages. For purposes of clarity in this book, all proportional input and output values are shown as percentages or, less commonly, in per mil. The following theoretical values can be found by use of simple formulas given in any book on operations research or queueing theory:

Customers served	9600	(customers arrive on an average of 24 per hour for 400 hours)
Percent time clerk is busy	80.0% (this is 24/30)	
Average number of customers in store	4.00	
Average customer time in store	10 minutes	

As can be seen, the results calculated from theory compare quite favorably with those obtained by the simulation. It is important, in both interpreting and using the results of a simulation, that the simulation has been allowed to

run long enough in terms of simulated time for the results to be accurate. In performing a simulation, one would like to obtain results that can be reproduced nearly identically if additional runs of the simulation are done with the same input data (e.g., number of clerks) but with different random numbers to govern the timing of events. There is no set answer to the question of how long a simulated time period is enough, as the proper number of time units to simulate for is a function of several variables. Another question without a set answer is how many runs of the simulation are needed. One variable is the nature of the simulation, i.e., is the population infinite or finite? In the case just considered of an infinite population of customers, a Poisson distribution of their arrivals, and an exponential distribution of the time to serve them, a large number of runs have to be performed. In the case of a system in which the parameters being simulated cycle through the system (such as workers in a factory), not quite so many runs may be needed. The nature of the queue and the service facilities are also important. In addition, if the statistical distributions are relatively uniform, such as a normal distribution with a small standard deviation, the simulation runs tend to achieve a level of stability rapidly. This last result is important (and comforting) for the person doing simulations who has a lot of data that is normally distributed. This is often the case for working times in a factory, truck haulage rates along a road, manufacturing times, etc. If the statistical distributions are nonsymmetrical to a large extent, the number of runs required can be great, as demonstrated by means of an example later in this chapter.

Returning to the convenience store example allows a comparison of the effect of varying the simulated time. Suppose that the simulation was done for periods of less than 400 hours. What would the results have been? The answer depends, in part, on the sequence of random numbers, but it is instructive to redo the simulation for less than 400 hours and examine the results. The following table summarizes the results from these different simulated times:

Simulated Time (hours)	Customers Served per Hour	Amount of Time Clerk Is Busy (%)	Average Number of Customers in Store	Average Customer Time in Store (minutes)
8	26.9	92.2	4.34	9.58
50	23.6	81.7	4.105	7.87
100	24.0	82.6	4.061	10.63
200	23.9	79.7	4.810	11.52
400	24.2	80.3	4.122	10.2
Theoretical Values	24.0	80.0	4.00	10

As can be seen, the results of simulating for 8 hours are quite different from the theoretical ones already presented and repeated here. Simulating for 200 hours yields results that are becoming close to the theoretical ones, except for the average number of customers in the store. At 400 hours, the simulated results are quite close to the theoretically expected ones. It should be noted that, if this problem was for a real store, the simulation well may have been run for an even longer simulated time.

SIMULATION WITH NONSYMMETRICAL VS. SYMMETRICAL DISTRIBUTIONS

In mining, it is common to find that most statistical distributions are symmetrical. Nevertheless, whenever the modeler is presented with raw data, he or she always should plot the data set and determine whether it is represented by one of the more common distributions such as the normal, uniform, triangular, Poisson, etc. Nonsymmetrically distributed data in mining simulations often can be represented by the Poisson (or exponential) distribution. In any case, GPSS/H easily allows the modeler to sample from the *exact* statistical distribution.

Whenever the statistical distributions that represent the discrete events are nonsymmetrical, the number of simulated-time units for which a simulation is run to achieve an acceptable result may have to be very large. This requirement is easy to understand, since it is desired that the system be modeled over every possible situation (i.e., combination of discrete events). The goal is, in theory, to run the simulation over enough simulated time units, and repeat it with different random numbers, until there is no variation in the results of the simulation runs.

To illustrate the concept of the large number of simulated-time units required to satisfactorily model a situation in which an event has a nonsymmetrical distribution, consider the following simple example involving the chances of winning at roulette.

Suppose a man is modeling his behavior on a day-to-day basis, weekends not included. Each day, this man stops at the local casino and bets $2 on number 7 on a roulette wheel. He makes only this single bet; then, whether he wins or loses, he leaves. The probability of winning is 1/38 (the wheel has numbers running from 1 to 36 as well as a zero 0 and a double zero 00). (Incidentally, a roulette wheel is physically designed so that a player's chances of winning are less than the chances of losing. The wins and losses form a classic case of a nonsymmetrical distribution. If the player's chances of winning and not winning were symmetrically distributed, i.e., equally distributed on both sides of the mean, then the casino would not make money and would go out of business.)

How many simulated days are needed for a simulation of roulette wins and losses to produce satisfactory results (i.e., reasonably in concert with either real or theoretical results)? Certainly not 38, as the expected number of wins is only 1. How about 380 or 3,800? To answer this question, a short GPSS/H program was written. Twelve simulations were performed; they involved four different quantities of simulated-time units (380, 3,800, 38,000 and, finally, 380,000 days; the simulated-time unit is 1 day) and three different sequences of random numbers. The following table summarizes the results of these 12 simulations.

Note that the error decreases when the simulated (i.e., the observed) result approaches the expected result. As can be seen, the observed number of wins and the theoretically expected number of wins start to approach each other

Number of Days	Runs 1-4 Using First Set of Random Numbers, Observed wins	Runs 1-4 Observed error	Runs 5-8 Using Second Set of Random Numbers, Observed wins	Runs 5-8 Observed error	Runs 9-12 Using Third Set of Random Numbers, Observed wins	Runs 9-12 Observed error	Expected Wins
380	8	-20.0%	13	+30.0%	5	-50.0%	10
3,800	95	-5.0%	95	-5.0%	95	-5.0%	100
38,000	1,019	+1.9%	1,019	+1.9%	1,017	+1.7%	1,000
380,000	10,024	+0.2%	10,029	+0.2%	10,021	+0.21%	10,000

only after a long simulation time, i.e., a large number of simulated-time units. In fact, the results for 380 simulated days vary from the theoretically expected number of wins by as much as 50%. Thus, for situations involving a nonsymmetrical distribution of events, one always must be aware that a simulation may require a very large number of simulated-time units to yield a model that approaches what is expected or real. There's no guarantee that even 380,000 is enough, depending on the problem!

In the roulette example given here, when the number of simulated days is increased to 380,000, the expected and observed wins differ by only 0.2% for all three runs with different random numbers—certainly, a satisfactory result. This near-coincidence in the results of the two methods (theory and simulation) indicates that the simulation represents both precision (the simulation results are nearly identical when the simulation program is repeatedly run with different random numbers) and accuracy (the simulation results closely model the system).

It may be interesting to note here that the outcome of all three runs for 380,000 simulated days shows a net loss of $57,336 because the casino pays out at a rate of 36 times the bet whenever the number 7 comes up. It is only for one of the runs of 380 days that a gain can be found (second set of random numbers in the table). Although the outcome will be slightly different for each new simulation, one is soon convinced that, over time, the casino will always come out ahead. There is an old saying in economics, "We can all be rich if only we could live long enough." This adage was not intended for a person who frequents casinos.

In models of mining operations, once precision is attained by running the program with different random numbers, the question of accuracy remains. In general, if the simulated results and the actual mining results are within 2%, the simulation model is considered as "acceptable." Often the data set used in developing the simulation model is the main source of error. The mining engineer must be constantly aware that accurate data from the mine are always required. The statistical distributions of the various mining data need to be continually refined and updated. In modeling mining operations, even if there is no change in the equipment used, the distances in the mine change over time. Thus, the values of some of the parameters used in the simulation will need to be changed. The most obvious of these are the travel or haulage distances. Most of the distributions that are used to represent these parameters—such as the travel from the loader to the dump or crusher or travel from

the mine to the repair shop—are represented by statistical distributions that are symmetrical. Thus, although these distributions need to be changed in the simulation with time, the number of simulated-time units need not be increased. As equipment ages, the failure distributions change. Since the number of failures and, thus, the probability of failure tend to increase over time, the distributions representing this fact often become more symmetrical, and so a shorter simulated time may be needed. Fortunately, for most mining situations, the simulation can be successfully performed with a reasonable number of simulated-time units. Also, fortunately, the GPSS/H language allows for easy and rapid modification of the various statistical distributions used in the models.

WHY DO A SIMULATION?

As has been stated, it is rarely possible to obtain exact solutions to any but the most elementary problems that lead to queueing situations. We all understand queueing situations as we experience them daily whenever we enter a bank that has many customers and we have to wait for a teller, whenever we shop in a large grocery store and wait in the checkout line, etc. One would think that problems involving such situations could be solved quite easily, but this is not the case. Although the field of queueing theory has been studied by the mathematicians for many years, very few problems have been solved through the use of theoretically based calculations. A computer simulation, on the other hand, can rapidly and accurately predict the outcome of most elementary and complex problems in which one encounters queueing situations.

WHAT IS MEANT BY A "SOLUTION TO A QUEUEING PROBLEM"?

When we use simulation to obtain a solution to a queueing problem, it is important to understand just what is meant by our solution. A simulation model does *not* solve a problem but tells us *how a system will operate under a given set of parameters.* For example, the model might tell you that, if you have 9 trucks in your mine, the daily profit will be $457. Adding a 10th truck and *doing another simulation* might then tell you that the new profit will be $505. Doing a further simulation for 11 trucks might tell you that the new profit will be $480. Thus, we conclude that the optimum number of trucks to have in the mine is 10.

ANOTHER EXAMPLE OF A SIMULATION MODEL

Consider a simple example of 1 shovel loading trucks at a construction site. This shovel can load only 1 truck at a time. After each truck is loaded, it travels to a dump area where it dumps its load and then returns to the shovel. If the shovel is free (no other truck is being loaded), the truck immediately begins to be loaded again. If not, it waits in a queue until the shovel is free.

Assume that you are going to study this system. The trucks and shovel and the travel paths make up the system. You are told by the engineer in charge, who has studied the shovel and the haulage routes, that the shovel can load a truck

in *exactly* 5 minutes (this is a slow shovel, but don't worry about this fact for the present). It takes *exactly* 8 minutes to drive to the dump, *exactly* 2 minutes to dump, and *exactly* 6 minutes for a truck to return to the shovel.

If you had a single truck in the system, it would be loaded in 5 minutes and then take 16 minutes to return to the shovel. The shovel would be busy for every 5 minutes out of 21 or 23.8% of the time. There would be a load of ore dumped every 21 minutes for a load dump rate of approximately 3 per hour. In an 8-hour shift, you would expect that there would be slightly less than 24 loads dumped (this construction site does not allow the workers any breaks).

If you added another truck to the system, you would expect the production to double and the shovel to be twice as busy. Adding a 3rd truck would likewise increase production, and the shovel would be busy about 72% of the time. What will happen when you add a 4th truck? The answer is that the system still will experience no queueing problems as, at any one time, there can be a truck being loaded (this takes 5 minutes) and the other 3 can be traveling to or from the dump. It is only when you have 5 trucks working that you start to have queueing problems. However, production will increase by having 5 trucks as compared to having 4. The question is how much and will this increase be worth it? To answer this last question, additional data are needed. In particular, each load carried by the trucks somehow represents a contribution to profit of $45, and each truck costs $225 per day to run.

For the above problem, a computer simulation was run for 10 simulated days with each day being an 8-hour shift. At the start of the simulation, all of the trucks were at the shovel, and the trucks worked for 10 days straight. Average event times were used. The results of the simulation are as follows:

Number of Trucks	Number of Loads	Shovel Utilization (%)	Average Number of Trucks in Queue	Profit Per Day ($)
1	229	23.9	0.000	805
2	458	47.7	0.000	1,611
3	686	71.5	0.003	2,412
4	914	95.2	0.006	3,213
5	959	100	0.807	3,190
6	959	100	1.807	2,965

It is easy to see that the optimum number of trucks to have for maximum profit is 4. Adding a 5th truck will increase production but will not result in a greater profit.

A CHANGE TO THE PROBLEM

The problem just completed assumed that all of the times used in the model were constant. This is certainly not the case in real life. Things do not happen in exact times. The time to load a truck will vary depending on several parameters, the time to drive to the dump will not be constant, etc. If you did some time studies at the construction site, you might find that the time to load a truck *averaged* 5 minutes but that the statistical distribution that best describes loading time is the exponential distribution. You also might find that the travel times and the dumping times are best described as following normal distributions:

	Mean (minutes)	Standard Deviation (minutes)
Travel to dump	8	1.5
Dump	2	0.3
Return to shovel	6	1.15
Total	16	

As can be seen, the mean times (16 + 5 = 21 minutes per load) have remained the same.

The computer program was modified to allow for statistically based rather than average event times and run for 100 simulated days because the exponential distribution was used. (Whenever this distribution is used, the number of simulations or the simulated time required for an acceptable result is substantially increased–more on this in succeeding chapters.) The following table gives the results of the simulations.

Number of Trucks	Number of Loads	Shovel Utilization (%)	Average Number of Trucks in Queue	Profit Per Day ($)
1	2,281	24.1	0.00	801
2	4,355	44.5	0.10	1,509
3	6,056	62.7	0.35	2,050
4	7,515	76.8	0.73	2,481
5	8,285	87.6	1.36	2,603
6	8,948	93.6	2.08	2,676
7	9,243	97.4	2.94	2,584
8	9,434	98.9	3.87	2,405
9	9,423	99.7	4.84	2,215
10	9,547	99.9	5.82	2,046
11	9,550	100	6.82	1,820
12	9,523	100	7.82	1,585

Notice that the number of loads keeps increasing as the number of trucks is increased until 11 trucks are working. Also note that the daily profit is a maximum for 6 trucks. This profit is $2,676, which is considerably less than the previous profit of $3,213. Thus, in the more realistic simulation, the number of trucks needed is 50% more than before, and the optimum profit is 21% less. Also note that it really would not make much difference if 5 trucks were used rather than 6. This simulation shows what happens when too many trucks are in the system. The average queue length for 9 trucks is 4.84 trucks, and for each additional truck, the average number of trucks waiting in the queue will increase by 1. This means that each additional truck will, in effect, add to the average queue length. No increase in production will result.

Although this example seems simple, it and variations of it were studied by numerous investigators in the 1960s and 1970s to determine the optimum number of trucks to have for construction projects. For example, Griffis (1968) studied the determination of the optimum fleet size using queueing theory. Numerous lengthy reports and papers were written on this problem alone.

REFERENCES

Griffis, F.H. 1968. Optimizing Haul Fleet Size Using Queueing Theory. *Journal of the Construction Division*, Proceedings of the American Society of Civil Engineers, January: 75–87.

Phillips, D.T., Ravindran, A., and Solberg, J.J. 1976. *Operations Research.* New York: John Wiley and Sons, chapter 7.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

PART II

The GPSS/H Simulation Language

CHAPTER 5

Sample GPSS/H Programs

RUNNING A GPSS/H PROGRAM

The best way to learn GPSS/H is to run numerous programs each time you are introduced to a new topic. A GPSS/H program will have many different commands, so it will not be until a few more topics are introduced that you will be able to write programs by yourself. However, it is possible to type up the programs and run them right away. For the present, do not worry about what all the different commands are—each will be explained later.

WHAT YOU WILL NEED

You will need a modern PC that has GPSS/H loaded. You must know how to create and edit ASCII files. Files can be created with the DOS editor (probably easiest), or with word processing software such as WordPerfect or Microsoft Word. The creation and editing of files is not covered here.

A FIRST TASTE OF GPSS/H

The first problem we are going to simulate with the GPSS/H language involves a barbershop. People enter the shop every 10 minutes. It takes the barber *exactly* 13 minutes to give a haircut. Obviously, this situation is unrealistic and will soon lead to the barbershop's being overloaded with customers, but the problem will introduce us to what the GPSS/H language and the program output look like. The problem is to simulate the shop for 1 hour, starting at $t = 0$ when there are no customers in the shop. Figure 5.1 illustrates the time scale with customers arriving and leaving for 60 minutes.

Figure 5.1 shows that the barber will have nothing to do until $t = 10$ when the first customer arrives. The double lines at $t = 10, 20, ..., 60$ represent the customers arriving at the shop. The single lines without the circles, starting at $t = 23$, represent customers leaving. It should be easy for us to understand this system and be able to explain what is happening.

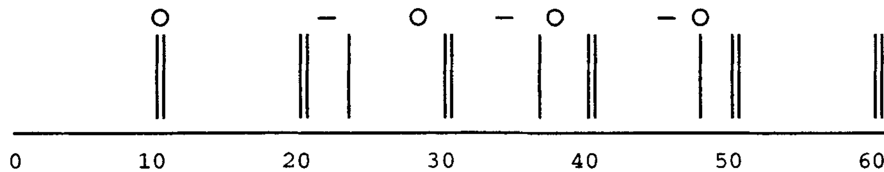


FIGURE 5.1 Representation of what is happening in the barbershop. See text for explanation.

Summary of GPSS/H Input Format

Before we can write the program to simulate the activity in the barbershop, however, we need to understand a bit about GPSS/H code. The GPSS/H commands are either blocks or statements (we will learn which as each is introduced). Each line has only one such command.

In the explanations of GPSS/H code in this book, lowercase letters are used to indicate a generic form of the code. Optional components are enclosed in parentheses. Italic type is used to show variables that may take on values, such as *n*. The positions of the GPSS/H code components (shown by the numbered line; the numbers in squares indicate positions that are usually blank in most lines of code) and the general form of a GPSS/H program block are

```
1234567890123456789012345678901234567
◇label    □operation □operand □comment
```

For clarity, in this book, position 1 of input code is shown by an open diamond (◇) to remind the reader that, in the fixed format used here, this position is almost always blank. Other positions that are almost always blank are position numbers 10 and 21. Also the operation is always separated from any auxiliary operator by one space, and any comment is preceded by at least one space (this book precedes comments with the required space plus an optional one). The required spaces are shown by open squares (□). *The diamond and squares are not to be typed in actual GPSS/H code.* The history of GPSS/H formatting is described in Appendix C.

As shown in Table 5.1, actual labels in lines of code for blocks consist of only capital letters or of capital letter(s) followed by number(s). Actual operations consist of only capital letters. Actual labels for a statement can be a number or letter(s) plus number(s), but must at least start with a number. Actual operands can be numbers or capital letters or numbers and capital letters. Comments can include any characters, and the letters can be lowercase. More details are given in the section in this chapter entitled Fixed Format, and code to allow printing out symbols to improve the readability of output is discussed in Chapter 8.

The simplest line of code consists of a single operation and takes the form of

```
◇         □operation
```

TABLE 5.1 Summary of GPSS/H input format

Component	Positions	Possible Forms of GPSS/H Code Components: In Blocks	In Statements
LABEL*†	2–9	only capital letters	only capital letters
		capital letters followed by numbers	capital letters followed by numbers
			only number(s)
		In Blocks and Statements	
OPERATION	11–20	only capital letters	
OPERANDS†	22 until finished	numbers	
		capital letters	
		numbers followed by capital letters	
		capital letters followed by numbers	
		certain nonalphanumeric ASCII characters are also used in some operands to indicate variable type and mathematical operations	
Comments	begin 2 spaces after operands	any characters including lowercase letters	

Note: Positions 1, 10, and 21 are generally blank. See text for more information.

* Because GPSS/H has many reserved words that are normally from 1 to 3 characters in length with a few as long as 4 characters in length, it is good programming practice to have labels 4 or more characters in length. Furthermore, labels are only used for "cross-referencing" lines of code. Thus labels are not normally used for blocks and are needed in only certain statements.

† The maximum number of characters is 8.

Examples of this simplest form of a line of GPSS/H code are

```
◇        ▯SIMULATE
```

or

```
◇        ▯END
```

Most operations require at least 1 operand:

```
◇        ▯operation ▯operand
```

An example of this type of operation is

```
◇        ▯GENERATE  ▯10
```

where the operand is 10. The operand can *start* in position 22 to position 25 (though the operand can also *start* farther right if the OPERCOL statement is used, as discussed in the second half of this chapter). Certain operations require multiple operands. Comments can follow the operand(s) for showing the flow of elements through the program. Code lines may need to be cross-referenced, which is made possible by using a label:

```
◇label   ▯operation ▯operand1,operand2,operandn ▯comment
```

The term "operand*n*" indicates the *n*th operand. An example of a code line with two operands and a comment is

```
◇PARTS   ▯FUNCTION  ▯RN1,D3 ▯Parts come along assembly line.
```

The label is not included in all code lines. If a label is included but is not referred to by another line of code, a warning message will appear on the screen; however, the program will not be terminated. In the following form of GPSS/H code, no label is needed, but the operation (opn) requires an auxiliary or relational operator (opr), and these must be separated by one space, for example,

```
◇        □opn□opr    □operand1,operand2 □comment
         □TEST□E     □N(NDONE),N(INSHOP) □Is shop empty?
```

Auxiliary or relational operators are described beginning in Chapter 16.

It is important to remember that *blanks (i.e., spaces) are not permitted in labels, operations, or operands.* Thus, if the block is to be

```
◇        □GENERATE  □,,,4    □THIS A GENERATE BLOCK
```

(note that there are no spaces after the commas), it would be incorrect to have

```
◇        □GENERATE  □, , , 4 □THIS IS A GENERATE BLOCK
```

Although it is not necessary to follow the format given below for this exercise (because of the changes in GPSS/H discussed in Appendix C and under the heading Free Format in this chapter), try to type the program exactly as given here for ease of debugging. Imagine that you are typing the program on a numbered grid and each column has a position number as shown on page 36.

First GPSS/H Program

In the following program to simulate the barbershop, there are no labels; therefore, the blocks or statements will begin in position 11. In some cases, there will be associated operands, and these will begin in position 22. Also, *no lowercase letters are allowed in GPSS/H program lines.* This restriction does not apply to comments. No comments appear in the following program, however.

```
◇        □SIMULATE
◇        □GENERATE  □10
◇        □QUEUE     □WAITAREA
◇        □SEIZE     □BARBER
◇        □DEPART    □WAITAREA
◇        □ADVANCE   □13
◇        □RELEASE   □BARBER
◇        □TERMINATE
◇        □GENERATE  □60
◇        □TERMINATE □1
◇        □START     □1
◇        □END
```

If you have never seen a GPSS/H program, this code must look strange, but, like any programming language, it will become familiar to you with practice. Notice that there are no commands that correspond to input or output such as READ or WRITE. There is no READ because you normally do not read data

into a simulation program. There is no WRITE, so what about output? Here is where GPSS/H is so helpful. Whenever certain blocks appear in the program, there will *automatically* be output associated with those blocks. (Full output is in FILENAME.LIS; custom output is usually more useful [see Chapter 8].) If you are studying a queueing situation (as happens in our barbershop simulation), output will automatically be produced.

The filename of the program you wrote *must* have the extension .GPS for the program to run under GPSS/H. Suppose the name of it is BARBER.GPS. To run it, you need to be at a DOS prompt in the GPSS/H directory. Then type

```
GPSSH BARBER NOXREF NODICT  <cr>
```

Although the extension .GPS must be part of the filename BARBER.GPS, the extension need not be typed in the command line; <cr> means "enter" (actually, "carriage return" from the days of the trusty typewriter). The commands NOXREF and NODICT are optional. They represent "no cross reference" and "no dictionary." If they are omitted, there will be considerable output if the program has an error. Generally, you do not need all the output.

First GPSS/H Output

If your program was written with no errors, you will see a screen such as the following (in this book, "output" is shown in italics):

```
GPSS/H RELEASE 3.0j-C10          12 July 1999          14:15:57
File:    barber.gps
Compilation begins.

Pass 1 (with source listing)...

Pass 2...

Simulation begins.

GPSS/H IS A PROPRIETARY PRODUCT OF,
AND IS USED UNDER A LICENSE GRANTED BY,
WOLVERINE SOFTWARE CORPORATION
7617 LITTLE RIVER TURNPIKE
ANNANDALE, VIRGINIA 22003-2603, USA

C:\GPSSH>
```

The return of the DOS prompt means that the program successfully ran. At the completion of the program, GPSS/H creates a list file that has the same name as the original file but now with the extension .LIS. To view the list file, you can either use the same text editor you used to create the .GPS file or simply type

```
TYPE  BARBER.LIS | MORE
```

The output will look, in part, as follows, depending on the version of GPSS/H that you are using:

```
    STUDENT GPSS/H RELEASE 3.0j-C10 (UL206)    12 July 1999    16:43:42     FILE: barber.gps
    LINE# STMT#  IF DO  BLOCK#  *LOC      OPERATION      A,B,C,D,E,F,G   COMMENTS

        1     1                           SIMULATE
        2     2              1            GENERATE   10
        3     3              2            QUEUE      WAITAREA
        4     4              3            SEIZE      BARBER
        5     5              4            DEPART     WAITAREA
        6     6              5            ADVANCE    13
        7     7              6            RELEASE    BARBER
        8     8              7            TERMINATE
        9     9              8            GENERATE   60
       10    10              9            TERMINATE  1
       11    11                           START      1
       12    12                           END

STORAGE REQUIREMENTS (BYTES)

COMPILED CODE:       220
COMPILED DATA:        80
MISCELLANEOUS:         0
ENTITIES:            344
COMMON:            10000
-----------------------
TOTAL:             10644

Simulation begins.

RELATIVE CLOCK: 60.0000   ABSOLUTE CLOCK: 60.0000

BLOCK CURRENT     TOTAL
1                     5
2          1          5
3                     4
4                     4
5          1          4
6                     3
7                     3
8                     1
9                     1

          --AVG-UTIL-DURING--
FACILITY  TOTAL  AVAIL  UNAVL     ENTRIES    AVERAGE  CURRENT  PERCENT  SEIZING  PREEMPTING
           TIME   TIME   TIME               TIME/XACT  STATUS   AVAIL     XACT       XACT
  BARBER  0.833                         4      12.500   AVAIL                5

   QUEUE     MAXIMUM      AVERAGE      TOTAL       ZERO     PERCENT      AVERAGE
            CONTENTS     CONTENTS     ENTRIES     ENTRIES    ZEROS      TIME/UNIT
WAITAREA           1        0.467           5           1     20.0          5.600

     $AVERAGE     QTABLE     CURRENT
    TIME/UNIT     NUMBER    CONTENTS
        7.000                      1
```

```
STATUS OF COMMON STORAGE

    9424 BYTES AVAILABLE
     576 IN USE
     680 USED (MAX)

Simulation terminated. Absolute Clock: 60.0000

Total Block Executions: 30
Blocks / second:        30000
Microseconds / Block:  33.33

Elapsed Time Used (SEC)

PASS1:            0.05
PASS2:            0.06
LOAD/CTRL:        0.16
EXECUTION:        0.00
OUTPUT:           0.06
---------------------
TOTAL:            0.33
```

Let us now examine your output. It is going to look strange at first, but you soon will become accustomed to interpreting the results. At present, we will simply ignore most of it. (Note, however, for future reference that all the lines that are blocks are numbered.) The first line of interest to us will be

```
RELATIVE CLOCK: 60.0000   ABSOLUTE CLOCK: 60.0000
```

This line indicates that the simulation went for 60 simulated time units. Notice that there are two imaginary clocks in GPSS/H. Both start at time $t = 0$. We shall learn how to run a simulation for a certain period, stop execution, and restart it with most, but not all, of the statistics set back to zero. When this procedure is done, the absolute clock keeps going, but the relative clock is reset to zero each time the program is restarted.

Skipping down to under MAXIMUM CONTENTS, we see that there was a maximum of 1 person waiting. The TOTAL ENTRIES indicates that 5 people entered the queue. Even the first person who entered the barbershop and went immediately to the barber's chair is counted. This person is listed under the next heading ZERO ENTRIES as being 1. Of the 5 people who entered the shop, 1 did not remain in the waiting area, so the PERCENT ZEROS is 20.0. The average time for *each* person to wait in the queue was 5.600 minutes. This is determined by noting that 5 people entered the queue. The first was there for 0 minutes, the second for 3 minutes, the third for 6 minutes, the fourth for 9 minutes, and the fifth for 10 minutes. Thus, 28 divided by 5 is 5.6. The next entry, $AVERAGE TIME/UNIT, is the total time in the queue but now divided by only those people who actually remained in the queue, namely 4. The CURRENT CONTENTS is 1, which indicates that 1 person is in the queue at the end of the simulation. Compare these results with the diagram in Figure 5.1. Note that a few other results appear in the .LIS file, but those discussed here are the main ones.

Example 5.1

Suppose that the arrival times and haircut times were reversed so that the customers arrive every 13 minutes and the barber can give a haircut in 10 minutes. The modifications necessary to do this simulation are as follows:

change the line

```
◇         □GENERATE  □10
```

to

```
◇         □GENERATE  □13
```

change the line

```
◇         □ADVANCE   □13
```

to

```
◇         □ADVANCE   □10
```

Make these changes and see if you can interpret the results.

A CLOSER LOOK AT THE GPSS/H CODE

We have just run our first program and then some of the results were interpreted. You were told to be very careful about where and how the various commands were typed. This is because GPSS was first introduced when computers exclusively used punched cards for reading the program. This method is no longer used; so the restrictions (see Appendix C) inherent with punched cards no longer need apply. There are two ways to write your program, fixed and free format, and both will be presented here. We shall learn that, in GPSS/H, there are basically two types of program commands: one is called a *statement* and the other a *block*. The various properties of each will be discussed in subsequent chapters. The forms of both are similar, and the discussion here will apply to both blocks and statements.

Fixed Format—The Details

In GPSS/H, each separate line of the program will either be a statement or a block. (In a few situations, the GPSS/H statement will be continued for 2 or more lines). The general format of a GPSS/H block consists of 4 separate items (see Table 5.1):

1. label
2. operation (may be followed by an auxiliary or relational operator; see Chapter 16)
3. operands
4. comments

The fixed-format form used in this book will always be

```
1234567890123456789012345678901234567
◇label    □operation □operand □comment
```

1. The *label* starts in position 2 and goes through position 9. *Spaces and punctuation are NOT permitted within a label.*

   There is (normally) nothing in position 1 (marked in this book by an open diamond: ◇) or 10 (marked in this book by an open square:▯). An exception is a line that in its entirety forms a comment (in some other languages, called a remark). A comment line begins with an asterisk in position 1. *Any line beginning with an asterisk is ignored;* i.e., such a line becomes a "megacomment."

   Furthermore, most blocks and many statements do not use labels; they are only used where "cross-referencing" between lines of code is needed. In fact, if a block has a label that is not referenced anywhere in the program, there will be a warning message sent to the screen to this effect.

2. The *operation* begins in position 11 and can go through position 20. *Spaces are NOT permitted within operations;* in fact, the allowable operations are predefined in GPSS/H. The main ones are covered in the rest of Part II of this book.

   Position 21 (▯) is almost always blank

3. *Operands* begin in position 22 to position 25 (or farther right; see OPERCOL statement, discussed next) and can continue as far as position 71. *Spaces are NOT permitted, even after commas that separate the multiple operands or indicate the positions of unused operands.* (Keep this prohibition in mind later when you are learning how to use arithmetic in operands, as it is tempting to leave spaces around the plus signs or minus signs.)

   The exception is that spaces are permitted in textual material that is merely intended to be written to the screen or a file; more on this topic is covered in Chapter 8. Such textual material must be enclosed in single quotes and parentheses.

   It is possible to continue an operand list to another line by putting the underscore (_) character in or to the left of position 72 in the operand field (not the operation field). The next line is read by the compiler starting with the first nonblank (i.e., nonspace) character anywhere in positions 1–19. In the following example, if the underscore is placed after the comma (but with no spaces!), the code is continued to the next line:

   ```
   ◇        ▯ADVANCE   ▯10,4
   ```

   and

   ```
   ◇        ▯ADVANCE   ▯10,_
                   4
   ```

   are identical. Once you use an underscore to continue a line, you can begin the continuation in any space up to position 25.

4. A *comment* can be placed after a blank space (▯) after the operand. In this book, comments will be preceded by the required blank and usually at least one additional blank. Most programs that have comments will generally have them all starting in the same column for ease of reading them. But be careful–not all statements have operands. *For statements that don't have operands, it is still necessary to place comments starting in position 26*

or anywhere to the right of that position. (If OPERCOL [described next] is used, comments may start in position OPERCOL + 1 or to the right of that position.)

Even though rule 3 specifies that operands are to begin in position 25 (or before), the programmer can specify where the operands are to begin by means of the OPERCOL statement. This has the form

```
◇        □OPERCOL   □n
```

where *n* is the position at which the operands can begin. By default, *n* is 25, but it can have a larger value. Thus,

```
◇        □OPERCOL   □30
```

will tell the compiler to scan for the operands up to position 30. This modification can be very useful in certain situations, such as those that have output that includes items that are nearly the same and you want them to line up underneath each other. It can also come in handy when you use nested DO loops (see Chapter 23).

The above may sound like a lot of "thou shall nots," but, in practice, the form of the GPSS/H lines of code are easy. The last item, about the OPERCOL, is rarely used if one sticks to fixed format.

The label may not be more than 8 characters in length. It is possible to mix numbers and capital letters, providing the first character is a letter (see Table 5.1). Examples of labels for *blocks* include

```
◇JOEANNE
◇BILLYBUD
◇UPTOP
◇DOWN1
◇BACK1
◇UPTOP7
◇A123
```

but not

```
◇1JOE
◇B23$K
◇-1
◇J&B
◇123
◇uptop
◇BACK UP
```

For *statements,* the following labels would be acceptable:

```
◇BACK
◇UP123
◇1
◇15
```

but not

```
◇1A
◇uptop
◇X-Y
```

To illustrate how the above works, take the program you just wrote and change it as follows.

```
***----------------------------***
*                                *
*    MY FIRST GPSS/H PROGRAM     *
*                                *
***----------------------------***
*
*
*    I WONDER HOW LONG BEFORE IT STARTS TO
*    MAKE SENSE
*
*------------------------------*
◇        □SIMULATE
*
◇COME    □GENERATE  □10   This has to do with people arriving.
◇        □QUEUE     □WAITAREA
◇        □SEIZE     □BARBER
◇        □DEPART    □WAITAREA
◇        □ADVANCE   □13        I bet this has to do with
*         getting a haircut---------------------------------x
*        The next line shows continuation to 2nd line.
◇        □RELEASE    □BA_
  RBER
*  I hope this will work - it looks strange.
*
*
*
◇        □TERMINATE
◇        □GENERATE  □60
◇        □TERMINATE □1
◇        □START     □1
◇        □END
```

Except for the way the RELEASE block was written to illustrate how a continuation to the next line can be made, the above is the form most GPSS/H programmers follow. The initial comments, preceded by asterisks in position 1, describe the program; later comments are placed in the program by inserting them after two blanks in the line after the operand. The comments in most of this book are in capital letters, but they could be in lowercase letters as they are ignored by the compiler.

Run the program as before. At the DOS prompt, type

```
GPSSH BARBER NOXREF NODICT <cr>
```

Notice that when you run the program this time, you will receive a "warning ..." because the block

```
◇COME  ▯GENERATE  ▯10
```

was never referred to. Do not be alarmed; this is just the way GPSS/H works. Once you give a block a label, GPSS/H expects you to refer to it in the main program. It looks a bit messy, but such warning messages can be a real help in debugging long programs.

There are other commands besides NOXREF and NODICT that can be used when you initiate the running of a GPSS/H program. Probably the other most common one is "TV" (for TV mode). If you run a program in this manner, the screen is split; by keying in S <cr> or S *n* <cr>, where *n* is an integer, you can step through the program. This procedure can be useful in debugging programs. However, the use of on-screen output has greatly reduced the need for using the TV mode for debugging.

Free Format

Since GPSS/H programs now are written with only the screen of a PC showing, the restrictions that apply to fixed format have been relaxed (see Appendix C). The way GPSS/H statements may now be typed is as follows:

1. Labels begin in *either* column 1 or 2. They can be up to 8 characters in length, as can operands.
2. If no label is used, the operation portion can start in column 3 (or farther right).
3. Operands start following the operation. There need be only 1 blank between the operation and the operand. There may be more blanks, in which case the operand must begin in or before the position given by the OPERCOL.
4. Comments are placed in the program just as for fixed format, e.g., following an asterisk, which may be in position 1.
5. Statements may be continued to another line as in fixed format.

Thus, it is possible to return to our first program and type it as follows:

```
*----------------------------------------
*
*   This is very free format.
*234567890123456789012345678901
*      (column numbers)
*
*----------------------------------------
      COME  GENERATE 10
         QUEUE WAITAREA
     SEIZE        BARBER
       DEPART  WAITAREA
      ADVANCE 13
      RELEASE  BARBER
         TERMINATE
       GENERATE   6_
          0
```

```
    TERMINATE
    GENERATE 60
    TERMINATE 1
        START 1
END
```

There really is no advantage in writing programs this way. The programs presented in this book are all written in fixed format for the sake of uniformity. If you choose to write your programs in free format, however, it is perfectly acceptable.

Arithmetic in Operands

It is possible and indeed common to have arithmetic in the operands. The arithmetic operations in GPSS/H are

+	addition
-	subtraction
/	division
*	multiplication
@	modular division

Modular division results in only the remainder being returned. Thus, 5@7 is 2, 9 @30 is 3, and 7@8 is 1.

GENERATE 480*10 is the same as GENERATE 4800, ADVANCE 60*100 is the same as ADVANCE 6000. GPSS/H does integer arithmetic if it is dealing with integers, so ADVANCE 60/18 would be the same as ADVANCE 3. However, ADVANCE 60.0/18 would be ADVANCE 3.33333.

Example 5.2

Even though we still have a way to go in writing programs, we can still use the one program we have written to learn a bit more about GPSS/H. From now on we will speak of the "processor" when we refer to how the program is being executed. To simulate the same shop for when the barber is working a bit faster, simply replace the line

```
        ADVANCE     13
```

with

```
        ADVANCE     12
```

Now the barber is cutting hair in 12 minutes (the shop still will overflow with people). To simulate for 2 hours, change the line

```
        START       1
```

to

```
        START       2
```

By now, you should be able to understand the results of the second simulation.

Example 5.3

To simulate the situation when the barber gives haircuts in 9 minutes, change the line

```
◇        □ADVANCE   □13
```

to

```
◇        □ADVANCE   □9
```

Now there will not be a buildup of customers. Run your program with this change for 2 hours of simulated time and interpret the results.

THE SIMULATE STATEMENT

Every GPSS/H program must contain a SIMULATE statement. Although it need not be the first statement in the program, it usually is. Most programmers always start their programs with the SIMULATE statement. The general form of it is

```
◇        □SIMULATE  □A
```

The operand *A* is optional. If used, it limits the amount of time in minutes during which the program will run. This can be handy to use in the debugging stage to avoid being stuck in infinite loops. For example,

```
◇        □SIMULATE  □2.5
```

would limit the program to 2.5 (actual, not simulated) minutes running time. It is possible to have the time limited to so many seconds if the letter S is placed after the SIMULATE operand:

```
◇        □SIMULATE  □100S
```

would limit the execution time to 100 seconds.

There is a caution with this statement. Be careful that you do *not* put a comment after SIMULATE if you have not specified an operand. If you do add a comment, it must start in or after position 26 or in or after OPERCOL + 1.

THE END STATEMENT

GPSS/H programs must have an END statement. This is simply the last statement in the program and acts as a directive to the compiler. Of all the many

blocks and statements with GPSS/H, the SIMULATE and END should never cause you any problems.

Example 5.4

Let us do the same example of the barbershop again. (Soon we shall be doing other, more interesting problems.) You were told to run the last example for 2 hours of simulated time by changing the START 1 to START 2. The following program will simulate the barbershop for 2 customers. Write the program and run it.

```
◇        SIMULATE
◇        GENERATE  10
◇        QUEUE     WAITAREA
◇        SEIZE     BARBER
◇        DEPART    WAITAREA
◇        ADVANCE   13
◇        RELEASE   BARBER
◇        TERMINATE 1
◇        START     2
◇        END
```

After you run the program, you should examine the output to see what the program results are. After a few more exercises, you will understand the differences between the programs.

EXERCISES, CHAPTER 5

1. Parts come along an assembly line with an interarrival rate of 4, 5, or 6 minutes. A single worker takes either 4 or 5 minutes to work on each part at station A. This worker can work on only one part at a time. Next, parts go to station B where two identical workers take 8, 9, or 10 minutes to do further work on the parts. Finally, an inspector takes exactly 5 minutes to check the parts.

   Do a "hand simulation" that lists the events for the first hour of operation starting at time 0. The wait time is 1 minute. For the sake of simplicity, assume that the various times by the worker are in exact order. Thus, the first worker will complete work on the first part in 4 minutes, the second part in 5 minutes, etc. Your result should show the event, the time it takes place, and a description of what is being done. The first part of your solution will look as follows:

Event	Time	Description
1	4	First part comes to station A; next part will come at time 9.
2	4	First part will be worked on until time 8.
3	8	First part done at station A; first part goes to station B.

Event	Time	Description
4	8	First part is worked on at station B until time 16.
5	9	Second part comes to station A; next part will come at time 13.

The "hand simulation" is useful to assist the interpretation of the GPSS/H simulation. The GPSS/H program to do the simulation is as follows:

```
◇         □SIMULATE
◇         □STORAGE   □S(WORKERS),2
◇PARTS    □FUNCTION  □RN1,D3       □Parts come along
.333,4/.667,5/1,6
◇WORK1    □FUNCTION  □RN1,D2       □First worker
.5,4/1,5
◇WORK2    □FUNCTION  □RN1,D3       □Second workers
.333,8/.666,9/1,10
◇         □GENERATE  □FN(PARTS)    □Parts come
◇         □QUEUE     □WAIT1        □Join first queue
◇         □SEIZE     □FIRST        □First worker is used
◇         □DEPART    □WAIT1        □Leave the queue
◇         □ADVANCE   □FN(WORK1)    □Work on first part
◇         □RELEASE   □FIRST        □Free the worker
◇         □QUEUE     □WAIT2        □Join second queue
◇         □ENTER     □WORKERS      □Use one of the workers
◇         □DEPART    □WAIT2        □Leave the queue
◇         □ADVANCE   □FN(WORK2)
◇         □LEAVE     □WORKERS      □Free the worker
◇         □QUEUE     □WAIT3        □Join last queue
◇         □SEIZE     □LAST         □Use the worker
◇         □DEPART    □WAIT3        □Leave the queue
◇         □ADVANCE   □5            □Inspect part
◇         □RELEASE   □LAST         □Free the inspector
◇         □TERMINATE               □Part done
◇         □GENERATE  □60           □Timer transaction arrives
◇         □TERMINATE □1            □Simulation over
◇         □START     □1
◇         □END
```

Run the above program and interpret the results (as much as possible).

2. In order to run the above program for a simulated time of 8 hours (480 minutes), it is necessary to change the line of code

```
◇         □GENERATE 60
```

to

```
◇         □GENERATE 480
```

Make this change and rerun the program.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 6

The GENERATE and TERMINATE Blocks and the START Statement

BLOCKS VS. STATEMENTS

Control statements are used for specifications of data and for running the program. Blocks are the actual program lines. Transactions, which are explained next, move only from block to block once the program execution begins.

THE GENERATE BLOCK

The key to any simulation program is the GENERATE block. This block creates "transactions" that move through the program from block to block. These transactions can represent anything that you wish to model, such as the entering of ships into a harbor, the arrival of people at their place of work, cars moving on a highway, and a person entering a bank.

To give an idea of how a transaction is used, think of a person about to enter a barbershop with a single barber. The entrance of the person is the transaction. Upon entering the shop, he will make one of the following choices:

1. If the barber is free, the person immediately sits in the chair and the haircut begins.
2. If the barber is busy, the person waits until the barber is free.
3. If the shop has too many people waiting, the person will leave.

The Internal GPSS/H Clock

A system such as the barbershop normally involves the passage of time; therefore, an imaginary, internal clock is used by the GPSS/H processor. The internal clock starts at zero when the GPSS/H program begins execution. The processor moves the clock forward in time as the program is executed.

A knowledge of how the internal GPSS/H clock works is important for an understanding of how a GPSS/H program executes. The program does not go

from line to line as in other computer programming languages but follows an imaginary time axis.

The time units used by the internal clock can represent whatever amount of time that the programmer chooses, e.g, 1 second, 5 minutes, 12 hours, or 0.1 minute. The time unit–called the *basic time unit*–should be selected that best suits the problem being studied. In some cases, a basic time unit of 1 second may be selected; in other cases, the basic time unit may be 1 hour. If a store is being simulated and studies indicate that customers take 18 minutes to shop and 2 minutes to check out, an appropriate basic time unit is 1 minute. If the simulation is to run for a simulated time of 8 hours, then the 8 hours need to be converted into basic time units–in this case, 480 time units. The basic time unit to select is generally obvious from the statement of the problem being modeled.

The internal GPSS/H clock is advanced by the processor as the simulation steps forward from event to event. For example, if some event is to take place at time $t = 345.765$ and the next one at $t = 420.5$, the internal clock will be advanced from its previous time to time 345.765. It will then be advanced to time 420.5.

CREATING TRANSACTIONS

Transactions are created by the GENERATE block. It can have up to 9 operands and can be quite involved. The simplest form–either with or without an optional label, indicated by the parentheses and lowercase letters–is given by the following:

```
◇LABEL    □GENERATE  □A
```

where *A*, the operand, is either a positive integer or a variable. *A* controls the times at which a transaction can be created.

The label is optional. We will learn when to use one. For the present, it is not needed, but in this book, simple lowercase letter labels in parentheses (which are forbidden in actual GPSS/H code; labels must be alphanumeric; see Table 5.1) will be used just to identify lines of code for the reader's benefit. For example,

```
◇(a)      □GENERATE  □5
◇(b)      □GENERATE  □100
```

In (a), a transaction is created every 5 time units, whereas in (b), a transaction is created every 100 time units. In both (a) and (b), as long as the simulation is taking place, *transactions will continue to be created every 5 or 100 time units, respectively.*

In general, a transaction will be moved from block to block in a sequential manner, unless the transaction encounters a block that transfers it elsewhere in the program. Such a nonsequential transfer takes place only when the programmer specifies.

In most systems to be studied, there is a degree of randomness involved. People do not enter a bank every 35 seconds, a barber does not take exactly 8.5 minutes for a haircut, etc. Thus, it is necessary to have randomness as a part of the simulation. One of the features of GPSS/H is that it is so easy to incorporate randomness into a simulation.

One way to have randomness in the generation of transactions is to use the second form of the GENERATE block, which incorporates the *B* operand:

```
        GENERATE  A,B
```

B can be a variable, but in this book, it will always be a positive integer. This block generates a transaction over the time interval $A \pm B$, with each time interval having an equal probability of happening. For example,

```
        GENERATE  8,2
```

means that a transaction is created every 8 ± 2 time units. This means that a transaction will be created at an interarrival time of 6 to 10 time units with equal probability. Although *A* and *B* are integers, the times are sampled from throughout the interval (6.0000,10.0000).

One indicates an interval by using parentheses and/or brackets. For example, the interval from 0 to 10 might be (0,10). However, does this mean that 0 and 10 are included? Mathematicians use parentheses— ( ) —to mean that the end-points *are not* included. Square brackets— [ ] —are used to indicate that the endpoints *are* included. Thus, [0,10) would include the 0 but not the 10. This difference may seem a minor point, but there is a place in GPSS/H where it is very important.

Therefore, the parentheses around the 6.0000,10.0000 range indicate that the times of 6.0000 and 10.0000 are not included, but 6.0001 and 9.9999 are. This means that if the internal clock is at t = 1013.0000 and the GPSS/H processor sets t = 6.0001 as the time before the next transaction is created, the next transaction will enter the system at simulated time t = 1019.0001. Chapter 14 on functions shows how to generate transactions that enter the system according to *any* statistical distribution.

How GPSS/H Creates Random Numbers

Although it is not of concern how the processor works, you might be curious how the processor generates the various times by using random numbers. The processor has a built-in random-number generator that is referred to in the code from time to time. Suppose you want to generate times from 10 to 24 with equal probability. Call these times *X*. The random number will be called RN, where RN is a number from 0 to 1. Now consider the formula

$X = 10 + RN * (24 - 10)$

Every time a random number is called up, a new value of *X* is obtained. As can be seen, a stream of times *X* between 10 to 24 will be obtained. This approach is similar to the way that the GPSS/H processor works.

Prohibition of Negative Times

In using the GENERATE block, you must be careful to avoid a block with the *B* operand greater than the *A* operand, e.g.,

```
        GENERATE  10,12
```

as the specification of $B > A$ would eventually lead to the choice of a negative time at which to generate a transaction. Negative times are not allowed. However, it is possible to have

```
◇         GENERATE  10,10
```

Example 6.1

People arrive at a barbershop every 15 ± 6 minutes. The single barber takes 12 ± 4 minutes to cut hair. Simulate for 200 people having their hair cut. Assume that the barber works continuously, i.e., he does not leave the shop until 200 people have had their hair cut. Determine how busy the barber has been. The program to do this simulation is as follows:

```
◇         SIMULATE
◇         GENERATE  15,6        People enter the shop
◇         QUEUE     WAITAREA    Take a seat in waiting area
◇         SEIZE     BARBER      Engage barber if he is free
◇         DEPART    WAITAREA    Leave the seat in waiting area
◇         ADVANCE   12,4        Receive haircut
◇         RELEASE   BARBER      Haircut over, barber is free
◇         TERMINATE 1           Leave the shop
◇         START     200         Simulate for 200 customers
◇         END
```

Solution

Do not be too concerned at this time that you do not understand all the code given above. In fact, even interpreting the results will appear strange at this time. However, if you successfully run the program and look at the list file (your .GPS filename, but with the .LIS extension) created by GPSS/H, you will see the following as a part of the output:

```
RELATIVE CLOCK: 3005.4781   ABSOLUTE CLOCK: 3005.4781

         --AVG-UTIL-DURING--
FACILITY  TOTAL     ENTRIES
  BARBER  0.804       200
```

The output is interpreted as follows: The barber worked for approximately 3005 minutes or 50 hours straight to take care of the 200 customers. He was busy 80.4% of the time. This value of about 80% is to be expected. Customers arrive at random but with an average interarrival time of 15 minutes. The barber takes an average of 12 minutes to give a haircut. Hence, in the long run, one would expect the barber to be busy for 12/15 or 80% of the time. Furthermore, in this example, when the haircut for the 200th customer was finished, the barber closed shop and left, even though there may have been other customers waiting in the shop. Later, when we learn more about the GPSS/H language, it will be possible to write the code to model more realistic examples.

Example 6.2

Suppose the barber decides to buy special, fancy new equipment that can really speed up his hair cutting. He can now cut hair in 10 ± 3 minutes. To rerun Example 6.1 and examine the results, change the operands of the ADVANCE block:

```
◇        ADVANCE   10,3
```

MORE GENERAL CASES OF THE GENERATE BLOCK

Other operands can be used in longer forms of the GENERATE block. There are five operands that we will use at present (note that all are typically positive integers but can be variables; in practice, only *A* is used as a variable):

```
◇        GENERATE  A,B,C,D,E
```

where, as before, operand *A* controls the times at which a transaction can be created and operand *B* allows a range over which the transaction creation times can vary. Operand *C* is called the offset time for the first transaction, i.e., no transaction will enter the system until this time. Operand *D* is the maximum number of transactions to be generated. Operand *E* is called the priority. When the *E* operand is used, the transactions are given a priority level specified by it. Often, this operand is omitted and the priority level is, by default, 0. The priority levels can be integers between -2,147,483,632 and +2,147,483,632. In practice, only a few priorities are needed, and one normally uses priorities such as 1, 2, 5, 10, etc.

In most queueing systems, the service criterion is known as "first-in first-out" (FIFO). This means that, if the first person to arrive for service has to wait, he or she will be served before later arrivals. If, however, one transaction has a higher priority than another, the higher-priority transaction is placed *ahead* of the lower-priority transaction in queues as well as given preferential service in the case of a time tie between transactions. A typical example to illustrate the case of a time tie is the situation of the arrival of a car at a filling station. Suppose that there is only room for 6 cars total and, if there are 6 cars at the station, an arriving car will leave. A time tie occurs when a car arrives at exactly the same time as one is through being serviced; the arriving one will not leave if it has a higher priority than the one that is about to leave. This, of course, is what will happen in a real-life situation. We will examine this concept of priority in Chapter 28.

```
◇(a)     GENERATE  15,3,100,3,1
◇(b)     GENERATE  100,3,200,400,3
◇(c)     GENERATE  20,4,500,7,8
◇(d)     GENERATE  30,0,0
```

Blocks (a) to (d) represent several examples of the GENERATE block. In (a), a transaction is created every 15 ± 3 time units. The first transaction does not enter the system until $t = 100$. Exactly 3 such transactions will be created, and each will have a priority level of 1. In (b), a transaction is created every 100 ± 3 time units. The first transaction enters the system at $t = 200$, and exactly

400 such transactions will be created, each having a priority level of 3. In (c), transactions are created every 20 ± 4 time units. The first transaction enters the system at $t = 500$. Exactly 7 such transactions are created, each with a priority level of 8. In (d), transactions are created every 30 time units. The first transaction will enter the system at $t = 0$. Contrast block (d) with the block

```
(e)     GENERATE    30
```

In block (e), the first transaction will enter the system at $t = 30$, not at $t = 0$.

If you do not wish to use all the operands, you use commas instead–*not* blanks, i.e.,

```
(f)     GENERATE    100,,,1
```

will generate a single transaction at $t = 100$, and

```
(g)     GENERATE    100,,,,1
```

will generate a transaction every 100 time units, each transaction having a priority level of 1.

```
(h)     GENERATE    ,,,5
```

will generate 5 transactions immediately. This block form may seem strange but is an important option in many simulations. Block (h) is the same as block (i):

```
(i)     GENERATE    0,0,0,5
```

If you want to make a simulation involving 6 ships sailing from one port to another, you might start by using the block

```
(j)     GENERATE    ,,,6
```

Block (j) puts 6 ships into the system at time $t = 0$.

Example 6.3

Go back to Example 6.1. Now suppose that when the barber arrives at work, he finds 3 people waiting for haircuts. Change the program to reflect this change in the simulated situation. The program for this is as follows:

```
        SIMULATE
        GENERATE   ,,,3        People waiting
        TRANSFER   ,DOWN       Send them to the door
        GENERATE   15,6        People enter the shop
DOWN    QUEUE      WAITAREA    Take a seat in waiting area
        SEIZE      BARBER      Engage barber if he is free
        DEPART     WAITAREA    Leave the seat in waiting area
        ADVANCE    12,4        Receive haircut
        RELEASE    BARBER      Haircut over, barber is free
        TERMINATE  1           Leave the shop
        START      200         Simulate for 200 customers
        END
```

Example 6.4

Suppose that the customers in Example 6.3 arrive as before, but now it is desired to have them start arriving at 1 minute after the barbershop opens. Compare the output from the previous program after incorporating the following change in the first GENERATE block:

```
◇        GENERATE   15,6,1        People enter the shop
```

THE TERMINATE BLOCK

As shown in Example 6.3, it is possible to have more than one GENERATE block in a program; in fact, this is nearly always the case. The only caution is that a transaction can never *enter* a GENERATE block. To avoid this pitfall, one employs the TERMINATE operation:

```
◇        GENERATE   1
◇        -----
◇        -----
◇        -----
◇        -----
◇        TERMINATE
◇        GENERATE   5000
◇        TERMINATE  1
```

Transactions enter the system by means of the GENERATE block. Eventually, in most simulations, the transactions will have to leave the system. Their removal is accomplished by means of the TERMINATE block. Whenever a transaction enters a TERMINATE block, it is immediately removed from the system. TERMINATE blocks are quite simple in form:

```
◇        TERMINATE  A
```

where the operand *A* is a positive integer, including 0.

If *A* is omitted, as it often is, it is taken to be 0. The operand *A* in TERMINATE *A* has *nothing* to do with–i.e., does not specify or limit–the transaction being removed from the system. Only one transaction at a time enters the TERMINATE block. The TERMINATE block *always* removes this one transaction. As we shall see, the operand *A* is used to control the execution of the program in connection with the START statement, described next.

THE START STATEMENT

Every GPSS/H program *must* have a START statement. The simplest form of it is

```
◇        START      A
```

where operand *A* must be a nonzero positive integer.

The operand *A* is a counter for controlling the running of the program. While the program is being executed, the counter is being decremented. When it

becomes zero or negative, the program stops execution. The GPSS/H processor then creates an ASCII file, NAME.LIS, where NAME is the name of the original .GPS input file. This file contains the results of the simulation and can be viewed by using the same text editor used to create the original input file for GPSS/H processing.

The way the processor knows that the program is finished is as follows:

1. The counter operand, *A*, is "set aside."
2. Whenever a transaction goes through a TERMINATE block that has an operand, this operand value is subtracted from the counter operand.
3. After the transaction in which the remainder becomes 0 or negative, the simulation is finished, and the report (i.e., the NAME.LIS file) is produced.

For example, if the GPSS/H program executes the following TERMINATE and START lines of code,

```
◇      TERMINATE  2
◇      -----
◇      START      10
```

the program will continue until 5 transactions have passed through the TERMINATE 2 block (i.e., pass 1—START = 10, 10 - 2 = 8; pass 2—START = 8, 8 - 2 = 6; pass 3—START = 6, etc., until START = 0). For the following lines of code,

```
◇      TERMINATE
◇      -----
◇      TERMINATE  3
◇      -----
◇      START      13
```

the program will run until 5 transactions have passed through the TERMINATE 3 block. Any transaction that passes through the first TERMINATE block will have no effect on the execution time of the program (because there is no operand *A*). And for the following lines of code,

```
◇      TERMINATE  4
◇      -----
◇      START      1
```

the program will run until 1 transaction has passed through the TERMINATE 4 block.

Most of the programs so far have had blocks such as the following:

```
◇      GENERATE   480
◇      TERMINATE  1
◇      START      1
```

The effect of this sequence of blocks is to put a single transaction into the system at time 480. This transaction is immediately removed via the TERMINATE 1 block. The 1 in its operand causes the counter of 1 that was given by the START 1 statement to be decremented to 0. Thus, the program stops execution at time 480.0000. Transactions whose primary effect on the program is to stop its execution are called *timer transactions*.

A GPSS/H program starts execution when the *first* START statement is encountered. If there are more START statements after this first START statement, *they are initially ignored*. When the program is through with execution as specified by the first START statement, whatever commands are given after it are then completed.

During the compiling stage, transactions are primed to move through the system at times given by the GENERATE block. In the priming process, the transactions are placed on a time axis called the *current events chain* (CEC). When the program begins execution, the processor takes over and moves the transactions one at a time as far as each transaction can move. After the processor moves a transaction, the processor goes back to the CEC and moves the next transaction. This continues as long as the program is running. The first transaction to be moved is the one positioned on the CEC at the earliest time, the next is the one at the next earliest time, etc. The transaction is moved from block to block in a sequential manner, unless the program specifies otherwise. A transaction is moved until one of the following things happens:

1. The transaction is removed from the system via a TERMINATE block.
2. The transaction is put on another current events chain to be covered later.
3. The transaction is destructed and cannot enter the next sequential block.

It is important to keep in mind that the second and subsequent transactions to leave a GENERATE block are not scheduled to do so until the first transaction has left the block. Even though the effect of a block such as GENERATE 10 is to create a transaction every 10 time units, the series of transactions is created as follows: the first is scheduled at time 10 *during* compiling, and the second at time 10 after the first transaction has left the block.

EXERCISES, CHAPTER 6

1. What will happen when the following GENERATE blocks are used in a program?

```
◇(a)    GENERATE  100,30,,5
◇(b)    GENERATE  ,,1000,4
◇(c)    GENERATE  1000,200,20,100
◇(d)    GENERATE  500,400,1
◇(e)    GENERATE  ,,,4
```

2. You are observing trucks at point A. Every 4 minutes, 1 truck passes this point. The trucks travel along the road that passes point A for 5 minutes and then leave the system. In 20 minutes, how many trucks have you observed? Assume that the first truck does not pass you until 4 minutes have passed. The GPSS/H program to simulate this is as follows

```
◇	SIMULATE
◇	GENERATE  4
◇	ADVANCE   5
◇	TERMINATE 0
◇	GENERATE  20
◇	TERMINATE 1
◇	START     1
◇	END
```

3. In exercise 2, suppose that the trucks pass point A every 4 minutes but starting at $t = 0$. Change the appropriate line in your program.
4. In exercise 2, assume that a truck will pass point A every 4 ± 2.5 minutes. Change the appropriate line in your program.
5. In exercise 2, you may have noticed that, after 20 minutes, the computer program said that you observed only 4 trucks. What happened to the truck at time 20? Suppose you change the program to give the trucks you are observing a priority level of 1. What line needs to be changed? How does this affect the program results?
6. In exercise 2, simulate for 40 minutes.
7. Write the GPSS GENERATE blocks to
   a. Have transactions enter the system every 5 time units.
   b. Have transactions enter the system every 100.6 time units.
   c. Have transactions enter the system every 10 ± 6.5 time units.
   d. Have transactions enter the system every 7 time units starting at t =100.
   e. Have transactions enter the system every 120 ± 35.6 time units beginning at $t = 200$.
   f. Have 5 transactions enter the system at $t = 0$.
   g. Have transactions enter the system every 100 time units beginning at $t = 80$ and only 10 enter the system from this block.
   h. Have only 3 transactions enter the system at $t = 500$. These transactions have priority 5.
   i. Have transactions with priority 5 enter the system every 6 ± 3.4 time units, staring at $t = 400$. Only 6 of these transactions are to enter the system.
8. What would happen if you had the following block?

```
◇	GENERATE  120,130
```

9. What would happen if you had the following block?

```
◇	GENERATE  4,1,,,-1
```

10. What do the following lines of code do?

```
◇	GENERATE  ,,480,1
◇	TERMINATE 1
◇	START     1
```

11. The following code is used to time the running of a GPSS/H program. What time will the simulated clock show at the completion of the program?

```
(a)   GENERATE   480
      TERMINATE  2
      START      2
(b)   GENERATE   4800
      TERMINATE  5
      START      10
(c)   GENERATE   1000
      TERMINATE  3
      START      7
(d)   GENERATE   200
      TERMINATE  10
      START      1
```

12. What time will the simulated clock read when the following program is done executing?

```
      SIMULATE
      GENERATE   100
      TERMINATE  1
      GENERATE   150
      TERMINATE  2
      START      10
      END
```

SOLUTIONS, CHAPTER 6

1. a) Transactions are created every 100 ± 30 time units. Only 5 of these will enter the system.
   b) 4 transactions are created. They will enter the system at time 1000.
   c) Transactions are created every 1000 ± 200 time units. They will start to enter the system at time t = 20. 100 of these transactions will enter the system.
   d) Transactions are created every 500 ± 400 time units. The first enters the system at time t = 1.
   e) 4 transactions are created at time t = 0.000.
2. You will observe 4 trucks pass you. At time t = 20, the 5th truck is in front of you.
3. Change Line 2 to

   ```
   GENERATE    4,,0
   ```
4. Change the line

   ```
   ADVANCE    5
   ```
   to

   ```
   ADVANCE    4,2.5
   ```
   You still will observe the same number of trucks entering the system, but now they will remain for varying times.

5. Change Line 2 to

```
GENERATE    4,,,,1
```

The 5th truck entered the system before the program ended. This will be shown in the .LIS file.

6. This can be done in several ways. Some of them are:

a)
```
GENERATE    40          Line 5
```
b)
```
GENERATE    20          Line 5
START       2           Line 7
```
c)
```
GENERATE    10          Line 5
START       4           Line 7
```

7. a) `GENERATE    5`

b) `GENERATE    100.6`

c) `GENERATE    10,6.5`

d) `GENERATE    7,,100`

e) `GENERATE    120,35.6,200`

f) `GENERATE    ,,,5`

or `GENERATE    0,0,0,5` (the first is the commonly used method)

g) `GENERATE    100,,80,10`

h) `GENERATE    ,,500,3,5`

i) `GENERATE    6,3.4,400,6,5`

8. You cannot go backward in time. Thus, the spread of 120 ± 130 would be from −10 to 240. An attempt to generate a transaction at a negative time eventually would occur (depending on the random numbers used). This would lead to an error. The compiler will show the error as follows:

*** IN STATEMENT 2 - ERROR 163 - B - operand (modifier) exceeds A operand (mean).

9. A transaction is created every 4 ± 1 time unit with priority −1. (Only GPSS/H allows negative priorities. In previous version of GPSS/H, only positive non-negative integers were allowed for priorities).

10. The first transaction enters the system at time 480 according to the C operand. The D operand indicates that only 1 such transaction would enter the system.

You may note that this is equivalent to:

```
GENERATE   480
TERMINATE  1
START      1
```

If the aim of the analyst is to run the simulation for 480 time units, the effect of the different lines of code is exactly the same.

11. a) The internal GPSS/H counter initially is set at 2 because of the START 2 statement. At time 480, a transaction enters the system and immediately is terminated. The TERMINATE operand is 2, so the counter is decremented by this amount. Since this sets it to zero, the program stops at this time, namely, t = 480.0000.

b) t = 9600.0000. At time 4800.0000, the counter is decremented from 10 to 5. At time 9600.0000, the counter goes to 0.0000.

c) At time 1000.0000, the counter is decremented by 3 to 4. At time 2000.0000 the counter is decremented to 1. At time 3000.0000, the counter is decremented to -2. Since it is now less than zero, execution will stop.

d) At time 200.0000, the counter is decremented by 10 to -9. This stops execution of the program.

12. The clock will read 450.000. The START operand was set at 10. It was decremented at the times:

Time	Value
100	9
150	7
200	6
300	3
400	2
450	0

This is shown as follows:

Time	Counter	Decrement Amount
0	10	0
100	9	1
150	7	2
200	6	1
300	3	3
400	2	1
450	0	2

Thus, the clock will read 450.0000.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 7

The *TRANSFER* Block

A transaction normally will move from block to block in a sequential manner. When the processor moves it along the current events chain, the transaction moves as far as it can move. As we learned in Chapter 6, one of three things eventually happens to the transaction to cause the processor to rescan the CEC for the next transaction to be moved:

1. The transaction is terminated.
2. It is put on some other chain.
3. It is obstructed and denied access to the next block.

It is possible to have the transaction move in a nonsequential manner. This is the purpose of the TRANSFER block. The TRANSFER block acts much like the GO TO statement found in other programming languages such as Fortran. However, in GPSS/H, there are several forms that can be used. In this chapter, the three most common ones will be discussed. Other forms are covered in Chapter 22.

UNCONDITIONAL TRANSFER BLOCK

The form of the unconditional TRANSFER block is

```
◇          TRANSFER  ,B
```

where the operand *A* is blank (as indicated by the presence of the comma, which is *essential*) and operand *B* is the label of the destination block. When a transaction enters the TRANSFER block, it is immediately sent to the block with the label given by the TRANSFER block's operand *B*. Examples are

```
◇          TRANSFER  ,DOWN
◇          TRANSFER  ,WAIT
◇          TRANSFER  ,UPTOP
```

If the block to which the transaction is being transferred will not admit the transaction, it is held in the TRANSFER block until a later scan of the CEC by the processor. The TRANSFER block always admits transactions.

Because the GPSS/H processor automatically numbers all blocks (see the output near the end of this chapter for an example), it is possible to use sequential block numbers instead of block labels to indicate the destination block. The general form is the same:

```
◇        TRANSFER  ,B
```

where operand *B* is a nonzero positive integer. In this case, the transaction is transferred to the block whose sequential block number, counting from the top, is equal to the value in the TRANSFER block's operand *B*. Thus,

```
◇        TRANSFER  ,6
```

will send the transaction to the sixth block in the program. This approach using sequential block numbers can be a bit confusing in the case of large programs and is not as easy to follow as when block labels are used. Also, if a program is changed by adding additional blocks, all such TRANSFER blocks have to be changed. For these reasons, this form of the TRANSFER block will not be used in this chapter.

CONDITIONAL TRANSFER BLOCK

The form of the conditional TRANSFER block is

```
◇        TRANSFER  A,B,C
```

where operand *A* is a decimal fraction ($0 < A < 1$) of no more than 3 digits (it can have fewer) and operands *B* and *C* are block labels. Operand *B* is optional but operand *C* is required. Examples are

```
◇        TRANSFER  .123,DOWN,UPTOP
◇        TRANSFER  .5,,AWAY
◇        TRANSFER  .007,NEXT1,NEXT2
◇        TRANSFER  0.9,UPTOP,DOWNX
```

The conditional TRANSFER block proportions the transfer of transactions between the two blocks identified by the labels in the operands. Of the total number of transactions that enter this version of the TRANSFER block, a fraction equal to the value of operand *A* are sent to the block with the label in operand *C* of the TRANSFER block. The rest of the transactions are routed to the block with the label given in operand *B*. If operand *B* is missing (as it most commonly is), the transaction is routed to the next sequential block. Two examples of the conditional TRANSFER block cover the possible configurations:

```
◇(a)     TRANSFER  .333,DOWN,OUT
◇(b)     TRANSFER  .8,,AWAY
```

In example (a), 33.3% of the transactions are sent to the block with the label OUT. The remainder or 66.7% of the transactions are sent to the block with the label DOWN. In example (b), 80% of the transactions are sent to the block with the label AWAY. The remainder of the transactions continue to the next sequential block.

TRANSFER BOTH

The following form of the TRANSFER block is not as common as the first two cases but can be extremely helpful. It is called the TRANSFER BOTH mode. The general form of it is

```
          TRANSFER  A,B,C
```

where the word BOTH *must* be in operand *A* and operands *B* and *C* are block labels of blocks to which the transaction can be transferred. Operand *B* is optional.

The TRANSFER BOTH block works as follows: A transaction enters the TRANSFER BOTH block, and the processor attempts to send the transaction to the block whose label is indicated in operand *B*. If the transaction can enter that block, the transaction proceeds there. If not, it attempts to enter the block whose label is indicated in operand *C*. If it can, it does so. If it cannot enter either block, it remains in the TRANSFER BOTH block until a new scan of the CEC takes place.

If operand *B* is missing, each transaction entering the TRANSFER BOTH block moves to the next sequential block if permitted or to the block indicated by operand *C*. In fact, an example of the most common form of the TRANSFER BOTH block is

```
          TRANSFER  BOTH,,AWAY
```

where AWAY refers to a block that will always accept the transaction. Thus, the processor "looks ahead" to the next sequential block and attempts to move the transaction into it. If the transaction cannot enter, it is transferred to the block labeled AWAY. The next example illustrates an application of the TRANSFER BOTH block.

Example 7.1

People arrive at a shop every 8 ± 3.5 minutes. The shop has 5 servers who work at the rate of 40 ± 15 minutes. There is a single chair to wait in when all of the servers are busy. If the shop is full, i.e., all 5 servers are busy and a person is waiting in the chair, arriving customers leave and do not return. Simulate for 20 days operation (480 minutes per day) and determine how many customers are turned away. Even though you do not know enough about GPSS/H to follow all the lines of code, it will be instructive to run the program.

Solution

The program to solve Example 7.1 is as follows:

```
          SIMULATE
          STORAGE   S(WAITAREA),1/S(WORKER),5
          GENERATE  8,3.5
          TRANSFER  BOTH,,AWAY
          ENTER     WAITAREA
          ENTER     WORKER
          LEAVE     WAITAREA
```

```
◇           ADVANCE     40,15
◇           LEAVE       WORKER
◇           TERMINATE
◇AWAY       TERMINATE
◇           GENERATE    9600
◇           TERMINATE   1
◇           START       1
◇           END
```

To run for 2 seats for waiting customers, you change the line of code

```
◇           STORAGE     S(WAITAREA),1/S(WORKER),5
```

to

```
◇           STORAGE     S(WAITAREA),2/S(WORKER),5
```

EXERCISES, CHAPTER 7

Note: Exercises 6–15 are modifications from Gordon (1975).

1. An art museum has 2 galleries: A and B. Every 45 ± 20 seconds, people arrive, and 40% of them plan to visit only gallery A and then leave. The rest go to gallery B first. It takes 220 ± 80 seconds to visit gallery A and 300 ± 60 seconds to visit gallery B. Of the people who visit gallery B, 35% leave but the rest go on to gallery A. (Strangely enough, no one ever goes to gallery A first and then to gallery B). Simulate for 4 hours.
2. Change exercise 1 to simulate for 200 people who visit gallery B and then leave the museum without visiting gallery A.
3. People come to a moving sidewalk (a fancy, segmented metal conveyor belt) at an airport. They arrive every 35 ±15 seconds, and 30% choose to walk alongside it. Of the people who use the moving sidewalk, 25% walk or run on it, and they traverse it in 40 ± 12 seconds. The moving sidewalk transfers people who do not walk or run in exactly 55 seconds. Those who walk alongside of it take 60 ± 32 seconds to cover the distance. Simulate for 1 hour.
4. Change exercise 3 to simulate for 100 people who use the moving sidewalk. How long does it take to transfer them?
5. Figure 7.1 gives a diagram of traffic flow. All streets are one-way traffic. Cars enter at A every 10 ± 3.2 seconds. They take 20 ± 8 seconds to travel to B. At B, 30% go straight, and the rest travel to C. It takes 15 ± 6 seconds to travel to C from B. At C, 26% go straight, and the rest turn to D. It takes 14 ± 3.1 seconds to travel from C to D. At D, 31% turn to E (where you live), and the rest go straight. It takes 12 ± 3 seconds for cars to travel from D to E. You are going to protest that too many cars go by your house in an 8-hour period. Determine the number of cars passing your house in 8 hours. (Note: ignore the fact that traffic patterns vary during the day.)

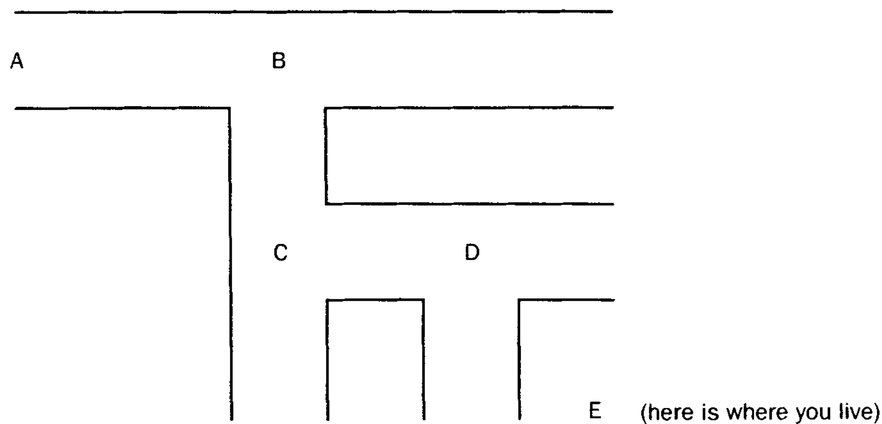


FIGURE 7.1 Sketch of traffic flow.

6. People arrive at a newsstand every 10 ± 5 seconds. Most people buy only 1 paper, but 20% buy 2 papers. It takes 5 ± 3 seconds to buy 1 paper and 7 ± 3 seconds to buy 2 papers. Simulate the sale of 100 papers, starting from the time the newsstand opens. How long does it take the dealer to sell 100 papers?
7. In exercise 6, suppose that every morning the newsstand always has 2 people waiting to purchase a newspaper. Add the code to include this fact. How does your answer change?
8. A series of moving stairways (escalators) carry customers in an upward direction between 4 floors of a department store. People arrive at the foot of the stairs, on the first floor, at the rate of 1 customer every second. Some people walk on the stairs. As a result, the time to transfer between any two floors is 20 ± 10 seconds. The destinations of the customers are as follows: second floor, 50%; third floor, 25%; and fourth floor, 25%. Simulate the arrival of 100 people on the top floor, starting from the time the store opens.
9. As a group, 20 people take a test that requires 5 ± 2 minutes. Their chance of success is such that 20% pass on each attempt. Those that fail wait 10 minutes before taking the test again (it still takes them 5 ± 2 minutes when they retake the test). They keep trying until they finally pass. How long does it take for everyone to pass?
10. Cars bring spectators to a sports event at the rate of 1 car every 20 ± 10 seconds. The percentages of cars with a given number of passengers are as follows: 1 passenger, 10%; 2 passengers, 30%; 3 passengers, 45%; and 4 passengers, 15%. Find how long it takes for 1,000 people to arrive.
11. People arrive at a cafeteria at the rate of one person every 15 ± 5 seconds. There are two counters, A and B, and people want items from them in the following proportions: only counter A, 30%; both counters, 60%; and only counter B, 10%. Simulate the arrival at the cafeteria of 100 people.
12. The delivery of some product is being limited by the availability of suitable reusable containers. A new container is made every 20 ± 5 minutes. The containers are filled and dispatched as soon as they are ready. Delivery

takes 40 ± 10 minutes. About 1 in every 50 containers is damaged beyond repair during delivery. The rest are returned, taking 40 ± 10 minutes, and immediately are reused for another delivery. Beginning from $t = 0$, find how many containers will be in the process of delivery after 8 hours.

13. A subway station has two entrances. Passengers arrive at entrance 1 at the rate of 1 person every 10 ± 5 seconds, and they move along a corridor that takes 15 ± 5 seconds to walk. At entrance 2, passengers arrive at the rate of 1 passenger every 5 ± 2 seconds, and they walk along a corridor that takes 20 ± 8 seconds. The two streams of passengers merge to pass along a third corridor for 5 ± 3 seconds. At the end of that corridor, 60% of the passengers turn for the northbound platform, and the rest turn for the southbound platform. Simulate the arrival of the first 100 passengers on the southbound platform, starting with an empty system. How long does it take for 100 passengers to gather for the first southbound train of the day?
14. Parts that are manufactured at the rate of 1 part every 50 ± 10 seconds go through an inspection that takes 30 ± 10 seconds. The inspection passes 85% of the parts. Of the remainder, 5% are scrapped, and the rest are sent for reworking. Reworking takes 100 ± 30 seconds, after which the parts are again sent for inspection with the same probability of rejection. Simulate the acceptance of 100 parts. How many parts have been reworked by that time?
15. The following program contains some GPSS/H lines of code that will be covered in later chapters–specifically, the two lines with the comments. What the program does is to generate 1000 transactions and transfer them to one of ten TERMINATE blocks (labeled by number in the output) at random.

```
◇         SIMULATE
◇         RMULT      12345        Random number seed
◇         GENERATE   ,,,1000
◇         TRANSFER   ,FRN1*10+3   Random transfer
◇         TERMINATE  1
◇         TERMINATE  1
◇         TERMINATE  1
◇         TERMINATE  1
◇         TERMINATE  1
◇         TERMINATE  1
◇         TERMINATE  1
◇         TERMINATE  1
◇         TERMINATE  1
◇         TERMINATE  1
◇         START      1000
◇         END
```

A portion of the output obtained by running the program is listed next. Note the block numbers from 1 to 12 that were automatically applied by the processor.

```
RELATIVE CLOCK: 0. ABSOLUTE CLOCK: 0.
BLOCK CURRENT      TOTAL  BLOCK CURRENT      TOTAL
1                   1000   11                  105
2                   1000   12                   95
3                    110
4                    101
5                    114
6                    104
7                     84
8                     93
9                     98
10                    96
```

The expected number of times each of the TERMINATE blocks was entered by a transaction is 100. The actual number of entrances in each TERMINATE block varied from 84 to 110 times. The maximum deviation from the expected value was 16 for the TERMINATE block that was entered only 84 times. The line of code

```
◇          RMULT      12345
```

is used to position the random-number generator. Rerun the program with a different random-number seed by changing the value of the operand of the RMULT statement to some other number. Then, rerun the program for 10,000 transactions and see how close the actual number of entries to each TERMINATE block is to the expected number of entries.

SOLUTIONS, CHAPTER 7

1. The program to do the simulation is:

```
      SIMULATE
      GENERATE     45,20            PEOPLE ARRIVE
      TRANSFER     .40,,ROOMA       40% TO ROOMA
      ADVANCE      300,60           SPEND TIME IN ROOM B
      TRANSFER     .35,,LEAVE       35% THEN LEAVE
ROOMA ADVANCE      220,40           SPEND TIME IN ROOM A
LEAVE TERMINATE                     LEAVE THE GALLERY
      GENERATE     3600*4           SIMULATE FOR 4 HOURS
      TERMINATE    1                ALL DONE
      START        1                START
      END
```

2. Change the block

```
LEAVE TERMINATE                     LEAVE THE GALLERY
```

to

```
LEAVE TERMINATE    1                LEAVE THE GALLERY
```

Delete the blocks

```
GENERATE             3600*4           SIMULATE FOR 4 HOURS
TERMINATE            1                ALL DONE
```

The START block is now

```
START  200
```

3. The program to do the simulation is:

```
       SIMULATE
       GENERATE     35,15          PEOPLE ARRIVE
       TRANSFER     .3,,ALONG      30% GO ALONG SIDE
       TRANSFER     .25,,FASTON    25% WILL WALK/RUN
       ADVANCE      55             SIDEWALK MOVES
       TRANSFER     ,DONE          ALL DONE FOR SOME
FASTON ADVANCE      40,12          WALK/RUN ON SIDEWALK
 DONE  TERMINATE                   END OF SIDEWALK
ALONG  ADVANCE      60,32          WALK/RUN ALONG SIDE
       TERMINATE                   END OF SIDEWALK
       GENERATE     3600           SIMULATE FOR AN HOUR
       TERMINATE    1              TIMER TRANSACTION LEAVES
       START        1              START PROGRAM
       END
```

4. Change the block

```
DONE  TERMINATE                 END OF SIDEWALK
```

to

```
DONE  TERMINATE     1           END OF SIDEWALK
```

Remove the two timer transaction blocks and change the START 1 block to:

```
START   100.
```

5. The program to do the simulation is:

```
     SIMULATE
     GENERATE     10,3.2         CARS ENTER AT A
     ADVANCE      20,8           TRAVEL TO B
     TRANSFER     .3,,OUT        70% TURN TO C
     ADVANCE      15,6           TRAVEL TO C
     TRANSFER     .26,,OUT       74% TURN TO D
     ADVANCE      14,3.1         TRAVEL TO D
     TRANSFER     .69,,OUT       31% TURN TO E
     ADVANCE      12,3           TRAVEL TO E
     TERMINATE                   CAR LEAVES
OUT  TERMINATE                   CAR LEAVES
     GENERATE     3600*8         TIMER TRANSACTION
     TERMINATE    1              END OF SIMULATION
     START        1              BEGIN SIMULATION
     END
```

6. The program to do the simulation is:

```
     SIMULATE
     GENERATE    10,5            CUSTOMERS ARRIVE
     TRANSFER    .2,,DOWN        20% WANT TWO PAPERS
     ADVANCE     5,3             REST BUY ONE PAPER
     TERMINATE   1               PEOPLE LEAVE NEWSSTAND
DOWN ADVANCE     7,3             BUY TWO PAPERS
     TERMINATE   2               PEOPLE BUY TWO PAPERS
     START       100             RUN FOR 100 PAPERS
     END
```

The time to run the above was 832.3134 time units.

7. The program to do the simulation is:

```
        SIMULATE
        GENERATE    ,,,2           CUSTOMERS WAITING
        TRANSFER    ,NEXT          SHOP OPENS - BUY PAPERS
        GENERATE    10,5           CUSTOMERS ARRIVE
NEXT    TRANSFER    .2,,DOWN       20% WANT TWO PAPERS
        ADVANCE     5,3            REST BUY ONE PAPER
        TERMINATE   1              PEOPLE LEAVE NEWSSTAND
DOWN    TERMINATE   2              PEOPLE BUY TWO PAPERS
        START       100            RUN FOR 100 PAPERS
        END
```

Now the time to sell 100 papers is reduced to 814.1544 seconds.

8. The program to do the simulation is:

```
        SIMULATE
        GENERATE    1
        TRANSFER    .5,,THIRD      50% GO TO SECOND FLOOR
        ADVANCE     20,10          WALK FROM FIRST TO SECOND FLOOR
        TERMINATE                  ARRIVE SECOND FLOOR
THIRD   TRANSFER    .5,,FOURTH     25% WANT TO GO TO THIRD FLOOR
        ADVANCE     20,10          WALK FROM FIRST TO SECOND FLOOR
        ADVANCE     20,10          WALK FROM SECOND TO THIRD FLOOR
        TERMINATE                  ARRIVE THIRD FLOOR
FOURTH  ADVANCE     20,10          WALK FROM FIRST TO SECOND FLOOR
        ADVANCE     20,10          WALK FROM SECOND TO THIRD FLOOR
        ADVANCE     20,10          WALK FROM THIRD TO FOURTH FLOOR
        TERMINATE   1              ARRIVE FOURTH FLOOR
        START       100
        END
```

Note that the above problem assumes that people who go to the second floor go only there and the same for floors three and four. The solution gives: 506 people entered the stairs; 257 went to the second floor; 137 to the third and 112 to the fourth. The program did not end until 100 people *arrived* at the fourth floor.

9. The program to do the simulation is:

```
        SIMULATE
        GENERATE    ,,,20          PEOPLE TO TAKE TEST
BACK    ADVANCE     5,2            TAKE TEST
        TRANSFER    .2,,DONE       20% PASS
        ADVANCE     10             WAIT 10 MINUTES
        TRANSFER    ,BACK          TAKE TEST AGAIN
DONE    TERMINATE   1              LEAVE SYSTEM
        START       20
        END
```

The time for the people to pass the test was 280.6469. This is quite large. The number of people taking the test, including repeats was 115. Do you think that the time of 280.6469 means much? Interestingly enough, most people would not be able to guess the "solution" to this problem. The answer depends on doing the problem many times and forming confidence limits. Actually, it would be very easy to run it many times using DO loops which are a part of GPSS/H. These are covered in a later chapter.

10. The program to do the simulation is:

```
      SIMULATE
      GENERATE   20,10        CARS ARRIVE
      TRANSFER   .90,,NEXT1   10% HAVE ONE PERSON
      TERMINATE  1            ONE PERSON GETS OUT
NEXT1 TRANSFER  .666,,NEXT2   30% HAVE TWO PEOPLE
*****************************************************
*  NOTICE THAT THE PERCENTAGE USED ABOVE IS 66.6%. THIS
*  IS BECAUSE AFTER 10% OF THE CARS DROP OFF THEIR ONE
*  PASSENGER, THE REMAINING NUMBERS THAT HAVE TWO PEOPLE
*  WOULD BE 30/90 AND 60/90 WITH MORE (3 OR 4)
********************************************************
      TERMINATE  2           TWO PEOPLE ARRIVE
NEXT2 TRANSFER   .25,,NEXT3  45% HAVE FOUR PEOPLE
      TERMINATE  3           THREE PEOPLE ARRIVE
NEXT3 TERMINATE  4           FOUR PEOPLE ARRIVE
      START      1000
      END
```

The time for the simulation to run was 7439.7176 time units. 380 cars came; 35 had 1 person; 119 had 2; 175 had 3, and 51 had 4.

11. The program to do the simulation is:

```
      SIMULATE
      GENERATE   15,5        PEOPLE ARRIVE AT COUNTER
      TRANSFER   .3,,AONLY   30% WANT A ONLY
      TRANSFER   .857,,AANDB 60% WANT A & B
      TERMINATE  1           A AND B
AANDB TERMINATE  1           B ONLY
AONLY TERMINATE  1           A ONLY
      START      100
      END
```

The simulation ran for 1484.4985 time units. The number of customers who went to A only was 27; B only 13, and A and B was 60.

12. The program to do the simulation is:

```
       SIMULATE
       GENERATE   20,5     NEW CONTAINERS ARRIVE
BACK   ADVANCE    40,10    MAKE DELIVERIES
       TRANSFER   .02,,DAMAGE  1 OF 50 DAMAGED
       ADVANCE    40,10    RETURN FOR RE-USE
       TRANSFER   ,BACK    READY FOR ANOTHER DELIVERY
DAMAGE TERMINATE           DAMAGED CONTAINER
       GENERATE   480      SIMULATE FOR 480 TIME UNITS
       TERMINATE   1
       START       1
       END
```

After 480 minutes, there were 24 containers that had been made. At an average of 3 per hour, this is the expected number. Two of these were damaged and discarded. Thus, there are 22 in the system.

13. The program to do the simulation is:

```
      SIMULATE
      GENERATE   10,5     ARRIVE AT ENTRANCE 1
      ADVANCE    15,5     MOVE ALONG CORRIDOR
      TRANSFER   ,DOWN    MERGE WITH OTHER PEOPLE
      GENERATE   5,2      ARRIVE AT ENTRANCE 2
      ADVANCE    20,8     MOVE ALONG CORRIDOR
 DOWN ADVANCE    5,3      MOVE ALONG THIRD CORRIDOR
      TRANSFER   .60,,NORTH  60% GO TO NORTH PLATFORM
      TERMINATE  1           REST GO TO SOUTH PLATFORM
NORTH TERMINATE
      START      100
      END
```

At the end of the simulation, 81 people had arrived from entrance 1 and 160 from entrance 2. The simulation ran for 821.4095 time units.

14. The program to do the simulation is:

```
      SIMULATE
      GENERATE   50,10    MANUFACTURE A PART
UPTOP ADVANCE    30,10    INSPECT A PART
      TRANSFER   .85,,PASS  85% PASS INSPECTION
      TRANSFER   .05,,SCRAP 5% OF THE REST ARE SCRAP
      ADVANCE    100,30    REWORK PARTS
      TRANSFER   ,UPTOP    BACK FOR RE-INSPECTION
PASS  TERMINATE  1         GOOD PART LEAVES
SCRAP TERMINATE            SCRAP PILE
      START      100
```

The simulation ran for 5091 time units. 13 parts needed rework and 0 were scrapped.

REFERENCE

Gordon, G. 1975. *The Application of GPSS V to Discrete System Simulation.* Englewood Cliffs, New Jersey: Prentice-Hall.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 8

The PUTSTRING and PUTPIC Statements and Writing to Files

THE PUTSTRING STATEMENT

The PUTSTRING statement is used to create text either on the screen or in a file. The value of this statement is that it simplifies the running of programs in an interactive mode. The general form of the PUTSTRING statement is

```
LABEL    PUTSTRING A,B
```

The label is optional. The *B* operand gives the text to be printed either to a file or to the screen. If the programmer desires the output of the PUTSTRING statement to be directed to a file (discussed later in this chapter), operand *A* is an assignment statement such as FILE=FILELBL, where FILELBL is the label of the block that defines the file to which the output is to be directed. However, often the programmer desires the output to be sent only to the screen. In this case, the *A* operand *and* its following comma are omitted. Some examples of this latter approach follow:

```
(a)      PUTSTRING (' HELLO THERE')
(b)      PUTSTRING (' THIS IS A GPSS/H PROGRAM')
(c)      PUTSTRING (' SIMULATION IN PROGRESS...')
(d)      PUTSTRING (' notice that small letters are allowed')
(e)      PUTSTRING (' A ± B')
(f)      PUTSTRING ('  ')
```

In (a), the message HELLO THERE is printed on the screen. In (b), the message THIS IS A GPSS/H PROGRAM is printed on the screen. In (c), the message SIMULATION IN PROGRESS... is printed on the screen. In (d), the message notice that small letters are allowed is printed on the screen. In (e) the message *A* ± *B* is printed on the screen. In (f), a blank line is printed on the screen. At least two spaces must be present in the PUTSTRING in order for the blank line to be printed.

It is clear that standard alphanumeric characters are easily output by using the PUTSTRING statement. Originally, the ASCII character set consisted of only 128 characters (letters, numbers, and punctuation). With the advent of

PCs, IBM decided to add 128 characters so that the full 256 character set (which includes certain mathematics symbols such as ±) would be available. These 256 characters are correctly called the *extended ASCII character set.* However, the common terminology now is to call the whole character set simply the ASCII character set.

Sometimes, such as in statement (e), it is useful also to include symbols such as the '±' sign in GPSS/H output. Though these symbols do not appear on a PC keyboard, they can be achieved by holding the <alt> key down on the keyboard and inputting the numerical specification for the symbol by using the numbers on the numerical keypad–not the ones on the top of the keyboard. It is possible to use any of the 256 ASCII characters in a PUTSTRING statement. For the extended symbols, such as ±, simply type the <alt> key plus the desired character's 3-digit number (these numbers are listed in standard programming texts).

```
◇       ▯PUTSTRING ▯(' This is the  sign.')
```

Type <alt>241 to achieve the plus or minus sign. The output of the PUTSTRING is

```
This is the ± sign.
```

However, to use a symbol in a PUTPIC statement, which is covered next, the following, more extensive code procedure is required. Be sure to type <alt>241 in the LET statement:

```
◇       ▯CHAR*1     ▯&J
◇       ▯LET        ▯&J='±'
◇       ▯PUTPIC     ▯&J
 The plus or minus sign is **.
```

The output of the PUTPIC is

```
The plus or minus sign is ±.
```

Other symbols can be used to make PUTSTRING output more readable. For example, the code for the █ is 219 and for ▄ is 220. The statement

```
◇       ▯PUTSTRING ▯(' █ █ █ IMPORTANT INFORMATION FOLLOWS █ █ █')
```

would result in the text

```
█ █ █ IMPORTANT INFORMATION FOLLOWS █ █ █
```

By using the ASCII code for the ▄ together with the █, it is possible to place the title to a program in a box. For example, you might have something like the following:

```
◇       ▯PUTSTRING ▯('  ▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄  ')
◇       ▯PUTSTRING ▯('  █                    █  ')
◇       ▯PUTSTRING ▯('  █  SIMULATION OF     █  ')
◇       ▯PUTSTRING ▯('  █  A SHOP SYSTEM     █  ')
◇       ▯PUTSTRING ▯('  █▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄█  ')
```

THE PUTPIC STATEMENT

Up to this point, all of the output of the example programs that we've run has been produced by the GPSS/H processor and placed by the processor in the .LIS file. Most of the output has been disregarded, as only a limited portion is needed. This is usually the case, as only a few of the values found in this file are ever used. It is possible (and desirable) to have the needed results printed out either to the screen or to a separate file. This procedure is done via the PUTPIC statement. One form of it is

```
◇        PUTPIC    A,B,C,D,E,...
```

where operand *A* indicates the number of lines to be output, operand *B* indicates which file is to receive the output (if any; this option is discussed in the next section, Writing to a File), and all the remaining operands indicate which values are to be output. Operands *A* and *B* (if present) are assignment statements. For example,

```
◇        PUTPIC    LINES=3,FILE=RESULTS,C,D,E,...
```

In this example, GPSS/H will output 3 lines. If only 1 line is to be output, the LINES=1 assignment statement (i.e., the *A* operand) can be omitted. LINE=1 (singular LINE) *cannot* be used even though it is grammatically correct; it will result in an error. The assignment statement FILE=RESULTS is in the *B* operand. If such an assignment statement is omitted, the output is written to the screen instead of to a file. The term "PUTPIC" is derived from "put a picture on the screen."

Some Common Standard Numerical Attributes (SNAs)

A standard numerical attribute (SNA) is any numerical property of a block or attribute of a simulation such as the simulation clock; the block counts; the random numbers; etc. Nearly all of the values listed in the .LIS file are SNAs. As each block is introduced in later chapters, the SNAs associated with it are also given. Other SNAs are also discussed. The SNAs associated with the various blocks are always printed out in the .LIS file, and selected ones can be printed out with the PUTPIC statement. Some SNAs already encountered are

AC1	the absolute clock value
N(LABEL)	how many times that the block labeled (LABEL) has been referenced
W(LABEL)	the current transaction count for the block labeled (LABEL)

The argument of an SNA is placed in parentheses or preceded by a single dollar sign ($). Thus, one could write Q$FIRST, which is the same as Q(FIRST). But the dollar sign approach will not work when the entity forming the argument of the SNA is given a number and not a name. For example, if the queue block is QUEUE 5, then reference to the queue length is given by either Q(5) or Q, but not Q$5. This is an old method of referencing in GPSS/H and will not be used in this book.

Using SNAs in PUTPIC Statements

In the lines of output for PUTPIC, groups of asterisks are used to specify the field and its format. These groups of asterisks may contain decimals. The SNAs are printed out in the asterisk fields in the order in which they appear in the PUTPIC statement, i.e., the first SNA is placed in the first asterisk field, the second in the next asterisk field, etc.

LINES=*n* in operand *A* is used to control the arrangement of the output at the discretion of the programmer. The following are three examples of input and output (note that for convenience where length may be a problem, ROOM1 is shortened to RM1 and ROOM2 is shortened to RM2 in the text of the output):

Option 1

```
◇        ⎕PUTPIC     ⎕LINES=1,AC1,N(ROOM1),N(ROOM2)
 CLOCK ***.** TIMES BLOCK RM1 ENTERED *** TIMES BLOCK RM2 ENTERED ***
```

Note that even though there will only be one line of output, you must specify that fact by using LINES=1, *not* LINE=1, or omit the LINES operand. This example might result in output as follows:

```
CLOCK 123.44 TIMES BLOCK RM1 ENTERED  56 TIMES BLOCK RM2 ENTERED 126
```

Option 2

```
◇        ⎕PUTPIC     ⎕LINES=2,AC1,N(ROOM1),N(ROOM2)
    CLOCK = ***.**
    TIMES BLOCKS ENTERED: RM1 = ***   RM2 = ***
```

This example might result in output as follows:

```
    CLOCK = 123.44
    TIMES BLOCKS ENTERED: RM1 =  56   RM2 = 126
```

Option 3

```
◇        ⎕PUTPIC     ⎕LINES=3,AC1,N(ROOM1),N(ROOM2)
                      CLOCK = ***.**
                      TIMES BLOCK ROOM1 ENTERED  ***
                      TIMES BLOCK ROOM2 ENTERED  ***
```

This example might result in output as follows:

```
                      CLOCK = 123.44
                      TIMES BLOCK ROOM1 ENTERED   56
                      TIMES BLOCK ROOM2 ENTERED  126
```

If the field as specified by the asterisks is not large enough for the value of the SNA, the SNA is still printed out. For example, if one used the vertical-line key to mark the ends of the output,

```
◇        ⎕PUTPIC     ⎕N(THIRD)
 |THE TIMES BLOCK THIRD ENTERED WAS **|
```

and the value of N(THIRD) was 1234, this number would be printed starting where the 2 asterisks were and the | would be moved right 2 spaces.

One can add blank lines to the output by placing a 0 (zero) in position 1 of an output line. The 0 is not printed. For example, the PUTPIC code

```
◇         □PUTPIC     □LINES=3,AC1,W(BLOCKA),N(BLOCKB)
0               RESULTS OF SIMULATION; CLOCK AT ****.**
0               CURRENT COUNT FOR BLOCK A    ***
0               FINAL COUNT FOR BLOCK B      ***
```

might result in output as follows:

```
RESULTS OF SIMULATION; CLOCK AT  3456.44

CURRENT COUNT FOR BLOCK A           6

FINAL COUNT FOR BLOCK B           143
```

Most of the time, however, it is sufficient simply to leave a blank line in the PUTPIC statement to achieve the double-spaced effect.

Note that it is not possible to include any symbol from the extended ASCII character set in the PUTPIC statement unless one uses it as a character. The code to define a symbol as a character was given in the first section of this chapter and will be covered further in Chapter 23.

Example 8.1

Cars arrive at a junction every 10 ± 6 minutes; 25% leave the system along road A, and the rest travel along road B to another junction, which takes 20 ± 8 minutes. Here, 30% of the cars that traveled from A to B leave the system along road C, and the rest take 10 ± 4 minutes to travel to D where they leave the system. Simulate for 100 hours. Have the output show how many cars entered the system, how many of these left the system along each road, and how many remained in the system when the simulation was over.

Solution

The program to simulate this system is as follows:

```
◇         □SIMULATE
◇CARSIN   □GENERATE  □10,6
◇         □TRANSFER  □.25,,AWAYA
◇ROADB    □ADVANCE   □20,8
◇         □TRANSFER  □.4,,AWAYB
◇ROADC    □ADVANCE   □10,4
◇ROADD    □TERMINATE
◇AWAYA    □TERMINATE
◇AWAYB    □TERMINATE
◇         □GENERATE  □60*100
◇         □TERMINATE □1
◇         □START     □1
◇         □PUTPIC    □LINES=6,N(CARSIN),N(ROADD),_
                      N(AWAYA),N(AWAYB),W(ROADB),W(ROADC)
 NUMBER OF CARS ENTERING          ****
 NUMBER REMAINING AT END          ****
```

```
NUMBER TO LEAVE AT A          ****
NUMBER TO LEAVE AT B          ****
NUMBER ON ROAD B AT END       ****
NUMBER ON ROAD C AT END       ****
◇        □END
```

The output is as follows:

```
NUMBER OF CARS ENTERING      595
NUMBER REMAINING AT END      265
NUMBER TO LEAVE AT A         153
NUMBER TO LEAVE AT B         174
NUMBER ON ROAD B AT END        2
NUMBER ON ROAD C AT END        1
```

The number of cars to enter the system was 595; the expected number is 599. (The program went for 6,000 time units, and a car came in on the average of every 10 time units. Six thousand divided by 10 is 600, but the time $t = 6{,}000$ was used to stop the program. The cars actually came into the system for 5,990 time units; 5,990 divided by 10 is 599. Because of the random numbers used, the program came up with 595.) The number remaining at the end was 265; those that left the system along road A totaled 153, and those that left the system along road B totaled 174. There were 3 cars still in the system at the end of the simulation. Notice that 265 + 153 + 174 + 2 + 1 = 595.

WRITING TO A FILE

Output commonly is sent to a file when the amount of output is large or when one wants to use the GPSS/H program to create a file to be used in an animation. In order to write to a file, the file has to be defined with the FILEDEF statement. The form of this is

```
◇filelbl □FILEDEF    □'FILENAME.EXT'
```

Some examples of this statement follow:

```
◇MYOUT   □FILEDEF    □'RESULT.OUT'
◇ATFIL   □FILEDEF    □'TEST.OUT'
◇AAAA    □FILEDEF    □'result.xyz'
◇BBBB    □FILEDEF    □'OUT123'
```

Notice that, whereas the label must be in capital letters (as usual) and must not have an extension, the filename need not be in capital letters and may have any valid extension desired or none at all.

In the PUTPIC, one might have

```
◇        □PUTPIC     □FILE=MYOUT,LINES=2,AC1,N(BLOCKA)
 THE CLOCK IS AT  ****.**
 BLOCK A COUNT    ****
```

The file named RESULT.OUT will have two lines in it. Notice that it makes no difference if FILE=MYOUT comes before LINES=2 or vice versa.

THE BPUTPIC BLOCK

It is possible to create continuous output during the running of a GPSS/H program by means of the BPUTPIC block. Every time a transaction passes through this block, output is created, so one must be careful when using BPUTPIC. It can be extremely useful in debugging a program. It operates in exactly the same way as the PUTPIC statement. If one wanted output every 100 time units in a program, one could include the segment

```
◇        ▯GENERATE  ▯100
◇        ▯BPUTPIC   ▯FILE=AAAA,LINES=2,AC1,N(BLOCKA),N(BLOCKB)
 TIME IS ****.**      BLOCK A COUNT **
 BLOCK B COUNT **
◇        ▯TERMINATE
```

Every 100 time units, the output at that point in the simulation would be placed in the file specified by the label AAAA. One needs to be careful in using this block, since *every* time a transaction passes through it, it is executed. File AAAA could become monstrously large.

Example 8.2

The following example may seem trivial, but the principle involved is useful when one is making files for animations. Write a program that creates a file that has the line TIME *n* where *n* goes from 0.00 to 10.00, to 20.00, etc., up to 100.00.

Solution

```
◇        ▯SIMULATE
◇MYOUT   ▯FILEDEF   ▯'MYOUT.OUT'
◇        ▯GENERATE  ▯10,,0
◇        ▯BPUTPIC   ▯FILE=MYOUT,AC1
 TIME  **.**
◇        ▯TERMINATE
◇        ▯GENERATE  ▯101
◇        ▯TERMINATE ▯1
◇        ▯START     ▯1
◇        ▯END
```

Notice that the timer transaction enters the system at time unit $t = 101$. This is because, if it were to enter at $t = 100$, the last BPUTPIC would not be executed. Alternatively, the transactions that enter the system from the first GENERATE block could have been given a high priority.

Finally, another very useful feature is that it is all right to have arithmetic in a PUTPIC statement or BPUTPIC block. Thus, it is permissible to have N(FIRST)+N(SECOND). In the following exercises, this feature will be applied to problem 3.

EXERCISES, CHAPTER 8

1. Customers arrive at a newsstand every 8 ± 4 seconds. 70% of the customers purchase 1 paper, which takes 5 ± 3 seconds. The rest purchase 2 papers, which takes 7 ± 3.5 seconds. Write the GPSS/H program to determine how long it takes in seconds to sell 1,000 papers. The output should show the time required and the number of people who purchased either 1 or 2 papers.
2. Self-loading vehicles arrive at a quarry. There is plenty of room to load each one so there is never any waiting. Loading each vehicle takes 10 ± 4 minutes. Each vehicles then takes 12 ± 3 minutes to haul, dump, and return. There are 4 self-loading vehicles working. Determine the number of loads dumped in a shift that consists of 360 minutes of actual work.
3. Modify the program in exercise 1 so that the screen will show the number of papers sold after each 10 minutes (600 seconds).

SOLUTIONS, CHAPTER 8

1. The program to do the simulation is:

```
      SIMULATE
      GENERATE 8,4
      TRANSFER.3,,AWAY
      ADVANCE 5,3
ONEP  TERMINATE      1
AWAY  ADVANCE 7,3.5
TWOP  TERMINATE    2
      START      1000
      PUTSTRING    ('  ')
      PUTSTRING    ('  ')
      PUTSTRING    (' OUTPUT FOLLOWS....')
      PUTSTRING    ('  ')
      PUTPIC       LINES=3,AC1,N(ONEP),N(TWOP)
THE TIME TO SELL 1000 PAPERS WAS ****.** SECONDS
NUMBER OF PEOPLE WHO PURCHASED ONE PAPER  ****
NUMBER OF PEOPLE WHO PURCHASED TWO PAPERS ****
      END
```

Notice the use of the PUTSTRING's to add blank lines for more readable output.

2. The program to do the simulation is:

```
       SIMULATE
       GENERATE   ,,,4 PROVIDE 4 VEHICLES
BACK   ADVANCE    10,4 LOAD A VEHICLE
       ADVANCE    12,3 HAUL, DUMP AND RETURN
DUMPED ADVANCE    0 DUMMY TRANSACTION
       TRANSFER   ,BACK
       GENERATE    360
       TERMINATE  1
       START      1
       PUTSTRING  ('  ')
       PUTSTRING  ('  ')
```

```
        PUTPIC      LINES=2,AC1,N(DUMPED)
        SIMULATION FOR ***.** MINUTES
        LOADS DUMPED  ****
        END
```

It was not necessary to have the ADVANCE 0 block. However, when a truck is done hauling and is ready to return, it enters this block. If this block was not here, the count of the loads actually dumped would be harder to determine.

3. The modifications to the program are:

```
        SIMULATE
        GENERATE  8,4
        TRANSFER  .3,,AWAY
        ADVANCE   5,3
   ONEP TERMINATE 1
   AWAY ADVANCE   7,3.5
   TWOP TERMINATE 2
        GENERATE  600
        BPUTPIC    AC1,N(ONEP)+N(TWOP)
TIME IS ****.** NUMBER OF PEOPLE TO BUY PAPERS ****
        TERMINATE
        START     1000
        PUTSTRING ('  ')
        PUTSTRING ('  ')
        PUTSTRING (' OUTPUT FOLLOWS....')
        PUTSTRING ('  ')
        PUTPIC    LINES=3,AC1,N(ONEP),N(TWOP)
THE TIME TO SELL 1000 PAPERS WAS ****.** SECONDS
NUMBER OF PEOPLE WHO PURCHASED ONE PAPER  ****
NUMBER OF PEOPLE WHO PURCHASED TWO PAPERS ****
        END
```

The modifications consisted of adding a dummy transaction that arrived every 600 time units. This transaction caused the BPUTPIC to add a line of output to the screen.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 9

The ADVANCE Block

THE ADVANCE BLOCK

The ADVANCE block is used to "hold up" a transaction while service is being performed. There are two forms of it:

```
(a)      ADVANCE    A
(b)      ADVANCE    A,B
```

where *A* and *B* can be positive numbers (not necessarily integers) or variables. *A* represents the number of time units during which time advances while the transaction is held up. *B* represents the random variability of the holding time. In (a), the transaction will be held up for a time equal to operand *A*. So,

```
         ADVANCE    5
```

will hold up the transaction until 5 time units have passed. In (b), the transaction will be delayed by a time value between the interval (A – B) and (*A* + B). *The endpoints are not included.* The time returned by GPSS/H when it processes this block has 4 decimal places, and each possible time unit has an equal probability of being returned. Thus,

```
         ADVANCE    12,3
```

will hold a transaction until a time between the interval 9.0001 and 14.9999 time units has elapsed. Each of the possible times will be returned by the GPSS/H processor with equal probability. Chapter 17 shows how to use *any* statistical distribution in the ADVANCE block.

When a transaction enters an ADVANCE block, it is taken off the current events chain (CEC) and put on a chain known as the *future events chain* or FEC. It will remain there until the time is reached that is given by the operand in the ADVANCE block. It is then put back onto the current events chain for further movement through the system.

ADVANCE blocks have been used in many of the previous examples. They are one of the easiest GPSS/H blocks to understand and apply.

Example 9.1

There are 15 people in the class learning GPSS/H. They all start writing a program at the same time. It takes each person 12 ± 3 minutes to write the program. Only 25% of the programs run successfully the first time. When a program has an error, 8 ± 3.5 minutes are required to do the debugging. After the first (and any subsequent) debugging, the revised programs run successfully 35% of the time. How long does it take for the whole class to finish writing their programs?

For the first time in this book, enough GPSS/H has been covered for the reader to understand the complete program. The program listing is

```
◇         SIMULATE
◇         GENERATE  ,,,15        15 PEOPLE WRITE PROGRAM
◇         ADVANCE   12,3         TIME TO DO WRITING
◇         TRANSFER  .25,,DONE    25% WORK FIRST TIME
◇AGAIN    ADVANCE   8,3.5        DEBUG PROGRAM
◇         TRANSFER  .65,,AGAIN   35% WORK ON RE-DO
◇DONE     TERMINATE 1            PERSON LEAVES ROOM
◇         START     15
◇         PUTPIC    AC1
◇ WHEN ALL 15 ARE FINISHED, THE TIME IS  **.**
◇         END
```

The output from the program is as follows:

```
WHEN ALL 15 ARE FINISHED, THE TIME IS  63.70
```

What happened is as follows. At time $t = 0$, 15 student transactions were created. Each, in turn, entered the ADVANCE block where each was assigned a time to be on the future events chain. For example, suppose that the students are referred to as S1, S2, ..., S15, and suppose that the following times are those assigned by GPSS/H during the first pass of the simulation:

Student	Time
S1*	12.5667
S2	13.4338
S3	9.5581
S4	9.0809
S5*	10.1210
S6	12.7301
S7	11.3542
S8*	9.3433
S9	14.8162
S10	14.1931
S11*	13.2222
S12*	9.4538
S13	9.9444
S14	12.1288
S15	11.6687

The asterisks next to students S1, S5, S8, S11, and S12 refer to the fact that each was successful in running the program the first time through. At time t = 9.5581, student S3 is taken off the future events chain and enters the TRANSFER .25,,DONE block. Notice that this block was entered 15 times (as determined by START1 and TERMINATE 15). Since students S1, S5, S8, S11, and S12 were successful the first time, they were transferred to the DONE TERMINATE 1 block. The rest went to the AGAIN ADVANCE 8,3.5 block where they were again put on the future events chain. As each transaction came off that chain, it entered the TRANSFER .65,,AGAIN block. If a student was successful in running the program, that student's transaction was transferred to the block DONE TERMINATE 1. Eventually, all 15 students were finished. The time for all of the students to finish was found to be 63.70 time units.

The results from this example have to be treated with care. The simulation was run only one time with a particular set of random numbers. What would happen if different random numbers had been used? To answer this, the simulation was run 19 additional times, and 19 different sets of random numbers were used. A summary of the results of all 20 simulations follows:

Simulation Number	Time to Finish (minutes)
1	63.70
2	71.72
3	57.27
4	103.59
5	59.12
6	52.84
7	74.61
8	62.46
9	73.75
10	80.05
11	100.33
12	78.83
13	62.40
14	45.10
15	73.94
16	64.04
17	39.09
18	66.33
19	98.61
20	45.06

The times varied from a low of 39.09 to a high of 103.59. The average time was 68.64.

A CAUTION IN WRITING PROGRAMS

There is a caution to keep in mind with this and other GPSS/H programs. In this simulation, the transactions were all created at time $t = 0$. They all left at time $t = 0$ and were put on the future events chain via the ADVANCE block. The ADVANCE block always admits transactions. Suppose we had a different problem and were going to generate transactions at *various, sequential* times as given by

```
◇      GENERATE  12,3
```

Suppose the times for the first 4 of these transactions to enter the system are 11, 23, 35, and 44 time units. If the block after the GENERATE block will not allow a transaction to enter it (some blocks will only allow 1 transaction at a time to enter; others may not allow *any* transactions to enter, depending on a particular condition), what happens is that the transaction remains in the GENERATE block until the next block will allow it to enter. This delay may seem to be all right, but, because the transaction cannot leave the GENERATE block when it was originally scheduled to, the subsequent transactions also are delayed from leaving. Suppose that the third transaction cannot enter the next block but must remain in the GENERATE block for 5 time units. This condition means that, even though the transaction was originally scheduled to leave at $t = 35$, it cannot leave until $t = 40$. The effect of this delay on the fourth (and subsequent transactions) is to shift them all 5 time units forward (i.e., into the future) before they can leave. This programming approach is normally incorrect. When a transaction is scheduled to leave the GENERATE block, a block that will *always* accept the transaction should be provided by the programmer. One way around this is the following strategy:

```
◇      GENERATE  12,3
◇      ADVANCE   0
```

The ADVANCE block used here is a *dummy* block. It holds the transactions from the GENERATE block for zero time units. The only effect is to allow transactions to leave the GENERATE block at the times they were scheduled to leave. This strategy should be kept in mind for future programs.

Example 9.2

People arrive at an art exhibit every 4 ± 2 minutes. There are 3 rooms to view. Everyone goes to the entrance where it takes 5 ± 3 minutes to pick up a program and pay an entry fee. On entering, 80% of the people go to room A, and the rest to room B. Once a person misses a room, she or he does not go back to view it. When people leave room A, 75% go to room B, and the rest go to room C. Everyone who leaves room B will also go to room C. It takes 15 ± 3 minutes to view room A, 22 ± 6 minutes to view room B, and 12 ± 3 minutes to view room C. When a person leaves room C, he or she leaves the exhibit. Simulate for 100 people viewing the exhibit.

Solution

```
◇        □SIMULATE
◇        □GENERATE  □4,2          □PEOPLE ARRIVE AT THE EXHIBIT
◇        □ADVANCE   □5,3          □SPEND TIME IN THE ENTRANCE
◇        □TRANSFER  □.2,,ROOMB    □80% TO ROOM A
◇        □ADVANCE   □15,3         □VIEW ROOM A
◇        □TRANSFER  □.25,,ROOMC   □75% TO ROOM B
◇ROOMB   □ADVANCE   □22,6         □VIEW ROOM B
◇ROOMC   □ADVANCE   □12,3         □VIEW ROOM C
◇        □TERMINATE □1            □LEAVE THE EXHIBIT
◇        □START     □100
◇        □END
```

EXERCISES, CHAPTER 9

1. Consider Example 9.2 again. Add the code necessary to put the output on the screen, including the time it took for the 100 people to view the exhibit, the number of people who attended the exhibit (this value should be slightly more than 100), the number who viewed room A, and the number who viewed room B.
2. A mine has 6 self-loading trucks. Each loads in 10 ± 4 minutes. There is room for each to load so there is no waiting. Haulage takes 15 ± 2 minutes, dumping 1 ± 0.3 minutes, and the return trip takes 10 ± 3 minutes. Determine the production (i.e., the number of loads) for a shift of 320 minutes.
3. Trucks arrive at a repair shop every 25 ± 4 minutes. Each truck is inspected, which takes 4 ± 2 minutes. All trucks then are given minor service, which takes 30 ± 6 minutes. During this step, 20% are found to require major service, which takes 120 ± 40 minutes; the rest return to work. Simulate for 100 shifts of 480 minutes each. How many trucks are serviced during this time?

SOLUTIONS, CHAPTER 9

1. The modified program is as follows:

```
       SIMULATE
COMEIN GENERATE 4,2 PEOPLE ARRIVE AT THE EXHIBIT
       ADVANCE  5,3  SPEND TIME IN THE ENTRANCE
       TRANSFER   .2,,ROOMB 80% TO ROOM A
ROOMA  ADVANCE  15,3  VIEW ROOM A
       TRANSFER   .25,,ROOMC  75% TO ROOM B
ROOMB  ADVANCE  22,6  VIEW ROOM B
       ADVANCE   12,3  VIEW ROOM C
       TERMINATE     1
       START 100
       PUTSTRING     ('  ')
       PUTSTRING     ('  ')
       PUTPIC  LINES=4,AC1,N(COMEIN),N(ROOMA),N(ROOMB)
THE TIME FOR 100 PEOPLE TO VIEW THE EXHIBIT WAS ****.**
THE NUMBER WHO CAME IN WAS *****
THE NUMBER WHO VIEWED ROOM A WAS ****
THE NUMBER WHO VIEWED ROOM B WAS ****
END
```

2. The program to do the simulation is:

```
        SIMULATE
        GENERATE  ,,,6
BACK    ADVANCE   10,4      LOAD
        ADVANCE   15,2      HAUL
        ADVANCE   1,.3      DUMP
        ADVANCE   10,3      RETURN
UPTOP   TRANSFER  ,BACK
        GENERATE  320
        TERMINATE 1
        START     1
        PUTSTRING ('  ')
        PUTSTRING ('  ')
        PUTPIC    N(UPTOP)
PRODUCTION IS *** LOADS
        END
```

3. The program to do the simulation is:

```
        SIMULATE
        GENERATE  25,4
        ADVANCE   4,2     INSPECTION
        ADVANCE   30,6     MINOR SERVICE
        TRANSFER  .8,,BACK
        ADVANCE    120,40  MAJOR SERVICE
BACK    TERMINATE          BACK TO WORK
        GENERATE  100*480
        TERMINATE 1
        START     1
        PUTSTRING  ('  ')
        PUTSTRING  ('  ')
        PUTPIC     N(BACK)
TRUCKS  SERVICED ****
        END
```

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 10

QUEUE and DEPART Blocks

THE QUEUE BLOCK

Since GPSS/H is used so often for simulation of systems in which queues are formed at many places, it is natural to learn how the language handles queues. A discussion of queues and the mathematical theory associated with them can be found in any textbook on operations research. In fact, there are complete books devoted to this important topic. One thing becomes quite clear in the study of queueing theory: the number of queueing problems that have exact mathematical solutions is surprisingly small. This statement is especially true when dealing with a finite number of transactions, such as the case of transactions cycling though a system.

There are many cases when a transaction will be denied access to a block during a simulation. When a transaction is poised to use a facility that is already in use, the transaction is denied entry and has to remain in the block where it currently resides. In the system being simulated, this process gives rise to a queue. Such queues are commonly found in real-life situations, such as in a barbershop with only one barber, a checkout counter in a grocery store, a bank with many tellers, an airport with only a few runways. Often the purpose of the simulation study is to see where these queues form and how they might be eliminated or, perhaps, kept to a reasonable level. These queueing situations are handled in GPSS/H by the QUEUE block.

The QUEUE block never denies entry to a transaction, and this block can, in theory, contain any number of transactions. The normal form of the QUEUE block is quite simple. It is

```
          QUEUE     A
```

where the operand *A* is either a name that is not more than 8 characters or a positive integer. It also could be a variable. If a name is used, it is good programming practice always to use at least 3 characters to avoid unintentionally selecting a name that is reserved (e.g., C1).

Thus,

```
◇        □QUEUE     □1
◇        □QUEUE     □FIRST
◇        □QUEUE     □7
◇        □QUEUE     □ONE
◇        □QUEUE     □DUMP1
◇        □QUEUE     □STOPHERE
```

are examples of valid QUEUE blocks but

```
◇        □QUEUE     □-1
◇        □QUEUE     □LASTONEIN
```

are not valid.

Sometimes you will decide to use a number rather than a name for the QUEUE block's operand. If you choose this approach, the number cannot be arbitrary but will depend on the actual number of QUEUE blocks allowed in your system. Normally, at least 50 QUEUE blocks are allowed in most GPSS/H processors. Thus, if this is the maximum number allowed in your system, it would be all right to have

```
◇        □QUEUE     □27
◇        □QUEUE     □50
```

but not

```
◇        □QUEUE     □123
```

If you decide to use numbers in the QUEUE operands, simply remember to start numbering the QUEUE blocks with small numbers and you should not have any problem.

Whenever a QUEUE block is used, there automatically will be certain statistics as part of the .LIS file. These may have been observed when previous programs were run. Suppose the QUEUE block was specified by the operand WAIT. The output from the program might look as follows:

QUEUE	*MAXIMUM CONTENTS*	*AVERAGE CONTENTS*	*TOTAL ENTRIES*	*ZERO ENTRIES*	*PERCENT ZEROES*
WAIT	*3*	*0.312*	*264*	*90*	*34.1*

AVERAGE TIME/UNIT	*$AVERAGE TIME/UNIT*	*QTABLE*	*CURRENT CONTENTS*
5.665	*8.596*		*0*

The above actually is given in the output running across the screen. It is necessary to scroll to see it all. You may not want all the output, but GPSS/H gives it to you in the .LIS file regardless.

What each entry in the .LIS output file means is as follows:

QUEUE	WAIT this is the name of the queue as specified by the *A* operand.
MAXIMUM CONTENTS	The maximum contents of the queue WAIT at any time during the simulation was 3.
AVERAGE CONTENTS	At any time during the simulation, the contents in the queue WAIT averaged 0.312.
TOTAL ENTRIES	The number of transactions that entered the queue WAIT was 264.
ZERO ENTRIES	Of the 264 transactions that entered the queue WAIT, 90 of them immediately left and entered the next block.
PERCENT ZEROS	The percentage of ZERO ENTRIES—90 ZERO ENTRIES divided by 264 TOTAL ENTRIES—was 34.1%.
AVERAGE TIME/UNIT	For *all* the transactions that entered the queue WAIT, the average time in the block per unit transaction was 5.665 time units.
$AVERAGE TIME/UNIT	The average time in the queue WAIT for only the transactions that were actually delayed and held in it was 8.596 time units.
QTABLE	Chapter 19 shows how to construct histograms of various parameters associated with the simulation. One of these parameters is called a QTABLE. If one had been used in the simulation, its name would be here.
CURRENT CURRENTS	The contents of the queue WAIT at the end of the simulation is 0.

The preceding output items are all attributes associated with having a QUEUE block. In fact, they are standard numerical attributes (SNAs). There are reserved names for SNAs that deal with queues. In the following, (name) is a queue name (may include numbers if a letter is the first character), and *n* refers to a number:

SNA	Meaning
Q(name) or Q*n*	current queue content
QA(name) or Qa*n*	average queue contents
QC(name) or Qc*n*	queue entry count
QM(name) or Qm*n*	maximum queue content
QT(name) or Qt*n*	average time spent in the queue of *all* entries
QX(name) or Qx*n*	average time spent in the queue excluding the zero entries
QZ(name) or Qz*n*	zero entries

These SNAs can be used in the program as operands. For example, one could have

```
◇         ADVANCE    QM(WAIT)
```

The transaction entering the ADVANCE block would be put on the FEC for a time given by the maximum queue length (QM) at the QUEUE WAIT. The most common reason for using a QUEUE block is to gather the statistics of how long transactions are delayed before a facility is available for their use. Therefore, it is not always necessary to have a QUEUE block just because a queueing situation is to take place.

It is possible to have a second (*B*) operand with the QUEUE block such as

```
◇        QUEUE      FIRST,2
◇        QUEUE      WAIT,3
```

The *B* operand must be a positive integer. If it is a negative number, a compiling error results. If it is a positive real number with a decimal point, it will be truncated to an integer, and a warning message will be printed on the screen. If a *B* operand is used, it will affect the statistics of the QUEUE block as the *B* operand will cause the TOTAL ENTRIES count to be increased by the value of *B* (not by 1) and the CURRENT CONTENTS also to be increased by the value of *B*.

Should you ever decide to use a two-operand QUEUE block, you must be very careful to interpret your results accordingly. Actually, examples of a two-operand QUEUE block are very rare; they might be found in code that describes extremely complex manufacturing systems. No such examples will be encountered in this book.

Another characteristic of QUEUE blocks is the fact that a transaction can be in more than one QUEUE block at the same time. This may seem strange, but such situations occur in real life. Consider a large grocery store where a person has to take a number to purchase meat. The same person can elect to also take a number to purchase vegetables while waiting for the first number to be called. Thus, the person is in two queues at the same time. There will be occasions when, for the purpose of gathering statistics, we will use the fact that a transaction can be in more than one QUEUE block at the same time. In fact, a transaction can be in even more than two QUEUE blocks at the same time. The maximum number of QUEUE blocks that a transaction can be in at the same time is dependent on the particular processor, but is generally around 5.

THE DEPART BLOCK

If a transaction is in a QUEUE block, it must eventually leave this block. The DEPART block allows this action and, as the twin to the QUEUE block, has the same operand. Thus, referring to the previous examples of the QUEUE block, the following would be the corresponding DEPART blocks:

```
◇        DEPART     1
◇        DEPART     FIRST
◇        DEPART     7
◇        DEPART     ONE
◇        DEPART     DUMP1
```

and, in free format,

```
◇        DEPART     STOPHERE
```

The DEPART block will not be immediately after the QUEUE block but *must* appear in the program. (If it were immediately after the QUEUE block, the QUEUE block would give meaningless statistics as the transactions would immediately enter and leave both blocks.) It usually appears after one or two other blocks. These other blocks are the ones that, for one reason or other,

cause a queue to form. Just as with the QUEUE block, it is possible to have a second operand,

```
◇        DEPART     NAME,2
```

In this case, the current content of the QUEUE NAME is decreased by 2. If the transaction was a zero entry to the QUEUE, the zero-entry counter is incremented by 2. As with the second operand for the QUEUE block, use of this operand is very rare.

Example 10.1

Customers arrive in Joe's barbershop at a uniform rate of 1 customer every 18 ± 5 minutes. Joe can cut hair at the rate of 16 ± 4 minutes. Simulate for 8 hours.

Solution

The solution to this situation involves two blocks that we haven't had yet, SEIZE and RELEASE. Both of these are discussed in the next chapter.

```
◇        SIMULATE
◇        GENERATE   18,5        CUSTOMERS ARRIVE
◇        QUEUE      WAITAREA    JOIN THE QUEUE
◇        SEIZE      BARBER      BARBER BECOMES FREE
◇        DEPART     WAITAREA    LEAVE THE QUEUE
◇        ADVANCE    16,4        GET HAIRCUT
◇        RELEASE    BARBER      FREE THE BARBER
◇        TERMINATE              LEAVE THE SHOP
◇        GENERATE   480         TIMER TRANSACTION ARRIVES
◇        TERMINATE  1           STOP SIMULATION
◇        START      1           SIMULATION BEGINS
◇        PUTPIC     LINES=6,QM(WAIT),QA(WAIT),_
             QC(WAIT),QZ(WAIT),QT(WAIT),QX(WAIT)
 Q MAX  ***
 Q AVERAGE  ***.**
 Q COUNT  ****
 Q ZERO ENTRIES  ***
 Q TIME FOR ALL ENTRIES  ***.**
 Q TIME FOR NONZERO ENTRIES  ***.**
◇        END
```

For 1 run of the simulation program with 1 set of random numbers, the output will be

```
Q MAX  1
Q AVERAGE  0.04
Q COUNT   26
Q ZERO ENTRIES  17
Q TIME FOR ALL ENTRIES  0.74
Q TIME FOR NONZERO ENTRIES  2.16
```

During the 8 simulated hours, 26 customers entered the store; 17 of them did not have to wait for the barber to be free. The average contents of the queue was only 0.04 and the maximum contents of the queue was only 1. From the .LIS file, it can be seen that the barber was busy 85.8% of the time.

EXERCISES, CHAPTER 10

1. Refer to Example 10.1. Suppose that the first customer arrives exactly when the shop opens. Also, let the time to cut hair be 16 ± 7.5 minutes and the arrival rate be 18 ± 9 minutes. Do the statistics change much?
2. Refer to Example 10.1. Suppose that the time to give a haircut is 17 ± 4. How do the statistics change?

SOLUTIONS CHAPTER 10

1. The modified program is as follows:

```
          SIMULATE
          GENERATE    18,9
          QUEUE       WAIT
          SEIZE       BARBER
          DEPART      WAIT
          ADVANCE     16,7.5
          RELEASE     BARBER
          TERMINATE
          GENERATE    480
          TERMINATE     1
          START         1
          PUTPIC     LINES=6,QM(WAIT),QA(WAIT),_
                     QC(WAIT),QZ(WAIT),QT(WAIT),QX(WAIT)
QMAX                                ***
QAVERAGE                            ***.**
QCOUNT                             ****
Q ZERO ENTRIES                      ***
Q TIME FOR ALL ENTRIES              ***.**
Q TIME FOR NON ZERO ENTRIES         ***.**
          END
```

Now the output is changed as follows:

```
QMAX                           1
QAVERAGE                       0.18
QCOUNT                        26
Q ZERO ENTRIES                12
Q TIME FOR ALL ENTRIES         3.45
Q TIME FOR NON ZERO ENTRIES    6.31
```

As can be seen, there is a considerable change in the average times a customer remains in the shop.

2. The new output is as follows.

```
Q MAX                          1
Q AVERAGE                      0.11
Q COUNT                       26
Q ZERO ENTRIES                12
Q TIME FOR ALL ENTRIES         1.97
Q TIME FOR NON ZERO ENTRIES    3.65
```

As can be seen by examining this output and the previous one, the spread of a distribution can have more effect on the results of the simulation than the mean value. When the time to give service was changed from a mean of 16 to a mean of 17, this did not affect the results as much as when the spread was changed as in Exercise 10.1.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 11

SEIZE and RELEASE Blocks

THE SEIZE BLOCK

In GPSS/H, a single server is called a "facility." This server might be a barber giving a haircut, a bank clerk who waits on customers, or a checkout clerk in a grocery store. In a mine, a server might be a shovel loading a truck or a crusher where only 1 truck can dump at a time. In order for a transaction to use a facility, a SEIZE block is used. One form of the SEIZE block is

```
          SEIZE      A
```

where the operand *A* is generally either a number or name, but can be a variable. Thus,

```
          SEIZE      1
          SEIZE      ONE
          SEIZE      DUMP
          SEIZE      BARBER
          SEIZE      33
```

are examples of the SEIZE block. If a number is used for the operand of the SEIZE block, the number must be less than the number of SEIZE blocks allowed by your processor. If you always remember to number the blocks starting with small numbers, you should not have a problem.

When a transaction enters a SEIZE block, no other transaction can enter until the transaction leaves the SEIZE block. Transactions attempting to enter a SEIZE block that is already being used must (normally) remain in the block they are in. For example, consider the blocks

```
          GENERATE   10
          QUEUE      WAITAREA
          SEIZE      PAYPHONE
          DEPART     WAITAREA
          ADVANCE    25
```

Here a transaction is generated at a time of $t = 10$. It is moved to the QUEUE block and immediately attempts to use the facility PAYPHONE. Since PAYPHONE is not being used, the transaction seizes the facility and then enters the ADVANCE block where the transaction is put in the future events chain until a time of $t = 35$ (i.e., 10 + 25). A second transaction is generated at time $t = 20$. It enters the QUEUE WAIT block and attempts to enter the SEIZE PAYPHONE block. Since the facility PAYPHONE is in use, the second transaction is held in the QUEUE block until PAYPHONE is free. Now, suppose that you did not have the QUEUE WAIT block. Instead, suppose you had

```
◇        GENERATE   10
◇        SEIZE      PAYPHONE
◇        ADVANCE    25
```

The second transaction would have been held up in the GENERATE block until time $t = 35$. The effect of the lack of a QUEUE WAIT block would be for the GPSS/H processor to offset the time at which the third transaction leaves the GENERATE block from $t = 30$ to $t = 50$. (Draw a time diagram to convince yourself of this.) Since this effect is *not* what we want, either a QUEUE block or a dummy ADVANCE block should be used. Code with a dummy ADVANCE block would look as follows:

```
◇        GENERATE   10
◇        ADVANCE
************************************
•   NOTE:  YOU COULD ALSO HAVE     *
•      ADVANCE       0             *
************************************
◇        SEIZE      PAYPHONE
◇        ADVANCE    25
```

The ADVANCE block with no operand is taken to be ADVANCE 0.

Whenever a SEIZE block is used, certain statistics automatically are printed out at the end of the simulation. To illustrate the statistics, recall Example 10.1 that simulated the barbershop. Suppose that the barber cuts hair in 16 ± 4 minutes, and customers arrive at the rate of 18 ± 5 minutes. The program to simulate this situation for 8 hours is

```
◇        SIMULATE
◇        GENERATE   18,5         PEOPLE ENTER SHOP
◇        QUEUE      WAITAREA     TAKE SEAT IN WAITING AREA
◇        SEIZE      BARBER       IF BARBER FREE, BEGIN HAIRCUT
◇        DEPART     WAITAREA     LEAVE SEAT IN WAITING AREA
◇        ADVANCE    16,4         RECEIVE HAIRCUT
◇        RELEASE    BARBER       HAIRCUT OVER, BARBER IS FREE
◇        TERMINATE               LEAVE THE SHOP
◇        GENERATE   480
◇        TERMINATE  1
◇        START      1
◇        END
```

The output associated with the SEIZE block as found in the .LIS file is as follows:

```
           --AVG-UTIL-DURING--
FACILITY  TOTAL   AVAIL  UNAVL     ENTRIES    AVERAGE      CURRENT     PERCENT     SEIZING     PREEMPTING
           TIME    TIME   TIME                TIME/XACT     STATUS      AVAIL        XACT         XACT
  BARBER  0.858                         26      15.847      AVAIL                     27
```

The output is interpreted as follows:

FACILITY	The name of the facility was BARBER.
TOTAL TIME	The facility was busy 85.8% of the time the simulation ran.
AVAIL TIME	It is possible to "shut down" a facility and make it unavailable. This statistic would show the amount of time the facility was available.
UNAVL TIME	The time the facility was not available.
ENTRIES XACT	There were 26 entries.
AVERAGE	The average time the SEIZE block was used by a TIME/XACT transaction was 15.847. The time to give a haircut was 16 ±4 minutes; the mean of this distribution is 16. (The AVERAGE is 15.847 [as shown in the output]. The actual average of 16 ±4 is 16. The reason for the difference between 15.847 and the actual average of 16 is that GPSS/H uses random numbers to determine the times. As more random numbers are sampled, the closer the AVERAGE will be to the actual average of 16. There is also another reason for the difference: the simulation was stopped when a transaction was still being seized, and so the average time for all transactions using the SEIZE block will not include this time but will include the fact that this transaction did SEIZE the block.)
CURRENT STATUS AVAIL	The facility is currently available.
PERCENT AVAIL	Since the facility was never made unavailable, there is no entry here.
SEIZING XACT	If the facility was in use at the end of the program, XACT—the transaction number seizing it—would be given here.

Just as with the QUEUE block, the preceding are SNAs associated with the SEIZE block. Each is referred to by a special reserved name. In the following, (name) is a facility name (may include numbers if a letter is the first character), and *n* refers to a number:

SNA	Meaning
F(name) or F*n*	value of facility is 1 if the facility is currently being used; otherwise, its value is 0
FC(name) or FC*n*	number of times the facility has been siezed
FR(name) or FR*n*	utilization of the facility in parts per thousand
FT(name) or FT*n*	average holding time
FNU(name) or FNU*n*	true (value = 1) if facility is not in use
FU(name) or FU*n*	true (value = 1) if facility is in use
FS(name) or FS*n*	true (value = 1) if facility can be seized
FNS(name) or FNS*n*	true (value = 1) if facility cannot be seized

There are other SNAs associated with facilities, but these will not be used in this book. Note that the output from the program gives the utilization as a decimal, but the SNA FR is in parts per thousand. Thus, if one had used the block

```
◇        ADVANCE    FR(MACH1)
```

and the utilization of the facility MACH1 was 0.432 when the transaction entered the ADVANCE block, the transaction would be placed on the FEC for 432 time units.

THE RELEASE BLOCK

When a facility is used by a transaction via the SEIZE block, the facility must be freed eventually for other transactions to use it. Freeing the facility is accomplished by means of the "twin" of the SEIZE block, the RELEASE block. Some forms are

```
◇        RELEASE    JOE
◇        RELEASE    1
◇        RELEASE    BARBER
◇        RELEASE    CAR1
```

At this point you may wish to go back to some of the previous exercises and examine the output whenever a facility is used in the program.

It is very common in GPSS/H programs to have a sequence of blocks such as

```
◇        QUEUE      HERE
◇        SEIZE      THERE
◇        DEPART     HERE
◇        ADVANCE    20,3
◇        RELEASE    THERE
```

This sequence should be examined carefully and understood as it is repeated a great many times in GPSS/H programs.

We are now in a position to fully understand nearly all of the programs we have written in the previous chapters. The next few examples will enable us to use what we have learned so far. Each should be carefully studied and understood.

Example 11.1

Cars come to a garage for repairs. There are two types of repairs: minor and major. There is only one mechanic who does both types of repairs. Of the cars that enter, 70% need only minor repairs, and the rest are in for major repairs. Cars arrive every 28 ± 7 minutes. Major repairs take 45 ± 15 minutes whereas minor repairs take only 18 ± 6 minutes. The single mechanic is claiming that he is overworked. His union defines being overworked as working more than 85% of the time. Is he justified in his claim? Simulate for 5 days of 480 minutes each. Ignore the fact that the worker leaves at the end of each shift and the effect of weekends.

Solution

```
◇         SIMULATE
◇         GENERATE  28,7          CARS ARRIVE FOR REPAIRS
◇         TRANSFER  .70,,MINOR    70% NEED MINOR REPAIRS
◇         QUEUE     WAIT          JOIN QUEUE
◇         SEIZE     MECH          IS MECHANIC FREE?
◇         DEPART    WAIT          YES, LEAVE QUEUE
◇         ADVANCE   45,15         MAJOR REPAIR TAKES PLACE
◇         RELEASE   MECH          FREE THE MECHANIC
◇         TERMINATE               LEAVE GARAGE
◇MINOR    QUEUE     WAIT          JOIN QUEUE
◇         SEIZE     MECH          IS MECHANIC FREE?
◇         DEPART    WAIT          YES, LEAVE QUEUE
◇         ADVANCE   18,6          MINOR REPAIR TAKES PLACE
◇         RELEASE   MECH          FREE MECHANIC
◇         TERMINATE               LEAVE GARAGE
◇         GENERATE  480*5         TIMER TRANSACTION
◇         TERMINATE 1
◇         START     1
◇         PUTPIC    LINES=2,FR(MECH)/10,FC(MECH)
 THE MECHANIC WAS BUSY  ***.**%  OF THE TIME
 THE NUMBER OF CARS TO COME IN WAS  ***
◇         END
```

The output will be

```
THE MECHANIC WAS BUSY  94.0%  OF THE TIME
THE NUMBER OF CARS TO COME IN WAS  84
```

The results are interpreted as follows. During the 5 days of simulated time, 84 cars came in for repairs. The mechanic was busy 94% of the time. This percentage is beyond the 85% called for, so he has a legitimate complaint of being overworked.

Example 11.2

The owner of a small gold mine is wondering if he has the right number of trucks to haul the ore. Figure 11.1 gives a sketch of the operation. Trucks are loaded by a single shovel and then travel to the processing plant where they dump and then cycle back to the shovel. Only 1 truck at a time can be loaded, but at the processing plant there is no such limitation since the trucks dump the ore into a pile. It takes 3.5 ± 1.25 minutes to load a truck, 6 ± 2.75 minutes to haul to the dump, 2.1 ± 0.4 minutes to dump the ore, and 4.8 ± 1.8 minutes to return. Financial data associated with this operation are as follows:

Item	Cost
Truck driver's salary	$15.75/hour
Cost of running the shovel, dump, etc.	$275 per 8-hour day
Profit per load (after all other expenses)	$25.50

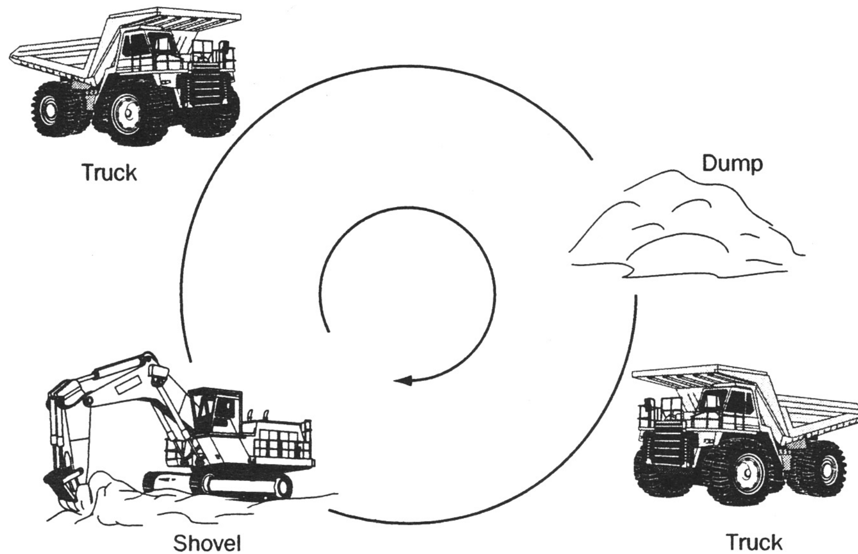


FIGURE 11.1 Sketch of mine and trucks.

Determine the correct number of trucks to have in the mine. The GPSS/H program that was used to obtain a solution for this problem is as follows:

```
◇        SIMULATE
◇        GENERATE  ,,,2          PUT TRUCKS IN THE MINE
◇UPTOP   QUEUE     WAIT          QUEUE AT THE SHOVEL
◇        SEIZE     SHOVEL        USE THE SHOVEL
◇        DEPART    WAIT          LEAVE THE QUEUE
◇        ADVANCE   3.5,1.25      LOAD A TRUCK
◇        RELEASE   SHOVEL        FREE THE SHOVEL
◇        ADVANCE   6.5,2.75      TRAVEL TO DUMP
◇        ADVANCE   2.1,.4        DUMP A LOAD OF ORE
◇        ADVANCE   4.8,1.8       RETURN TO SHOVEL
◇BACK    TRANSFER  ,UPTOP        JOIN QUEUE AGAIN
◇        GENERATE  480           SIMULATE FOR A SINGLE SHIFT
◇        TERMINATE 1
◇        START     1
◇        PUTPIC    N(BACK),FR(SHOVEL)/10,QA(WAIT)
 LOADS PER SHIFT  ***   UTIL OF SHOVEL  ***.**%  AVG QUEUE  ***.**
◇        END
```

The program was run repeatedly for 2 trucks, then 3 trucks, etc., up to 7 trucks. The results of the 6 simulations gave the following:

Number of Trucks	Loads per Shift	Percent Utilization of the Shovel	Average Queue
2	56	40.9	0.031
3	82	60.2	0.104
4	107	77.8	0.241
5	127	92.6	0.535
6	136	99.5	1.200
7	136	100.0	2.194

All that is needed from the simulation results is the loads per shift, but it is instructive to examine the other data. With only 2 trucks, the shovel is busy only 40.9% of the time. The average queue length is only 0.031. As the number of trucks in the mine is increased, the utilization of the shovel increases until it is working 100% of the time. The addition of a 7th truck has no effect on operation of the mine other than to add another truck to the queue. How does the 7th truck affect profit? In order to answer this question, the cost data need to be used. The following table gives the results of the cost and profit calculations:

Number of Trucks	Loads per Shift	Cost of Drives	Fixed Costs	Profit
2	56	$252	$275	$339
3	82	$378	$275	$620
4	107	$504	$275	$875
5	127	$630	$275	$1058
6	136	$756	$275	$1083
7	136	$882	$275	$954

The optimum number of trucks to have is 6 for maximum daily profit. Notice, however, that there would not be much of a difference in profit if 5 trucks were used. Of course, this situation is oversimplified in that it ignores trucks needing maintenance and repairs as well as possible driver absences.

EXERCISES, CHAPTER 11

1. A factory that formerly produced only widgets is branching out into the production of squidgets. To make each squidget, a person needs 30 ± 8 minutes to assemble various parts. Then the squidgets need to be fired. There is only one kiln, and only one squidget can be fired at a time. Firing takes 8 ± 3 minutes per squidget. Each squidget produced earns a tidy profit of $6.00. The kiln costs $40 per day whether it is fully utilized or not (fixed costs). The workers are paid $5 per hour. How many workers should be hired for maximum profit from the new line of squidgets?
2. In exercise 1, the workers initially began assembling a squidget. Suppose the following were the initial conditions (4 workers total): 3 assemblers are just beginning to assemble, and 1 kiln operator is just beginning to use the kiln. Change the program written for exercise 1 to include the above.
3. Change exercise 1 to reflect the following initial conditions for 4 workers: 1 assembler is beginning to assemble, 1 assembler has 10 minutes to go before completing an assembly, 1 kiln operator has 3 minutes to go before finishing with the kiln, and 1 assembler is waiting to hand over an assembled squidget to the kiln operator.
4. Two types of customers arrive at Joe's barbershop. The first type want only a haircut. They come every 35 ± 10 minutes. The second type want both a haircut and a shave. They arrive every 60 ± 20 minutes. It takes Joe 18 ± 6 minutes to give a haircut and 10 ± 2 minutes for a shave. Construct a model of the shop. Run it for 20 days of 8 hours straight (from Schriber, 1974).

a. Determine whether Joe is working too hard. (Working too hard is defined by the union as working more than 85% of the time.)

b. Suppose Joe decides to give preference to customers who want only a haircut. How does this change the situation?

c. Add other QUEUE blocks to the problem to gather statistics about the various queues that form in the barbershop.

5. Three types of mechanics arrive at a tool crib to check out tools. Only one clerk works at the crib. The arrival times and service times (both in minutes) are as follows:

Type	Distribution of Arrival Time	Distribution of Service Time
1	30 ± 10	12 ± 5
2	20 ± 8	6 ± 3
3	15 ± 5	3 ± 1

a. Model the tool-crib activity for 20 straight shifts. Each shift lasts for 480 minutes. Determine the average number of mechanics waiting to check out tools.

b. Suppose that preference is given to mechanics who take the least time for service, in this case, type 3 mechanics. Change the program to determine if the solution changes.

6. In an efficient shop, it can be shown that it is more efficient for the workers to perform tasks that can be finished in the shortest time, assuming that the cost/benefit ratio is the same. This exercise will illustrate this general concept.

A tool crib receives two types of requests for service. These are called type 1 requests and type 2 requests. There is a single worker to handle the requests. The interarrival and mean service times (both in seconds) for each are as follows:

Service	Arrival Times	Service Times
type 1	520 ± 360	340 ± 90
type 2	250 ± 100	70 ± 40

Write the GPSS/H program with no priority for service and then with priority given to type 2 service. Suppose that the people requesting service earn $12 per hour. Determine the savings each day by having a priority system. Also, run the program with type 1 service having priority. Each day is 480 minutes.

SOLUTIONS, CHAPTER 11

1. The GPSS/H program that can be used to solve the problem follows. This is for 3 workers. You will have to run the program for 4, 5, 6, etc. workers. Doing the computations after each run yields the following:

```
workers   profit
   3        60
   4        79
   5        93
   6        80
   7        44

          SIMULATE
          GENERATE     ,,,3        PROVIDE 3 WORKERS
UPTOP     ADVANCE      30,8        IT TAKES 30 ± 8 MIN TO WORK
          SEIZE        FIRE        USE THE FURNACE
          ADVANCE      8,3         FIRE A SQUIDGET
          RELEASE      FIRE        FREE THE FURNACE
BACK      TRANSFER     ,UPTOP      GO BACK TO DO ANOTHER JOB
          GENERATE     4800        SIMULATE FOR 10 SHIFTS
          TERMINATE    1           STOP SIMULATION
          START        1           START THE SIMULATION
        PUTPIC      N(BACK)
NUMBER MADE ****
            END                    OF COMPILING
```

2. The changes to the program are as follows:

```
          GENERATE     ,,,1        PROVIDE ONE ASSEMBLER
          TRANSFER     ,OVEN       SEND HIM TO THE OVEN
          GENERATE     ,,,3        PROVIDE 3 ASSEMBLERS
BACK      ADVANCE      30,5        ASSEMBLE A SQUIDGET
          OVEN SEIZE   OVEN        USE THE OVEN
          ADVANCE      8,2         FIRE A SQUIDGET
          RELEASE      OVEN        FREE THE OVEN
          TRANSFER     ,BACK
```

3. The complete program is:

```
          SIMULATE
          GENERATE     ,,,1        ONE TO BEGIN ASSEMBLING
          TRANSFER     ,UPTOP      SEND TO ASSEMBLE AREA
          GENERATE     ,,,1        ONE IS ASSEMBLING
          ADVANCE      10          THIS WILL TAKE 10 MINUTES
          TRANSFER     ,DOWN1      SEND TO OVEN
          GENERATE     ,,,1        ONE IS USING OVEN
          SEIZE        FIRE        USE THE OVEN
          ADVANCE      3           WAIT FOR THREE MINUTES
          RELEASE      FIRE        FREE THE OVEN
          TRANSFER     ,UPTOP      BACK TO DO ANOTHER
          GENERATE     ,,,1        ONE TO WAIT FOR OVEN
          TRANSFER     ,DOWN1      GO TO THE OVEN
UPTOP     ADVANCE      30,8        IT TAKES 30 +/- 8 MINUTES TO WORK
DOWN1     SEIZE        FIRE        USE THE FURNACE
          ADVANCE      8,3         FIRE A SQUIDGET
          RELEASE      FIRE        FREE THE FURNACE
BACK      TRANSFER     ,UPTOP      GO BACK TO DO ANOTHER JOB
          GENERATE     4800        SIMULATE FOR 10 SHIFTS
          TERMINATE    1           STOP SIMULATION
          START        1           START SIMULATION
        PUTPIC      N(BACK)
NUMBER MADE ***
          END                    OF COMPILING
```

4.

```
             SIMULATE
             GENERATE    35,10     HAIRCUT CUSTOMERS ARRIVE
             QUEUE       JOEQ      SIT IN THE CHAIRS
             SEIZE       JOE       SEIZE POOR JOE
             DEPART      JOEQ      LEAVE THE CHAIR
             ADVANCE     18,6      GET THE HAIRCUT
             RELEASE     JOE       FREE JOE
             TERMINATE             LEAVE
             GENERATE    60,20     OTHER CUSTOMERS ARRIVE
             QUEUE       JOEQ      SIT IN THE CHAIRS
             SEIZE       JOE       SEIZE GOOD OLD JOE
             DEPART      JOEQ      LEAVE THE CHAIR
             ADVANCE     10,2      GET SHAVE
             ADVANCE     18,6      GET HAIRCUT
             RELEASE     JOE       FREE JOE
             TERMINATE             LEAVE SHOP
             GENERATE    9600      TIMER TRANSACTION
             TERMINATE   1
             PUTPIC      FR(JOE)/10
JOE WORKED ***.**% OF THE TIME
             START       1
```

5. a.

```
             SIMULATE
             GENERATE    30,10     TYPE 1 ARRIVES
             QUEUE       WAIT      IN QUEUE
             SEIZE       CLERK     USE THE CLERK
             DEPART      WAIT      LEAVE THE QUEUE
             ADVANCE     12,5      CLERK GETS TOOL
             RELEASE     CLERK     FREE THE CLERK
             TERMINATE             LEAVE THE CRIB AREA
             GENERATE    20,8      TYPE 2 ARRIVES
             QUEUE       WAIT      IN QUEUE
             SEIZE       CLERK     USE THE CLERK
             DEPART      WAIT      LEAVE THE QUEUE
             ADVANCE     6,3       CLERK GETS TOOL
             RELEASE     CLERK     FREE THE CLERK
             TERMINATE             LEAVE THE CRIB AREA
             GENERATE    15,5      TYPE 3 ARRIVES
             QUEUE       WAIT      IN QUEUE
             SEIZE       CLERK     USE THE CLERK
             DEPART      WAIT      LEAVE THE QUEUE
             ADVANCE     3,1       CLERK GETS TOOL
             RELEASE     CLERK     FREE THE CLERK
             TERMINATE             LEAVE THE CRIB AREA
             GENERATE    480       ONE SHIFT PASSES
             TERMINATE   1
             START       20      DAYS
             PUTPIC      QA(WAIT)
AVERAGE NUMBER IN THE QUEUE WERE ***.**
             END
```

b.

```
The block   GENERATE      15,5   is changed to
            GENERATE      15,5,,,1
No other changes are needed.
```

6. The first program is:

```
          SIMULATE
          GENERATE     520,360
          QUEUE        WAIT
          SEIZE        MECH
          DEPART       WAIT
          ADVANCE      340,90
          RELEASE      MECH
          TERMINATE
          GENERATE     250,100
          QUEUE        WAIT
          SEIZE        MECH
          DEPART       WAIT
          ADVANCE      70,40
          RELEASE      MECH
          TERMINATE
          GENERATE     480*60
          TERMINATE    1
          START        1
          PUTPIC       QA(WAIT)
 AVERAGE WAIT ***.**
          END
```

Give an average queue length of 1.67 when running this. Since the workers are making $12/hr. this represents a loss of $160.32 per 8-hour shift. When the program is run with a priority given to type 2 request, the average number in the queue drops to .81. This represents a cost per shift of $77.76 for a savings of $82.56. When the simulation is run with the type 1 requests having the higher priority, the average queue is 1.84.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 12

ENTER and LEAVE Blocks

MULTIPLE SERVERS—THE ENTER BLOCK

It often happens that the system being studied has multiple servers that are identical. Transactions that attempt to use these will be denied access if all the servers are busy. They do not wait at each server but are held on the CEC in the previous block until one of the servers is free. Figure 12.1 illustrates this situation.

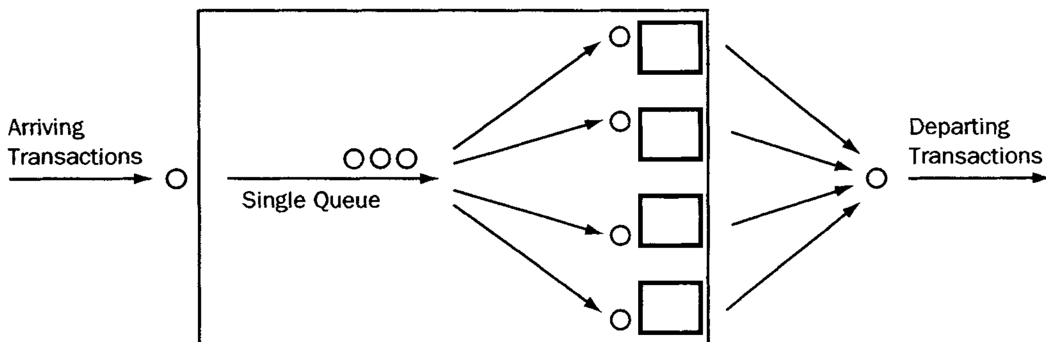


FIGURE 12.1 Sketch of multiple servers.

Examples might be the multiple berths for loading or unloading ships in a port, two barbers in a shop, six tellers in a bank, etc. GPSS/H handles these by means of a combination of a block and a statement. The block is used in the program for the transaction to actually use one of the parallel servers. The statement tells the processor how many of these servers are available. The block for the transaction in the programs is the ENTER block. One form of the ENTER block is

```
◇        ENTER     A
```

where the operand A can be a name or a number. If it is a number, it cannot exceed the maximum number of such ENTER blocks allowed in your GPSS/H system. It is also possible to have this operand be a variable.

Examples of the ENTER block are

```
◇          ENTER     DUMP
◇          ENTER     2
◇          ENTER     TWO
```

The operands DUMP, 2, and TWO refer to the names of the multiple servers, or "storages," as we shall now refer to them. There is no confusion in using the storages named DUMP and TWO, but care should be exercised with the one named "2." This name does *not* refer to the number of storages associated with the block

```
◇          ENTER     2
```

The number of storages for each ENTER block is specified by a separate statement. It is quite possible that the number of storages associated with each of the 3 examples of the ENTER block might be 4 for DUMP, 3 for "2," and 4 for TWO. In the next section, we will learn how to specify these numbers. It is possible to have a second or B operand with the ENTER block such as

```
◇          ENTER     HARBOR,2
```

In this case, $B = 2$ indicates that 2 storages are used. This example might represent a large ship entering a harbor, and 2 tugboats (in this case, the tugboats are the storages) are needed to guide it into port. Most of the time, only 1 of the storages is taken each time a transaction uses the ENTER block. However, there will be times when the second operand will come in very handy.

DEFINING THE NUMBER OF MULTIPLE SERVERS—THE STORAGE STATEMENT

Normally, when there are to be multiple servers, the number of these must be specified. This procedure is done by the STORAGE statement. There are two forms of it. The form may look a bit strange, but, if the programmer remembers that it is used in connection with the ENTER block, it is easy to write. Suppose there are to be 3 barbers in a shop. The customer will select one of them via the ENTER block.

```
◇          ENTER     BARBER
```

How the computer knows that there are 3 barbers is specified by the STORAGE statement, which must be placed before the ENTER block. Normally, for convenience, it is placed near the top of the program before any GENERATE block. It can have 2 forms, as follows, but their results are identical:

```
◇          STORAGE   S(BARBER),3   (a)
◇BARBER    STORAGE   3             (b)
```

The first STORAGE form is preferred in this book since more than one storage can be defined in a single line. If either one of these STORAGE statements is used, however, then when the block

```
◇         ENTER     BARBER
```

is encountered, the transaction will be able to enter it if there are fewer than 3 transactions in it already. If the transaction cannot move forward, it is held on the CEC until a later scan by the GPSS/H processor shows that the transaction can be moved.

The reason that STORAGE form (a) is used more often than (b) is because it is possible to have more than one STORAGE defined via the (a) form, as follows:

```
◇         STORAGE   S(BARBER),3/S(JOE),12/S(BILLY),7
```

This statement shows that the storage of BARBER is 3, that of JOE is 12, and BILLY is 7. If numbers, however, are used for storages, the form of the STORAGE statement may be somewhat simplified:

```
◇         STORAGE   S1,2/S4,7
```

Nevertheless, one can write the above as

```
◇         STORAGE   S(1),2/S(4),7
```

Since an operand can also be a number, the second STORAGE statement defines the storage of 1 to be 2 and that of 4 to be 7.

Whenever you use the ENTER block for parallel entities, you will obtain certain output when the program is over. Consider the following example.

Example 12.1

A barbershop has two identical workers. They can cut hair at a rate of 1 haircut every 13 ± 5.5 minutes. Customers arrive every 7 ± 2.6 minutes. Simulate for an 8-hour day. Determine how busy each barber is. The program to do this is

```
◇         SIMULATE
◇         STORAGE   S(BARBER),2  PROVIDE TWO BARBERS
◇         GENERATE  7,2.6        CUSTOMERS ARRIVE
◇         QUEUE     WAIT         FOR BARBER
◇         ENTER     BARBER       USE A BARBER
◇         DEPART    WAIT         LEAVE WAITING AREA
◇         ADVANCE   13,5.5       GET HAIRCUT
◇         LEAVE     BARBER       FREE A BARBER
◇         TERMINATE              LEAVE SHOP
◇         GENERATE  480          TIMER TRANSACTION ARRIVES
◇         TERMINATE 1
◇         START     1
◇         END
```

Solution

The simulation results, as always, in a .LIS file. A portion of the .LIS file is

```
RELATIVE CLOCK: 480.0000   ABSOLUTE CLOCK: 480.0000
BLOCK CURRENT     TOTAL
1                  68
2                  68
3                  68
4                  68
5        2         68
6                  66
7                  66
8                   1
9                   1

 --AVG-UTIL-DURING--
STORAGE  TOTAL  AVAIL  UNAVL     ENTRIES    AVERAGE    CURRENT   PERCENT   CAPACITY     AVERAGE
          TIME   TIME   TIME              TIME/UNIT     STATUS     AVAIL               CONTENTS
 BARBER  0.966                        68     13.635      AVAIL     100.0          2       1.932
      CURRENT     MAXIMUM
     CONTENTS    CONTENTS
            2           2
```

The interpretation of the .LIS output is as follows:

TOTAL TIME	The storage BARBER was busy 96.6% of the time. This time represents the workload of both barbers. It is not possible to tell the percentage of time each was busy.
AVAIL TIME	It is possible to "shut down" the storages by using a block to be introduced later. If a storage was shut down, the percentage of time available is given here.
UNAVL TIME	If the storage was shut down, the time shut down is given here as a decimal.
ENTRIES	The number of transactions who entered the ENTER block was 68.
AVERAGE TIME/UNIT	The average time a transaction (TIME/UNIT) was in this block was 13.635 time units.
CURRENT STATUS	The storage is currently available, as shown by AVAIL.
PERCENT AVAIL	The storage was available 100.0 percent of the time.
CAPACITY	The storage capacity was 2. This value was specified by the STORAGE statement.
AVERAGE CONTENTS	The average contents of the storage BARBER was 1.932.
CURRENT CONTENTS	When the program ended, the contents of the storage was 2.
MAXIMUM CONTENTS	The maximum number in the storage at any one time was 2.

SNAs Associated with Storages

The SNAs associated with any STORAGE statement are reserved. In the following, (name) is a storage name (may include numbers if a letter is the first character), and *n* refers to a number:

SNA	Meaning
S(name) or S*n*	current storage content
R(name) or R*n*	remaining storage content
SA(name) or SA*n*	average storage content
SC(name) or SC*n*	storage entry count
SF(name) or SF*n*	true (value = 1) means storage is full
SM(name) or SM*n*	maximum storage count
SR(name) or SR*n*	utilization of storage in parts per thousand
ST(name) or ST*n*	average holding time
SNE(name) or SNE*n*	true (value = 1) if storage is not empty
SNF(name) or SNF*n*	true (value = 1) if storage is not full

Notice that the utilization of the storage is given by SR(name) and is expressed in parts per thousand. The original storage specification is *not* an SNA. If it is desired to use this in the program, it is necessary to use S(name)+R(name), which always adds up to the original storage specification.

There are even more SNAs associated with storages. These other SNAs will not be used in this book.

Example 12.2

Actually, it is not necessary to specify a storage when you have an ENTER block. In this case, the processor sets aside 2,147,483,647 as the storage capacity. (The number is $2^{31} - 1$). It may appear that you would never omit giving the storage capacity, but there are examples when one actually does omit this. For example, suppose you are modeling a hardware store. Customers arrive and immediately take a shopping cart. Suppose customers arrive every 30 ± 8 seconds. If you have 20 carts available, you would model this situation with the following partial list of code:

```
        STORAGE    S(CART),20    PROVIDE 20 CARTS
        GENERATE   30,4          CUSTOMERS ARRIVE
        ADVANCE    0             DUMMY BLOCK
        ENTER      CART          SELECT A CART
```

Notice that an ADVANCE block is used with zero time units. This block is present so that, if all the carts are taken, the transaction will not be held up in the GENERATE block. The ADVANCE block ensures that transactions will leave the GENERATE block according to the uniform distribution of 30 ± 4 seconds. The way that the program is written, if a customer arrived and found all the carts taken, he or she would wait until a cart became free. This aspect of the program is a bit unrealistic. Suppose, instead, you wanted the customer to leave if no cart is available. The following approach is used:

```
        STORAGE    S(CART),20
        GENERATE   30,4
        TRANSFER   BOTH,,OUT
        ENTER      CART
        -----
        -----
OUT     TERMINATE
```

But, now, suppose that you wanted to determine the maximum number of carts ever used for the simulation period. To find out, you must rewrite the program so that arriving customers *always* are able to take a cart. One way would be to assign a very large storage for CART, such as

```
◇        STORAGE    S(CART),10000
```

However, in this simple example, we could just as easily omit the STORAGE statement. At the end of the program, the maximum number of entries into CART are listed. (How to print out a table of the statistical distribution of the number of carts used is covered in Chapter 19.)

THE LEAVE BLOCK

Once a transaction can use one of several parallel servers via the ENTER block, it eventually must indicate that it has done so and release the server it used. The block to do this is the LEAVE block. Just as the QUEUE and DEPART blocks are "twins," so are the ENTER and LEAVE blocks. The form is similar to the ENTER block, as it relates directly to it:

```
◇        LEAVE      A
```

where *A* is the name (or number) of the parallel servers. It can also be a variable.

Thus, you might have the following in a program:

```
◇        STORAGE    S(JOE),4
◇        -----
◇        ENTER      JOE
◇        -----
◇        -----
◇        LEAVE      JOE
◇        -----
◇        -----
```

It may appear that there is no difference between a single facility and a storage with a capacity of 1. This is almost the case. However, only a facility can be "preempted." This concept means that another transaction can replace the one that is using the facility. However, this concept will not be used in this book.

Example 12.3

A garage for inspecting a fleet of cars has 3 identical service areas. Cars arrive for inspection every 3 ± 0.3 minutes. These inspections take 8 ± 2 minutes. After leaving the inspection area, 70% are ready to return to service, but the rest need further service that takes 4 ± 1.5 minutes. If further service is needed, a single mechanic is assigned to do it. Simulate this system for 1 day.

The program to do this simulation is as follows:

```
◇         ▯SIMULATE
◇         ▯STORAGE    ▯S(SPACE),3    ▯PROVIDE SERVICE AREAS
◇COME     ▯GENERATE   ▯3,.3          ▯CARS ARRIVE
◇         ▯ADVANCE                   ▯DUMMY BLOCK (ZERO CAN BE OMITTED)
◇         ▯ENTER      ▯SPACE         ▯IS A SPACE AVAILABLE
◇         ▯ADVANCE    ▯8,2           ▯INSPECTION TAKES PLACE
◇         ▯LEAVE      ▯SPACE         ▯FREE INSPECTOR
◇         ▯TRANSFER   ▯.70,,OUT      ▯70% RETURN TO SERVICE
◇         ▯SEIZE      ▯MECH          ▯SINGLE MECHANIC
◇         ▯ADVANCE    ▯4,1.5         ▯MECHANIC WORKS ON CAR
◇         ▯RELEASE    ▯MECH          ▯FREE MECHANIC
◇OUT      ▯TERMINATE                 ▯CARS LEAVE
◇         ▯GENERATE   ▯480           ▯SIMULATE FOR A DAY
◇         ▯TERMINATE  ▯1
◇         ▯START      ▯1
◇         ▯PUTPIC     ▯LINES=2,N(COME),FR(MECH)/10.
 CARS TO COME IN  ****
 UTIL. OF MECH. ***.**%
◇         ▯END
```

Example 12.4

A hardware store consists of 4 aisles and a single checkout counter. Shoppers arrive with an interarrival time of 82.5 ± 26.4 seconds. After arriving, each customer who plans to shop in any 1 or more of the aisles takes a shopping cart. However, 12% of the customers simply go to the checkout counter where various items are for sale. These people do not take a shopping cart. The rest shop down each aisle as follows:

Aisle	Relative Probability of Customer Going Down Aisle	Time Required to Travel Aisle
1	0.80	125 ± 70
2	0.75	140 ± 40
3	0.85	150 ± 65
4	0.90	175 ± 70

When a shopper is finished, she or he will join the queue in front of the counter until each is checked out. The time to check out is 45 ± 12 seconds for those who shop in the aisles and 35 ± 12 seconds for those who go directly to the checkout counter.

The store owner is concerned that he does not have enough shopping carts. Customers who arrive and find none available tend to leave and shop elsewhere. In addition, the single person who works at the checkout counter is complaining of being overworked and is threatening to contact the union. Union regulations forbid a person to work more than 85% of the time. Determine the utilization of the checker's time and how many shopping carts the store should have. Simulate for one 8-hour shift (28,800 seconds).

Solution

```
◇        ▯SIMULATE
◇        ▯STORAGE   ▯S(CARTS),1000  ▯PROVIDE 1000 CARTS
◇COME    ▯GENERATE  ▯85.2,26.4      ▯CUSTOMERS ARRIVE
◇        ▯TRANSFER  ▯.12,,COUNTER   ▯12% GO TO COUNTER
◇        ▯ENTER     ▯CARTS          ▯REST TAKE A CART
◇        ▯TRANSFER  ▯.2,,AISLE2     ▯80% GO TO AISLE 1
◇        ▯ADVANCE   ▯125,70         ▯SHOP IN AISLE 1
◇AISLE2  ▯TRANSFER  ▯.25,,AISLE3    ▯75% GO TO AISLE 2
◇        ▯ADVANCE   ▯140,40         ▯SHOP IN AISLE 2
◇AISLE3  ▯TRANSFER  ▯.15,,AISLE4    ▯85% GO TO AISLE 3
◇        ▯ADVANCE   ▯150,65         ▯SHOP IN AISLE 3
◇AISLE4  ▯TRANSFER  ▯.10,,CHECK     ▯90% GO TO AISLE 3
◇        ▯ADVANCE   ▯175,70         ▯SHOP IN AISLE 3
◇CHECK   ▯QUEUE     ▯LINE           ▯STAND IN LINE
◇        ▯SEIZE     ▯WORKER         ▯READY TO CHECK OUT
◇        ▯DEPART    ▯LINE           ▯LEAVE THE QUEUE
◇        ▯ADVANCE   ▯45,20          ▯CHECK OUT
◇        ▯RELEASE   ▯WORKER         ▯FREE THE WORKER
◇        ▯LEAVE     ▯CARTS          ▯GET RID OF CART
◇        ▯TERMINATE                 ▯LEAVE THE STORE
◇COUNTER ▯QUEUE     ▯LINE           ▯STAND IN LINE
◇        ▯SEIZE     ▯WORKER         ▯READY TO CHECK OUT
◇        ▯DEPART    ▯LINE           ▯LEAVE THE QUEUE
◇        ▯ADVANCE   ▯35,12          ▯CHECK OUT
◇        ▯RELEASE   ▯WORKER         ▯FREE THE WORKER
◇        ▯TERMINATE                 ▯LEAVE THE STORE
◇        ▯GENERATE  ▯28800          ▯TIMER TRANSACTION ARRIVES
◇        ▯TERMINATE ▯1              ▯SIMULATION OVER
◇        ▯START     ▯20             ▯SIMULATE FOR 20 DAYS
◇        ▯PUTPIC    ▯LINES=4,N(COME),SC(CARTS),_
                 FR(WORKER)/10,SM(CARTS)
 NUMBER TO COME TO SHOP  *****
 NUMBER TO TAKE A CART   *****
 UTIL. OF WORKER           ***.*%
 MAX. NO. OF CARTS IN USE  ***
◇        ▯END
```

The output is as follows:

```
NUMBER TO COME TO SHOP  6539
NUMBER TO TAKE A CART   5747
UTIL. OF WORKER           51.0%
MAX. NO. CARTS IN USE     10
```

The number of customers arriving at the store in the 20 days was 6539. Of these, 5747 took a cart and shopped in the aisles, and 792 went directly to the checkout counter. The checker was busy only 51.0% of the time so any complaints of being overworked are not supported. The maximum number of carts ever in use was 10. It would be instructive to know how many times 10 carts were in use. In Chapter 19, it is shown how to make statistical distributions to reveal this information, but, for the present, we do not have these data. However, it would seem that providing about 12 carts should be sufficient. This quantity would take into account the maximum number obtained in the simulation as well as provide a safety factor of 2 extra carts.

Spare Truck Problem

The following is one of the most remarkable examples of the power of the GPSS/H language. The example is adapted from Schriber (1974).

A mine is in full operation. Trucks haul the ore and periodically break down. When they break down, they are taken to the repair area to be fixed. Once trucks are fixed, they are returned to the mine. They are then classified as spares, as the mine has more trucks available than are needed at any one time in the mine.

The mine needs 30 trucks for optimum production (this number was determined from a previous simulation study). The number of repair facilities can vary, but you estimate that either 3, 4, or 5 should be correct. The cost of each is $77 per 8-hour shift. Each spare truck you have costs $35 per shift no matter what it does. Thus, if you happen to have 33 trucks and 5 repair facilities, the cost per 8 hours is (5 × 77) + (3 × 35) = $490. For 3 repair facilities and 34 trucks, the costs are (3 × 77) + (4 × 35) = $371. It is possible to make a table of costs for various combinations of repair facilities and number of spare trucks.

Each time you do not have 30 trucks in the mine, it costs you $12 per hour or $96 per shift. The mean time between breakdowns for a truck is 230 ± 40 hours. The mean time for repairs of a truck is 15 ± 9 hours. Determine the optimum number of both trucks and repair facilities.

Solution

The program to do the simulation is quite compact:

```
◇        ▯SIMULATE
◇        ▯STORAGE   ▯S(MINE),30/S(REPAIR),2
◇TRUCKS  ▯GENERATE  ▯,,,32
◇BACK    ▯ENTER     ▯MINE
◇        ▯ADVANCE   ▯230,40
◇        ▯LEAVE     ▯MINE
◇        ▯ENTER     ▯REPAIR
◇        ▯ADVANCE   ▯15,9
◇        ▯LEAVE     ▯REPAIR
◇        ▯TRANSFER  ▯,BACK
◇        ▯GENERATE  ▯24*365*100
◇        ▯TERMINATE ▯1
◇        ▯START     ▯1
◇        ▯PUTPIC    ▯SR(MINE)/10.
 UTILIZATION OF MINE ***.**%
◇        ▯END
```

The above is for 32 trucks and 2 repair shops. The output from running this program gives the result that the utilization of the storage MINE is 94.59%. The cost of having the two spare trucks is $70. The cost for the two repair shops is $154. The penalty for not having 30 trucks working is found by considering that there were only 28.377 trucks working in the mine at any one time (30 × 0.9459). This result means that, at any one time, there were 1.632 trucks less that the 30 desired. The cost of this truck shortage is $155.81 per shift. Adding up the 3 costs gives $380 for the cost per shift. The program was

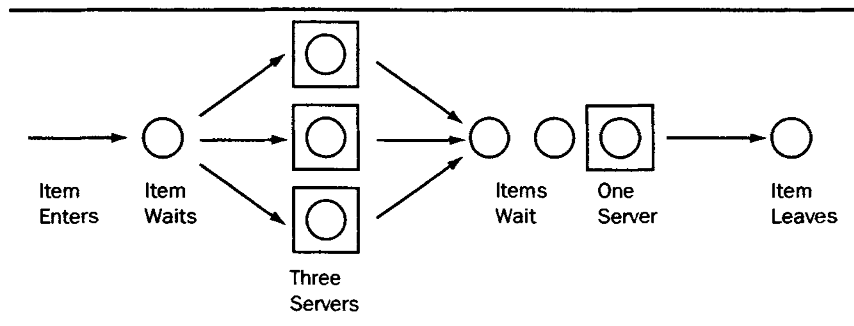
run for different combinations of trucks and repair shops; the dollar cost per shift for each combination is as follows:

	Repair Shops			
Spare Trucks	1	2	3	4
1	1523	407	385	446
2	1553	379	362	446
3	1558	368	364	454
4	1622	366	382	454
5	1662	378	410	485

As can be seen, the optimum number of trucks to have as spares is 2, and the optimum number of repair facilities is 3.

EXERCISES, CHAPTER 12

1.



Consider the above diagram. Items enter the system every 115 ± 30 seconds. They wait in a queue until 1 of 3 identical servers is ready. Service takes 335 ± 60 seconds. After this service, they move along and join another queue while they wait for a single server to perform another service. This server works in 110 ± 25 seconds. Design the model to measure the waiting-line behavior ahead of the two places where service is performed. Assume first that there is unlimited space between the 3 servers and the single server.

2. In exercise 1, suppose that there can be only 1 item in the waiting area before the second server. Modify the program to take this into account.
3. There are two tugboats that service a harbor. Ships of two types arrive at the harbor, where they unload their cargo. Type 1 ships are small, and they need only one tug to both dock and undock. Type 2 ships are large, and they need 2 tugs to both dock and undock. Because of their size differences, the ships dock at different berths and have different loading and unloading time requirements. There is always a berth available for a ship. Other data are as follows (all times are minutes):

	Ship Type	
	1	2
Interarrival Time	90 ± 30	190 ± 60
Docking Time	40 ± 7	50 ± 12
Number of Berths	6	3
Unloading Time	7500 ± 120	780 ± 240
Undocking Time	30 ± 5	42 ± 10

A type 2 ship has priority over a type 1 ship. Model the harbor for 1000 days, each day being 24 hours of operation. Suppose it cost $350 per hour for a ship of type 1 to wait for a tug either before docking or after loading or unloading and $500 per hour for a ship of type 2 to wait, and the cost of a tug is $125 per hour no matter if it is used or not. Would you recommend the addition of a third tug? (Its initial cost is immaterial.) What is the optimum number of tugboats to have?

4. A manufacturing system has parts coming along every 100 ± 20 seconds. Each needs to be assembled, formed, painted, and inspected in that order. There are 4 assembly machines, 2 forming machines, 3 painting machines, and 1 inspector. The time required to do each of these tasks is as follows:

Job	Time (seconds)
Assemble	350 ± 55
Form	170 ± 60
Paint	240 ± 30
Inspect	80 ± 25

After inspection, 5% of all parts are rejected outright. Of the remaining parts, 10% go back to the beginning for reassembly and 8% need to be repainted. Determine which of the jobs is the busiest. How many parts are made in a shift, consisting of 450 minutes of actual work?

5. In exercise 4, change the mean times for the machines and the inspector so that the utilizations of each is approximately 90%.

SOLUTIONS, CHAPTER 12

1. The program to simulate the system is:

```
        SIMULATE
        STORAGE   S(SERV1),3
COMEIN  GENERATE  115,30
        QUEUE     LINE1
        ENTER     SERV1
        DEPART    LINE1
        ADVANCE   335,60
        LEAVE     SERV1
        QUEUE     LINE2
        SEIZE     SERV2
```

```
          DEPART    LINE2
          ADVANCE   110,25
          RELEASE   SERV2
          TERMINATE
          GENERATE  280000        10 DAYS SIMULATION
          TERMINATE      1
          START          1
          PUTPIC  LINES=3,N(COMEIN),SR(SERV1)/10,FR(SERV2)/10
NUMBER PARTS TO COME INTO SYSTEM       ******
UTIL. OF THREE SERVERS                   ***.**%
UTIL. OF SINGLE SERVER                   ***.**%
          END
```

2. The changes to the program are given below:

```
          SIMULATE
          STORAGE    S(SERV1),3
COMEIN    GENERATE   115,30
          QUEUE      LINE1
          ENTER      SERV1
          DEPART     LINE1
          ADVANCE    335,60
          LEAVE      SERV1
          TRANSFER   BOTH,,AWAY     TRANSFER BOTH BLOCK
          SEIZE      DUMMY          DUMMY FACILITY
          QUEUE      LINE2
          SEIZE      SERV2
          RELEASE    DUMMY          SET DUMMY FREE
          DEPART     LINE2
          ADVANCE    110,25
          RELEASE    SERV2
          TERMINATE
AWAY      TERMINATE
          GENERATE   280000         10 DAYS SIMULATION
          TERMINATE       1
          START           1
          PUTPIC  LINES=4,N(COMEIN),SR(SERV1)/10,FR(SERV2)/10,_
                  N(AWAY)
NUMBER PARTS TO COME INTO SYSTEM       ******
UTIL. OF THREE SERVERS                   ***.**%
UTIL. OF SINGLE SERVER                   ***.**%
NUMBER TO BE SENT AWAY                  ****
          END
```

In this example, a dummy facility was used just prior to the QUEUE SERV2 block. Only one transaction can be in the block SEIZE DUMMY. The TRANSFER BOTH,,AWAY block tests to see if a transaction is in this block. If so, the transaction is routed to the block with the label AWAY.

3. The program to simulate the system is:

```
          SIMULATE
          STORAGE    S(TUG),2
          GENERATE   90,30       SMALL SHIP ARRIVES
          QUEUE      WAIT1       JOIN QUEUE
          ENTER      TUG         IS TUG AVAILABLE?
          DEPART     WAIT1       LEAVE QUEUE
          ADVANCE    40,7        DOCK
```

```
        LEAVE      TUG          FREE TUG
        ADVANCE    500,120      LOAD/UNLOAD
        QUEUE      WAIT2        WAIT FOR TUG
        ENTER      TUG          USE TUG
        DEPART     WAIT2        LEAVE QUEUE
        ADVANCE    30,5         UNDOCK
        LEAVE      TUG          FREE THE TUG
        TERMINATE               LEAVE THE HARBOR
        GENERATE   190,60,,,1   BIG SHIP ARRIVES
        QUEUE      WAIT3        JOIN QUEUE
        ENTER      TUG,2        ARE 2 TUGS AVAILABLE?
        DEPART     WAIT3        LEAVE QUEUE
        ADVANCE    50,12        DOCK
        LEAVE      TUG,2        FREE TUGS
        ADVANCE    780,240      LOAD/UNLOAD
        QUEUE      WAIT4        WAIT FOR TUGS
        ENTER      TUG,2        NEED 2 TUGS
        DEPART     WAIT4        LEAVE THE QUEUE
        ADVANCE    42,10        UNDOCK
        LEAVE      TUG,2        FREE THE TUGS
        TERMINATE               LEAVE THE HARBOR
        GENERATE   480*3*1000   SIMULATE FOR 1000 DAYS
        TERMINATE  1
        START      1
        PUTPIC  LINES=4,QA(WAIT1),QA(WAIT2),QA(WAIT3),QA(WAIT4)
AVERAGE NO. SMALL SHIPS WAITING TO DOCK   ****.**
AVERAGE NO. SMALL SHIPS WAITING TO UNDOCK ****.**
AVERAGE NO. LARGE SHIPS WAITING TO DOCK   ****.**
AVERAGE NO. LARGE SHIPS WAITING TO UNDOCK ****.**
        END
```

Running the program with 2 tugs available give the results:

```
AVERAGE NO. SMALL SHIPS WAITING TO DOCK     .52
AVERAGE NO. SMALL SHIPS WAITING TO UNDOCK   .48
AVERAGE NO. LARGE SHIPS WAITING TO DOCK     .24
AVERAGE NO. LARGE SHIPS WAITING TO UNDOCK   .27
```

This results in an expected hourly cost of: 350*(.52 + .48) + 500*(.24 + .27) + 125 *2 or $855.

Running the program with three tug boats gives:

```
AVERAGE NO. SMALL SHIPS WAITING TO DOCK     .03
AVERAGE NO. SMALL SHIPS WAITING TO UNDOCK   .03
AVERAGE NO. LARGE SHIPS WAITING TO DOCK     .04
AVERAGE NO. LARGE SHIPS WAITING TO UNDOCK   .08
```

This results in an expected hourly cost of: $338.

Thus, the optimum number of tugs to have is 3. There is no need to run the program with 4 tugs as their cost alone is $500/hour.

4. The program to do the simulation is:

```
        SIMULATE
        STORAGE      S(ASSEM),4/S(FORM),2/S(PAINT),3
        GENERATE     100,20         PARTS COME ALONG
        ADVANCE      0              DUMMY BLOCK
```

```
BACK1  ENTER       ASSEM
       ADVANCE     350,75
       LEAVE       ASSEM
       ENTER       FORM
       ADVANCE     170,60
       LEAVE       FORM
BACK2  ENTER       PAINT
       ADVANCE     240,30
       LEAVE       PAINT
       SEIZE       INSP
       ADVANCE     80,25
       RELEASE     INSP
       TRANSFER    .05,,REJECT
       TRANSFER    .1,,BACK1
       TRANSFER    .08,,BACK2
 MADE  TERMINATE
REJECT TERMINATE
       GENERATE    280000
       TERMINATE   1
       START       1
       PUTPIC      LINES=5,SR(ASSEM)/10,SR(FORM)/10,SR(PAINT)/10,_
                   FR(INSP)/10,N(MADE)/10
UTIL. OF ASSEMBLING MACHINES      ***.**%
UTIL. OF FORMING MACHINES         ***.**%
UTIL. OF PAINTING MACHINES        ***.**%
UTIL. OF INSPECTOR                ***.**%
NUMBER OF PARTS FINISHED/SHIFT  ****
       END
```

The results of the simulation are:

```
UTIL. OF ASSEMBLING MACHINES    97.41%
UTIL. OF FORMING MACHINES       94.51%
UTIL. OF PAINTING MACHINES      94.97%
UTIL. OF INSPECTOR              94.61%
NUMBER OF PARTS FINISHED/SHIFT  262
```

Notice that no QUEUE blocks were used in the program as the statistics which they gather are not needed. However, a dummy ADVANCE 0 block was needed after the first GENERATE block to ensure that the transactions created by it will always leave at the scheduled time.

5. Making the following changes:

```
Assembly:    325 ± 75
Forming:     162 ± 60
Painting:    227 ± 30
Inspecting:   76 ± 25
```

Gives the results:

```
UTIL. OF ASSEMBLING MACHINES    90.37%
UTIL. OF FORMING MACHINES       90.22%
UTIL. OF PAINTING MACHINES      89.96%
UTIL. OF INSPECTOR              90.08%
NUMBER OF PARTS FINISHED/SHIFT  262
```

6. The program to do the simulation is:

```
       SIMULATE
       STORAGE        S(CART),1000  PROVIDE FOR 1000 CARTS
       GENERATE       82.5,50
       TRANSFER       .12,,CHECK
       ENTER          CART
       TRANSFER       .2,,NEXT1
       ADVANCE        125,70     AISLE 1
NEXT1  TRANSFER       .25,,NEXT2
       ADVANCE        140,40     AISLE 2
NEXT2  TRANSFER       .15,,NEXT3
       ADVANCE        150,65     AISLE 3
NEXT3  TRANSFER       .1,,NEXT4
       ADVANCE        175,70
NEXT4  QUEUE          WAIT
       SEIZE          CHECKER
       DEPART         WAIT
       ADVANCE        45,12
       RELEASE        CHECKER
       LEAVE          CART
CHECK  QUEUE          WAIT
       SEIZE          CHECKER
       DEPART         WAIT
       ADVANCE        20,8
       RELEASE        CHECKER
       TERMINATE
       GENERATE       3600*100
       TERMINATE      1
       START          1
       PUTPIC         LINES=2,FR(CHECKER)/10,SM(CART)
CHECKER WORKED  ***.**% OF THE TIME
MAXIMUM CARTS IN USE ****
       END
```

The results of the simulation yield:

```
CHECKER WORKED   71.94% OF THE TIME
MAXIMUM CARTS IN USE  11
```

This means that the checker is not overworked. The simulation ran for 100 hours of 3,600 seconds each. It probably should be run for more time to see if this maximum number changes before making a recommendation.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 13

The CLEAR, RESET, and RMULT Statements

MULTIPLE START STATEMENTS

A GPSS/H program is compiled. During this phase, transactions are poised on the CEC to be moved by the processor when the actual program begins. There will be one transaction from each GENERATE block poised to be moved. As soon as this transaction is moved, another one is scheduled to be moved.

When a program encounters the *first* START statement, it begins execution. If there are other statements after the START statement, the processor will execute them *after* the initial program is executed. As far as understanding how the running of a given program takes place, you can initially ignore any statements after the first START statement.

For example, suppose you had other START statements one after the other such as

```
◇          GENERATE   10,5
◇          -----
◇          -----
◇          -----
◇          GENERATE   480
◇          TERMINATE  1
◇          START      1
◇          START      1
◇          START      1
◇          END
```

The effect of the above is to run the program three times (just as if you had START 3), but now there will be additional output from the three times (at 480, 960, and 1440 time units).

It is possible to have the START statement with operands as follows:

```
◇          START      A,NP,B,1
```

NP must be the letters "NP" (this stands for "no printout," and no output file is created). *B* is called the "Snap" interval. Every time the *A* counter is decremented by this amount, there will be output. If the last operand is used, it must be the number 1. When used, the processor outputs the various chains (current events chain, future events chain, etc.). As an example, if you had

```
◇       □START      □100,,10
```

you would get output for when the counter was 90, 80, 70, etc. At one time, this statement was used mainly for debugging purposes, but now the BPUTPIC has made this use nearly obsolete.

When you have multiple START statements, the program keeps running from the point where the last transaction was made. The transactions remain where they were, and the program keeps going. All statistics are updated from where they were.

Often you will want to run the program multiple times with different values in the blocks. One of several ways to do this is as follows:

```
◇       □SIMULATE
◇       □GENERATE  □12,6
◇       □ADVANCE   □10,4
◇       □TERMINATE □1
◇       □START     □1
◇       □END
```

Add labels to the blocks you want to change, e.g., if you want to change a GENERATE block, give the original GENERATE block a label such as KEY1, for example,

```
◇KEY1   □GENERATE  □12,6
```

The program begins execution at the first START statement and ignores all the ones that follow it.

Suppose that you want to run the program a second time with the block GENERATE 12,6 changed to GENERATE 11,6. You change the program to the following:

```
◇       □SIMULATE
◇KEY1   □GENERATE  □12,6
◇       □ADVANCE   □10,4
◇       □TERMINATE □1
◇       □START     □1
◇KEY1   □GENERATE  □11,6
◇       □START     □1
◇       □END
```

Again, the program runs when it encounters the first START 1 statement and produces output. Then, it replaces the block

```
KEY1     GENERATE  12,6
```

with the block

```
KEY1     GENERATE  11,6
```

Because the next block is another START 1, the program executes a second time with the new GENERATE block. Note that the operands are different for the two GENERATE blocks.

The program will run initially with the block GENERATE 12,6 and produce an output report. It will then run again with the GENERATE 12,6 block replaced by the GENERATE 11,6 block and will produce a second output report.

The only problem with this procedure is that the second running of the program will have the statistics and transactions from the previous run already in the system. This situation is probably not what you want. The next section will show how to run the program a second time with the previous statistics zeroed out.

THE CLEAR STATEMENT

The CLEAR statement allows a program to be run multiple times with the transactions and statistics of the system set to zero each time. Thus, the following code

```
         SIMULATE
KEY1     GENERATE  12,6
         ADVANCE   10,4
         TERMINATE 1
         START     1
         GENERATE  480
         TERMINATE 1
         START     1
         CLEAR
KEY1     GENERATE  11,6
         START     1
         END
```

will run the program twice, the first time with GENERATE 12,6 and the second time with GENERATE 11,6; fresh statistics will be generated during the second running. It would make no difference if the CLEAR statement followed the GENERATE 11,6 block instead of preceding it.

When you run a program with this type of replacement, there will be a warning message that you have named the block multiple times—you ignore the message as you intentionally did this. Remember that, whenever you label a block in GPSS/H and do not reference it in the program, there will be a warning message.

You can stack up as many START statements together with the replacements as desired. The effect of this approach is to make the coding of the program

long. Chapter 23 shows how to run a program by using DO loops, which makes this type of cumbersome replacement obsolete.

Since the CLEAR statement clears all statistics, there will be a time when you will want only selected items to be cleared. You can preserve items by putting the items you do *not* want cleared after the CLEAR statement. The effect of the following CLEAR statement is to clear all items *except* item1, item2, ..., and any others listed up to item *n*.

```
◇       □CLEAR     □item1,item2,...,item n
```

THE RESET STATEMENT

There will be times when you do not want to clear out the transactions (or the various chains). For example, suppose you want to run a program for 1 hour (beginning when the model is empty) and then discard the statistics but *not* the positions of the transactions. You write the following:

```
◇       □START     □1,NP          □Simulate for 1 hour
◇       □RESET                    □Reset the statistics
◇       □START     □100           □Simulate for 100 hours
```

The RESET statement is often used when the first hour's (or so) statistics are to be discarded owing to instability in the system.

Note: Whenever you use CLEAR or RESET, you do *not* change the position of the random-number generator. This method that GPSS/H uses to acquire random numbers is a pseudo-random-number generation (see Chapter 6), which means that the numbers can be repeated. This fact is important in designing a comparison of models. How, then, can you restart the random-number stream at the same place? This is done by the RMULT statement.

THE RMULT STATEMENT

The RMULT statement followed by any number in the range ±2,147,483,646 to be used in the operand starts the random-number stream at a particular point. Thus, if you had

```
◇       □SIMULATE
◇       □RMULT     □54321
◇       □-----
◇       □-----
◇       □-----
◇       □START     □1
◇       □CLEAR
◇       □RMULT     □54321
◇KEY    □GENERATE  □10,6
◇       □START     □1
◇       □END
```

The second time the program is run, the same random numbers would be used.

GPSS/H programs can use any number of random-number streams in a given run. How to specify which one to use will be covered in Chapter 14. Until now, we have been using only the first, by default. When you are using multiple streams and want to reset each, the RMULT statement is

```
◇        RMULT      1234,6543,,99,,231,999
```

Here random-number streams 1, 2, 4, 6, and 7 are reset.

Encountering a group of CLEAR, RESET, and RMULT statement at the end of a program may seem confusing at first, but remember that these are *control statements* and *not* program blocks. Always remember that the program starts execution after each START statement and ignores the remaining statements until execution ends for the START statement it has encountered. When the RESET statement is used, the absolute clock is unchanged, but the relative clock is set to zero. The SNA for the relative clock is C1.

EXERCISES, CHAPTER 13

1. What will the GPSS/H absolute clock read at the end of execution of the full program that has the following three START statements?

```
◇        -----
◇        -----
◇        GENERATE   480
◇        TERMINATE  1
◇        START      1
◇        START      2
◇        START      3
◇        END
```

2. What will the GPSS/H absolute and relative clocks read when the following program finishes execution?

```
◇        -----
◇        -----
◇        GENERATE   100
◇        TERMINATE  1
◇        START      4
◇        RESET
◇        START      8
◇        END
```

3. What will the GPSS/H absolute clock read for the following whenever there is output sent to the .LIS file?

```
◇        -----
◇        -----
◇        TERMINATE  5
◇        START      100,,5
```

4. What is the effect of using the following control statements in a program?

```
◇         ▯-----
◇         ▯-----
◇TRUCK    ▯GENERATE  ▯,,,5
◇         ▯-----
◇         ▯-----
◇         ▯GENERATE  ▯480*100
◇         ▯TERMINATE ▯1
◇         ▯RMULT     ▯12345
◇         ▯START     ▯1
◇         ▯CLEAR
◇         ▯RMULT     ▯12345
◇TRUCK    ▯GENERATE  ▯,,,6
◇         ▯START     ▯1
◇         ▯END
```

SOLUTIONS, CHAPTER 13

1. The absolute clock will read 1920.0000. The first START stopped the clock at 480.000, the second at 960.0000 and the third at 1920.000.
2. The absolute clock will read 500.0000 and the relative clock will read 300.0000.
3. The absolute clock times will be: 25.0000, 50.0000, 75.0000, and 100.0000.
4. The program will run the first time with the random the number generator starting in position 12345. The simulation runs until 48000 time units have gone by and the first output report created. Then the statistics are cleared out. The random number generator is reset to position 12345. The block `◇TRUCK ▯GENERATE ▯,,,4` is replaced with the block

   `◇TRUCK ▯GENERATE ▯,,,5`

   and the simulation rerun for 480000 time units.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 14 Functions

FUNCTIONS IN GPSS/H

The only statistical distribution we have used so far is the uniform distribution. In GPSS/H, it is possible to sample from *any* statistical distribution. This is very important, since to model most things that are moving through a system, it is necessary to sample from different distributions. Among the possibilities are the exponential or Poisson distribution, the normal or Gaussian distribution, and even distributions that are unique to the model. There are, in fact, many times when we will refer to a function to return a value to be used in the simulation. There are several types of functions in GPSS/H that will be considered here.

DISCRETE FUNCTIONS

A function first must be defined and then referenced when a value is to be obtained from it. The first form of a function to be discussed here is known as a *discrete function*, because, when referenced, it will take on only one of several set (or discrete) values. These possible values are specified when the function is defined. For example, an order for goods may take 4, 5, 6, or 7 weeks to arrive. The number of people away from a class owing to illness might be from 0 to 12.

Definition of a Discrete Function

The way to reference discrete functions in GPSS/H is first to define the function by means of the FUNCTION statement. This definition must be made before the function is referenced. (How to reference a function is explained in the next main section.) The form of the first line of the function definition is as follows:

```
◇LABEL    □FUNCTION  □A,B
```

where the label is the name or number of the function. This label will be used to reference the function.

Operand *A* is *any* standard numerical attribute. The most commonly used is RN*j*, which supplies random numbers from random-number stream *j*. Initially in this section, the SNA considered will be RN*j*, but *any* SNA can be used.

Operand *B* is the term D*n*, in which the letter D is for "discrete" and *n* is a positive integer. The value of *n* in D*n* gives the exact (i.e., discrete) number of values that the function may take on, which is numerically identical to the number of pairs of values to be used during the statistical sampling procedure that selects which value is actually taken. The first number in each pair forms one endpoint of the range within which the random number returned must fall to allow the second number of the pair to be used.

Whenever RN*j* [or RN(*j*)] is referenced, a random number is returned. If the reference is in connection with a function, the number returned is between 0.000000 and 0.999999 (from 0 to 1 but *never* 1.000000). When RN*j* is used in other situations, the number returned is between 000 and 999.

For example,

```
◇FIRST    □FUNCTION  □RN1,D2
```

would be a function named (i.e., labeled) FIRST. It can take on 2 values. These values will be found on the next line or lines below the FUNCTION statement, as explained at the end of this section.

```
◇TOP      □FUNCTION  □RN3,D5
```

is the function TOP that will take 1 of 5 values.

```
◇FIVE     □FUNCTION  □RN6,D15
```

refers to the function FIVE that can have 1 of 15 values. It is also possible to have labels for functions that are numbers such as

```
◇6        □FUNCTION  □RN1,D5
```

On the line or lines below the FUNCTION statement are the D*n* possible values that the function might assume. These are in pairs separated by a comma and usually in ascending order, although they may be in descending order for a discrete function. The pairs are themselves separated by a slash (/). These values can start in position 1 *(this is one of the few times in GPSS/H that anything can be in position 1)* and go up to and include position 72. The pairs can occupy more than one line, if necessary. If so, the slash can be omitted from the end of the previous line. The first number of the pair refers to the GPSS/H-generated random number.

How a Discrete Function Works

When a function having RN*j* as an operand is referenced, a random number is obtained from the random-number stream referenced (i.e., *j*). In the case of RN1, the first random-number stream is used because of the 1 in RN1. RN5 would reference random-number stream 5. There is no limit to the number of random-number streams in GPSS/H, although most references will be to RN1

for convenience. The random number returned is then used to obtain a value to be returned. Thus,

```
◇SALLY □FUNCTION  □RN1,D3
.1,5/.6,8/1,10
```

will return 5, 8, or 10 and no other values. If the random number obtained from the random number stream is between 0.000000 and 0.100000, the number returned is 5; if the number is from 0.1000001 to 0.600000, the number returned is 8; if the number is from 0.600001 to 0.999999 the number returned is 10. This approach to selecting the value of the function is better seen as follows:

Random Number	Value Returned
0.000000 ≤ RN ≤ 0.100000	5
0.100001 ≤ RN ≤ 0.600000	8
0.600001 ≤ RN ≤ 0.999999	10

This procedure is known as Monte Carlo sampling. Simulations done by using this form of sampling are known as Monte Carlo simulations.

REFERENCING FUNCTIONS IN GPSS/H

Note: The former method of referencing functions was to use a single dollar sign ($). Thus, to reference the function FAST, one would have something like ADVANCE FN$FAST. GPSS/H still supports this method, but it will not be used here.

Functions are referenced in GPSS/H by the letters FN followed by the name of the function in parentheses. (In the case of using a number for a function, reference is FN(j) or simply by FNj where j is the number of the function.) Some examples are as follows:

```
◇(a)    □ADVANCE   □FN(TIME)
◇(b)    □GENERATE  □FN(SPEED)
◇(c)    □GENERATE  □,,,4,FN(AMOUNT)
◇(d)    □GENERATE  □100,25,FN(TIMEIN)
```

In (a), the transaction will be placed on the FEC for a duration given by reference to the function TIME. In (b), a transaction will be generated according to the time given by the function SPEED. This means that, during compiling, a transaction is scheduled to leave the GENERATE block. As soon as this transaction leaves, another is scheduled to leave. The times to leave are given by reference to the function SPEED. The statement in (c) will generate 4 transactions at time $t = 0$. The priority of each will be given by the function AMOUNT. The statement in (d) will generate transactions every 100 ± 25 time units. The first transaction will enter the system at a time given by reference to the function TIMEIN.

The number returned when a function is referenced need not be an integer. Thus, it would be possible to have

```
◇TEST6    □FUNCTION    □RN3,D4
.25,3.5/.44,5.1/.7,7.8/1,9.89
```

Here the returned values would be 3.5, 5.1, 7.8, or 9.89. If you had

```
◇TEST7    □FUNCTION    □RN1,D3
.1,4/.5,6/.8,9
```

and a random number greater than 0.8 was returned when RN1 was sampled, the value of the function would be 9.

Example 14.1

Suppose that a shopper takes 3, 4, or 5 minutes to load a shopping cart and that 30% of the time it takes 3 minutes, 30% of the time it takes 4 minutes, and the remaining 40% of the time it takes 5 minutes. To use a GPSS/H function, it is necessary to make a cumulative probability distribution. The following illustrates this distribution:

Time (minutes)	Relative Probability	Cumulative Probability
3	.30	.30
4	.30	.60
5	.40	1.00

Since 30% of the time it takes the shopper 3 minutes to load the cart, we would like to have the GPSS/H function return a time of 3 minutes also 30% of the time. This matching is done by assigning the value of the random number generated by the GPSS/H processor to the corresponding probability:

Time (minutes)	Cumulative Probability	Random Number
3	0.30	.000000 to .300000
4	0.60	.300001 to .600000
5	1.00	.600000 to .999999

If the random number is between 0.000000 and 0.300000, the time is taken as 3; if the random number is from 0.300001 to 0.600000, the time is 4, and, if the random number is from 0.600000 to 0.999999, the value is 5. Notice that, by using cumulative probability distributions, the correspondence between the random number and a time is unique, i.e., for each random number there is 1 and only 1 value that corresponds. Notice, also, that there is a very slight error since the random number is never equal to 1.000000. This error is so small that it is not significant.

Returning to the original problem, we would write the function as

```
◇SHOPR    □FUNCTION □RN1,D3
.3,3/.6,4/1,5
```

Notice that there is no decimal after the 1 in the pair 1,5. The decimal is optional here. There is no slash at the beginning or end of the pairs of numbers.

It also would have been all right for the function describing the shoppers to be written as follows:

```
◇SHOPR   ▯FUNCTION   ▯RN1,D3
.3,3/.6,4
1,5
-----
-----
-----
◇        ▯ADVANCE    ▯FN(SHOPR)
```

or even to have each pair of values on a separate line.

Consider the following:

```
◇TIME    ▯FUNCTION ▯RN1,D4
.25,8/.50,9/.9,10/1,11
```

Later in the program, if you had

```
◇        ▯ADVANCE   ▯FN(TIME)
```

the transaction would be put on the future events chain for a time of 8, 9, 10, or 11 time units. The number of time units will be 8 for 25% of the transactions; 9 for 25% of the transactions; 10 for 40% of the transactions; and 11 for the remaining 10% of the transactions. If we had

```
◇        ▯ADVANCE   ▯FN(TIME)5
```

the transaction would be on the future events chain for one of the following times: 8 ± 5, 9 ± 5, 10 ± 5, or 11 ± 5 time units.

CAUTION IN USING FUNCTIONS WITH ADVANCE AND GENERATE BLOCKS

When either an ADVANCE block or a GENERATE block has a *function* in the *B* operand, the effect is to *multiply* the value in the *A* operand by the value of the function. Thus, if you had the following functions

```
◇ONE     ▯FUNCTION  ▯RN1,D3
.3,10/.6,15/1,20
◇TWO     ▯FUNCTION  ▯RN1,D2
.4,2/1,3
```

and you used

```
◇        ▯ADVANCE   ▯FN(ONE),FN(TWO)
```

and if FN(ONE) had the value of 15 and FN(TWO) had the value of 3, the transaction would be placed on the future events chain for 45 time units, *not* 15 ± 3 time units. In Chapter 18, we shall see how to achieve the ± result in the ADVANCE block.

Example 14.2

Customers arrive at a parking lot every 100 ± 23 seconds. It takes them 13 ± 5 seconds to park and 12 ± 2 seconds to walk to the store. Shopping time varies as follows:

Time (seconds)	Relative Frequency	Cumulative Frequency
50	0.12	0.12
55	0.25	0.37
65	0.27	0.64
75	0.22	0.86
90	0.14	1.00

The store is small and has only one checkout counter. The time required to check out is as follows:

Time (seconds)	Relative Frequency	Cumulative Frequency
80	0.25	0.25
90	0.20	0.45
100	0.30	0.75
120	0.25	1.00

Of the total customers arriving, 10% make no purchase and leave the store. The time to return to their cars is 20 ± 8 seconds, and the time to leave the parking lot is 8 ± 4 seconds. Simulate for 1000 people arriving at the store, shopping, and driving away.

Solution

```
◇         ▯SIMULATE
◇SHOP     ▯FUNCTION  ▯RN1,D5
.12,50/.37,55/.64,65/.86,75/1,90
◇CHKOT    ▯FUNCTION  ▯RN1,D4
.25,80/.45,90/.75,100/1,120
◇         ▯GENERATE  ▯100,23        ▯CUSTOMERS ARRIVE
◇         ▯ADVANCE   ▯13,5          ▯PARK CAR
◇         ▯ADVANCE   ▯12,2          ▯WALK TO STORE
◇         ▯ADVANCE   ▯FN(SHOP)      ▯SHOP
◇         ▯TRANSFER  ▯.1,,OUT       ▯10% MAKE NO PURCHASE
◇         ▯QUEUE     ▯LINE          ▯REST QUEUE AT CHECKOUT
◇         ▯SEIZE     ▯CHECKER       ▯USE CHECKER
◇         ▯DEPART    ▯LINE          ▯LEAVE QUEUE
◇         ▯ADVANCE   ▯FN(CHKOT)     ▯CHECK OUT
◇         ▯RELEASE   ▯CHECKER       ▯FREE CHECKER
◇OUT      ▯ADVANCE   ▯20,8          ▯RETURN TO CAR
◇         ▯ADVANCE   ▯8,4           ▯LEAVE PARKING LOT
◇         ▯TERMINATE ▯1
◇         ▯START     ▯1000
◇COMEIN   ▯TERMINATE ▯1
◇         ▯START     ▯1000
◇         ▯PUTPIC    ▯LINES=3,AC1/60,FR(CHECKER)/10,QA(LINE)
 TIME FOR 1000 PEOPLE TO SHOP:  *******.**  MINUTES
```

```
UTIL. OF CHECKER                    ***.**%
AVERAGE PEOPLE IN CHECKOUT LINE     ***.**
◇         ⎕END
```

The output from this program is

```
TIME FOR 1000 PEOPLE TO SHOP:     1672.54  MINUTES
UTIL. OF CHECKER                   89.01%
AVERAGE PEOPLE IN CHECKOUT LINE     0.27
```

Although this example illustrates the use of the FUNCTION statement, it is not realistic. The most glaring deficiency is the fact that the various shopping and checkout times are given by discrete values. Certain times such as 51 seconds for shopping and 97 seconds for checking out are not considered. They could have been included by making the FUNCTION statement much longer. This approach would be tedious and is not necessary, as we shall learn in the next section. However, the following points also should be considered in building an actual model of a grocery store:

1. Shoppers do not arrive during the day according to the same statistical distribution. At certain times (between 4:30 p.m. and closing, for example) there are more shoppers than during other times. A more nearly accurate function should reflect this.
2. The parking lot holds only a finite number of cars. If another customer drives into the lot when it is full, the customer may leave. This possibility can be programmed by means of the STORAGE statement and the ENTER block.
3. It might be better to have the checkout time vary, depending on the number of items purchased.
4. Some shoppers may not join the queue if it is too long, but will prefer to continue shopping.
5. If the queue reached a certain length, the one checker may be able to call for another helper.
6. The time to walk from the parking lot to the store (and subsequently return) should vary with the number of cars in the lot. The more cars in the lot, the farther away the next car that arrives will have to park.

As more GPSS/H code is introduced, it will be possible to incorporate the above changes into the model.

CONTINUOUS FUNCTIONS

A continuous function is defined and referenced in much the same manner as a discrete one. In the case of discrete functions, only a finite number of values can be returned. The number is specified by the operand D*n*, and the possible values are the second numbers in the pairs that follow D*n*. For continuous functions, a value is returned that can be considered as being a decimal number within a specified range, i.e., if the range is from 4 to 7, any possible value from 4.000000 to 6.999999 can be returned. Notice that the range is an "open" one in that the right endpoint is not included.

The two values forming the pairs that give the ranges must be in ascending order, and the number of these pairs is given by the *n* in C*n* (the C indicates continuous). One form of a continuous function might have the first line as

```
◇TEST1   □FUNCTION □RN1,C5
```

This line means that the function labeled TEST1 will use random-number stream 1 to return a random number. The value of the function will be determined by sampling from 5 pairs of numbers. The ordered pairs of numbers that come on the line(s) after the function line specify the intervals within which the value of the function is to be obtained. The value is determined by a linear interpolation between the pairs. For example, the function

```
◇TEST2   □FUNCTION □RN1,C3
0,1/.7,4/1,5
```

will return the following values:

Value Returned from Interval	If Random Number Generated Is
1–4.000000	0–0.7000000
4.000001–4.999999	0.700001–0.999999

Suppose that, when the function is referenced, the random number generated is 0.300000. The value of the function, 2.285714, is obtained by linear interpolation (done automatically by the GPSS/H processor). The calculation is as follows:

$$(y - 1)/(0.3 - 0) = (4 - y)/(0.7 - 0.3)$$

Consider the function

```
◇TEST3   □FUNCTION □RN1,C2
0,1/1,5
```

This function will return a value between 1.000000 and 4.999999 but not 5.000000 because the random number is always from 0.000000 to 0.999999.

USE OF FUNCTIONS WITH ANY SNA

In a function definition, it is possible to use *any* SNA instead of a random number. Thus, one could have the following:

```
◇TIMES   □FUNCTION □Q(WAIT),D5
0,10/1,8/2,6/3,5/4,4
◇FIRST   □FUNCTION □FR(MACH1),D3
100,5.5/500,6.6/923,7.6
◇SECOND □FUNCTION □SR(BOOTH),D2
500,20/999,30
```

The function with the label TIMES will return a value of 10, 8, 6, 5, or 4 depending on the current length of the queue WAIT; note that descending order for the possible values is acceptable for discrete functions, but not for continuous functions. The function FIRST will return a value of 5.5, 6.6, or 7.6 depending on the fractional utilization of the facility MACH1. The function

SECOND will return a value of either 20 or 30 depending on the utilization of the storage BOOTH. An example of the use of such a function reference might be a barber who cuts hair at a rate that depends on the number of customers waiting in the shop. As the queue increases, the barber will increase the cutting rate. For example, suppose the normal time is 10 minutes if the queue is 0. If 1 customer is waiting, the time is 9 minutes; if 2 are waiting, the time is 8.5 minutes; and if 3 or more are waiting, the time is 8 minutes. The program might have the following lines of code:

```
◇CUTTING ▯FUNCTION   ▯Q(WAIT),D4
0,10/1,9/2,8.5/3,8
◇        ▯-----
◇        ▯-----
◇        ▯ADVANCE    ▯FN(CUTTING)
```

OTHER FORMS OF FUNCTIONS

There are three other forms of functions that are used in GPSS/H.

List Functions

Many times the first number in each of a function's referencing pairs is the sequence of integers 1, 2, 3, ... , *N*. In this case, the function is called a "list" function, and this fact is specified by L*n* (where *n* = the number of pairs in list L of possible values), as follows:

```
◇EXAMPLE ▯FUNCTION   ▯Q(WAIT),L3
1,1/2,4/3,5
```

It would be wrong to begin the list with 0:

```
◇EXAMPLE ▯FUNCTION   ▯Q(WAIT),L4
0,1/1,1/2,4/3,5
```

A list function is preferable to use if a list type of situation is involved in a simulation. Not only does a list function take less time to execute, but, if the value of the SNA is outside the range 1 to *N*, an execution error occurs. Therefore, using a list function can be useful in debugging a program. Thus, in the example cited here, if the queue at WAIT was 0, the program would return an error. As we learn more programming code, there will be more examples of list functions. Since the *x*-values of every list function must start with 1 and be incremented by 1, these values can be omitted. Thus, a correct way to write the preceding function would be

```
◇EXAMPLE ▯FUNCTION   ▯Q(WAIT),L4
,1/,1/,4/,5
```

Attribute-Valued Functions

It is possible to have functions that reference other functions. These are known as attribute-valued functions and are specified by E*n*. For example,

```
◇SPEED   ▯FUNCTION   ▯RN1,E3
.25,FN(ONE)/.6,FN(TWO)/1,6
```

If the value of the random number is less than or equal to 0.25, the value returned is obtained by reference to the function ONE; if the value of RN1 is from 0.25 to 0.6, the value is obtained by reference to function TWO; otherwise the value is 6.

If an attribute-valued function is also a list function, the letter E is replaced by the letter M. It is possible for an attribute-valued function to reference functions that are themselves attribute-valued functions.

There is still another type of function, the entity type. This function is noted by the letter S, but will not be used in this book.

EXERCISES, CHAPTER 14

1. Build a function to return the following times:

Time (minutes)	Probability
5	0.10
6	0.20
7	0.40
8	0.30

2. Refer to example 1. Suppose the parking lot held only 2 cars and any other cars that came would be turned away. Assume the store is in operation for 5 days per week for 10 hours per day. Run the simulation for room first for 3 and then 4 cars to park.
3. In a study of the time it takes Joe to give a haircut, 155 customers were timed and the following data were obtained:

Time (minutes)	Number of People
<10	0
10 to 11	87
11 to 12	35
12 to 13	20
13 to 14	13
>14	0

What FUNCTION statement could be used in the ADVANCE block to simulate getting a haircut from Joe? This should be a continuous function.

4. The following data give the interarrival rates of parts coming into a repair shop.

Time Between Arrivals (minutes)	Relative Frequency
< 10	0
10 to 11	0.10
11 to 12	0.25

(continues)

Time Between Arrivals (minutes)	Relative Frequency
12 to 13	0.35
13 to 14	0.20
14 to 15	0.08
15 to 16	0.02
>16	0

There is only one machine for doing repairs, and it can only repair one part at a time. There are only two spaces for parts to be held in; if they are both full, no more parts can be repaired by the machine, and any others arriving at the shop will be routed to another shop. This means that the maximum number of parts in the repair shop is 3. Service time varies according to the following data:

Time for Service (minutes)	Relative Frequency
<30	0
30 to 31	0.25
31 to 32	0.22
32 to 33	0.28
33 to 34	0.18
34 to 35	0.07
>35	0

Simulate the repair shop to determine how many customers are turned away in a typical day.

5. The results of exercise 4 show that the system is highly inefficient in that too many parts are being sent to another repair shop. Run the program with the following changes and determine the effect of each. (Note: for problems 5c and 5d, assume that all you need to do is to change the ADVANCE block.)
   a. There are places for 5 parts to be held in, rather than 2.
   b. A new machine can repair the parts twice as fast as the old one.
   c. Another new machine can repair the parts 2.5 times as fast as the old one.
   d. Still another machine can be purchased that is 3 times as fast as the old machine.
6. A widget manufacturing process is as follows:
   a. A worker takes a partially finished widget from a large pile. (This pile can be considered as infinitely large.)
   b. The worker then finishes assembling the widget.
   c. The worker takes it to a painting machine for final painting. There is only one of these painting machines, and the person who assembled the widget is responsible for painting it.
   d. The worker finishes the painting, puts the widget on a conveyor belt, and returns to the original work station to begin the final assembly of another widget.

The various associated times are

a. Finish assembly:

Time (minutes)	Relative Probability	Cumulative Probability
20	0.05	0.05
21	0.12	0.17
22	0.18	0.35
23	0.21	0.56
24	0.25	0.81
25	0.15	0.96
26	0.04	1.00

b. Take widget to paint area: 1 ± 0.4 minute

c. Paint widget:

Time (minutes)	Relative Probability	Cumulative Probability
6	0.10	0.10
7	0.25	0.35
8	0.35	0.70
9	0.23	0.93
10	0.07	1.00

d. Return to work station: 1 ± 0.4 minute

Each widget earns the company a profit of $12.75. A worker is paid $9.50 per hour. The fixed cost of the operation is $75 per day. Determine the optimum number of workers to have making widgets. Simulate for 100 shifts of 480 minutes each.

7. Refer to Exercise 6. Suppose the widget manufacturing plant determined that the demand for widgets was such that a second painting facility was justified. Determine the correct number of new workers to hire to make the additional widgets.

8. Customers come to use a single server. They will always wait in a queue if the server is busy. Suppose that the arrival and service rates for the different possible distributions in minutes are as follows:

Distribution	Arrival	Service
Uniform	5 ± 3	4 ± 1
Normal	Mean = 5, Std. Dev. = 1	Mean = 4, Std. Dev. = 0.7
Exponential	Mean = 5	4

Write the GPSS/H program to compare the utilization of the server for the 3 different distributions. Simulate for 500 shifts of 480 minutes each.

SOLUTIONS, CHAPTER 14

1. One such function would be:

```
 TIMES FUNCTION  RN3,C3
.1,5/.3,6/.7,7/1,8
```

2. The changes to the program are shown below for the case of 2 parking places:

```
       SIMULATE
 SHOP  FUNCTION     RN1,D5
.12,50/.37,55/.64,65/.86,75/1,90
 CHKOT  FUNCTION    RN1,D4
.25,80/.45,90/.75,100/1,120
       STORAGE      S(LOT),2
       GENERATE     100,23
       TRANSFER     BOTH,,AWAY
       ENTER        LOT
       ADVANCE      13,5
       ADVANCE      12,2
       ADVANCE      FN(SHOP)
       TRANSFER     .1,,OUT
       QUEUE        LINE
       SEIZE        GIRL
       DEPART       LINE
       ADVANCE      FN(CHKOT)
       RELEASE      GIRL
  OUT  ADVANCE      4,2
       ADVANCE      8,4
       LEAVE        LOT
COMEIN TERMINATE
 AWAY  TERMINATE
       GENERATE     3600*8*5*2
       TERMINATE    1
       START        1
       PUTPIC       LINES=4,N(COMEIN)/2,FR(GIRL)/10,QA(LINE),N(AWAY)
NUMBER OF PEOPLE SERVED PER WEEK:  *******
UTIL. OF CHECKOUT GIRL                 ***.**
AVERAGE PEOPLE IN CHECKOUT LINE        ***.**
PEOPLE WHO ARE TURNED AWAY             ***
       END
```

The results of the program for 2 spaces are:

```
NUMBER OF PEOPLE PER WEEK:        1102
UTIL. OF CHECKOUT GIRL             66.84
AVERAGE PEOPLE IN CHECKOUT LINE      .04
PEOPLE WHO ARE TURNED AWAY        670
```

If a third parking place can be found, the results are:

```
NUMBER OF PEOPLE PER WEEK:        1416
UTIL. OF CHECKOUT GIRL             86.10
AVERAGE PEOPLE IN CHECKOUT LINE      .19
PEOPLE WHO ARE TURNED AWAY          33
```

For 4 parking spaces, the results are:

```
NUMBER OF PEOPLE PER WEEK:          1437
UTIL. OF CHECKOUT GIRL                89.35
AVERAGE PEOPLE IN CHECKOUT LINE         .28
PEOPLE WHO ARE TURNED AWAY            2
```

3. The following function will work:

```
      GIVECUT  FUNCTION  RN1,C5
0,10/.561,11/.787,12/..916,13/1,14
```

4. The program is as follows:

```
         SIMULATE
COMEIN   FUNCTION   RN1,C7
0,10/.1,11/.35,12/.7,13/.9,14/.98,15/1,16
SERVICE  FUNCTION   RN1,C6
0,31/.25,32/.47,33/.75,34/.93,35/1,36
         STORAGE    S(ROOM),2
INSHOP   GENERATE   FN(COMEIN)
         QUEUE      WAIT
         TRANSFER   BOTH,,AWAY
         ENTER      ROOM
         SEIZE      WORKER
         DEPART     WAIT
         LEAVE      ROOM
         ADVANCE    FN(SERVICE)
         RELEASE    WORKER
         TERMINATE
   AWAY  DEPART     WAIT
   GGGG  TERMINATE
         GENERATE   480*100
         TERMINATE
         START      1
         PUTPIC     LINES=4,N(INSHOP),N(GGGG),SR(ROOM)/10,_
                    FR(WORKER)/10
NUMBER TO ARRIVE AT SHOP              *****
NUMBER TO LEAVE WITH NO SERVICE       *****
UTIL. OF THE ROOM                       ***.**%
UTIL. OF MACHINE                        ***.**%
         END
```

The results of the simulation yield the following:

```
NUMBER TO ARRIVE AT SHOP          3842
NUMBER TO LEAVE WITH NO SERVICE   2396
UTIL. OF THE ROOM                   96.08%
UTIL. OF MACHINE                    99.98%
```

5. a. The first change is simply to the STORAGE statement. It now is:

```
         STORAGE    S(CHAIR),5
```

The results of the simulation are:

```
NUMBER TO ARRIVE AT SHOP          3847
NUMBER TO LEAVE WITH NO SERVICE   2396
UTIL. OF THE ROOM                   90.44%
UTIL. OF MACHINE                    99.98%
```

This says that having more waiting space is not the answer. Once the repair facility is full, the parts have to leave as before.

b. The change now is to the ADVANCE FN(SERVICE). It becomes:

```
ADVANCE  FN(SERVICE)/2
```

The new results of the simulation are:

```
NUMBER TO ARRIVE AT SHOP          3853
NUMBER TO LEAVE WITH NO SERVICE    951
UTIL. OF THE ROOM                   92.20%
UTIL. OF MACHINE                    99.98%
```

c. This time the ADVANCE is changed to ADVANCE FN(SERVICE)/2.5

The results are:

```
NUMBER TO ARRIVE AT SHOP          3851
NUMBER TO LEAVE WITH NO SERVICE    223
UTIL. OF THE ROOM                   98.54%
UTIL. OF MACHINE                    99.90%
```

There are still too may parts being sent away.

d. The ADVANCE block is changed to ADVANCE FN(SERVICE)/3.0

The new results are:

```
NUMBER TO ARRIVE AT SHOP          3850
NUMBER TO LEAVE WITH NO SERVICE      0
UTIL. OF THE ROOM                    0.13%
UTIL. OF MACHINE                    88.50%
```

6. The program for the simulation is:

```
              SIMULATE
      WORKER  FUNCTION    RN1,D7
.05,20/.17,21/.35,22/.56,23/.81,24/.96,25/1,26
       PAINT  FUNCTION    RN1,D5
.1,6/.25,7/.7,8/.93,9/1,10
      NUMBER  GENERATE    ,,,2        PROVIDE WORKERS
      UPTOP   ADVANCE     FN(WORKER)  ASSEMBLE A WIDGET
              ADVANCE     1,.4        TAKE WIDGET TO PAINT AREA
              QUEUE       WAIT        QUEUE FOR PAINT SHOP
              SEIZE       PAINT       USE THE PAINT SHOP
              DEPART      WAIT        LEAVE THE QUEUE
              ADVANCE     FN(PAINT)   PAINT A WIDGET
              RELEASE     PAINT       FREE THE PAINTER
              ADVANCE     1,.4        RETURN TO WORK STATION
              TRANSFER    ,UPTOP      READY TO MAKE ANOTHER WIDGET
              GENERATE    480*20      SIMULATE FOR 20 SHIFTS
              TERMINATE   1
              START       1
              CLEAR
NUMBER    GENERATE    ,,,2        RUN FOR TWO WORKERS
              START       1
              PUTPIC      LINES=2,N(NUMBER),(N(WIDGET)*12.75)/100_
                          -9.5*8*N(NUMBER)-75
NUMBER OF WORKERS ***
PROFIT PER SHIFT ****.**
```

```
              CLEAR
              NUMBER      GENERATE  ,,,3
              START       1
              PUTPIC      LINES=2,N(NUMBER),(N(WIDGET)*12.75)/100_
                             -9.5*8*N(NUMBER)-75
NUMBER OF WORKERS ***
PROFIT PER SHIFT ****.**
              CLEAR
              NUMBER      GENERATE  ,,,4
              START       1
              PUTPIC      LINES=2,N(NUMBER),(N(WIDGET)*12.75)/100_
                             -9.5*8*N(NUMBER)-75
NUMBER OF WORKERS ***
PROFIT PER SHIFT ****.**
              CLEAR
              NUMBER      GENERATE  ,,,5
              START       1
              PUTPIC      LINES=2,N(NUMBER),(N(WIDGET)*12.75)/100_
                             -9.5*8*N(NUMBER)-75
NUMBER OF WORKERS ***
PROFIT PER SHIFT ****.**
              CLEAR
              NUMBER      GENERATE  ,,,6
              START       1
              PUTPIC      LINES=2,N(NUMBER),(N(WIDGET)*12.75)/100_
                             -9.5*8*N(NUMBER)-75
              END
```

The table below summarizes the results of the simulation and the cost calculation results.

Workers	Widgets Made/Day	Gross Profit	Fixed Costs	Workers' Salaries	Net Profit
2	28.75	$367	$75	$152	$140
3	42.75	$545	$75	$228	$242
4	54.35	$706	$75	$304	$327
5	59.60	$760	$75	$380	$305
6	59.60	$760	$75	$456	$229

As can be seen, the optimum number of workers to have is 4. This results in a profit/day of $327.

7. The following changes need to be made to the program:

   A storage of size 2 needs to be added:

```
              STORAGE     S(PAINT),2
```

   The blocks SEIZE PAINT and RELEASE PAINT now become ENTER PAINT and LEAVE PAINT. The number of workers also needs to be increased in the simulation. The results of the simulation follow.

Number of Workers	Profit/Shift
5	$461
6	$564
7	$659
8	$738
9	$749
10	$688
11	$615

8. The program to simulate the uniform distribution is:

```
          SIMULATE
          GENERATE   5,3
          QUEUE      WAIT
          SEIZE      SERVER
          DEPART     WAIT
          ADVANCE    4,1
          RELEASE    SERVER
          TERMINATE
          GENERATE   480*500
          TERMINATE  1
          START      1
          PUTPIC     LINES=3,FC(SERVER)/500.,FR(SERVER)/10,QA(WAIT)
PEOPLE WHO COME IN PER SHIFT  *******.**
UTIL. OF SERVER                   ***.**%
AVERAGE QUEUE LENGTH              ***.**
          END
```

The changes to the program for using the normal distribution are to replace the GENERATE and ADVANCE blocks with:

```
          GENERATE   RVNORM(1,5,1)
```

and

```
          ADVANCE    RVNORM(1,4,.7)
```

For the exponential distribution, the following are used:

```
          GENERATE   RVEXPO(1,5)
```

and

```
          GENERATE   RVEXPO(1,4)
```

The results of the simulation are:

Number	Uniform Dist.	Normal Dist.	Exp. Dist.
Arrive/Shift	95.70	95.93	95.03
Util. of Worker	79.80	79.97	79.67
Average Queue	0.17	0.05	2.97

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 15

More on Standard Numerical Attributes: Arithmetic in GPSS/H

We have learned that every time a transaction encounters a certain block such as a QUEUE, ENTER, or SEIZE block, certain statistics are gathered and kept for printing out at the end of the program. These are known as standard numerical attributes (SNAs) and can be used by the programmer in other blocks when the program is being run. When the QUEUE, SEIZE, and ENTER blocks were introduced (Chapters 10–12), the various SNAs associated with them were listed, but those SNAs have not been used much to this point. There are, however, many uses for them that will become apparent as more GPSS/H blocks are presented. For example, the length of a queue may be used to determine whether a transaction will enter the queue. A facility is either in use or idle, as denoted by a 1 or a 0. If a facility is being used, a transaction may be sent to a different block. Another facility might work faster or slower as its utilization increases. We shall learn how to use these SNAs to greatly increase our programming skills.

A FEW OTHER SNAs

There are many other SNAs. Appendix B gives a list of the SNAs used in this book. Some have been encountered already without specifically referring to them as such. These are known as “system SNAs.”

FRN1	This SNA returns a fractional random number between 0.0 and 1.0. The endpoints 0.0 and 1.0 are *excluded*.
K*j*	In older versions of GPSS, every constant needed to be listed as K*j*, where *j* was the value of the constant. Thus, one would have GENERATE K480, ADVANCE K30,K5. This approach is no longer required, but GPSS/H still supports it.
M1	The SNA M1 is used to track how long a transaction has been in the system. When a transaction enters the system, the transaction is tagged with the time of entry. Whenever M1 is then referenced, the transaction's time of entry is subtracted from the current clock value. M1 is the difference between these two times. Suppose the time of entry was 5040.1234, and when M1 is referenced, the clock is at 5880.1235. M1 will be 840.0001. M1 is a *floating-point* number.
N*i*	The SNA N*i* is the total number of transactions that have entered the block with the label *i*.
TG1	The SNA TGI is the current value of the termination counter.
RN*j*	RN*j* indicates a random number from 0 to 1. This can also be written as RN(*j*). This SNA was used in defining functions. If it is used in connection with a function, the value returned is from the interval [0, 1) i.e., from 0.000000 to 0.999999. If used in any other context, the value returned is from 000 to 999.
W*i*	W*i* is the number of transactions currently at the block with the label *i*. If there are 4 cars in the QUEUE LINE block, W*i* = 4. This result is exactly the same as that of the SNA Q(LINE). However, most blocks do not have such an SNA, so the W*i* must be used.
XID1	Every transaction has a unique ID number. This is given at birth! This SNA is very important when studying animation.

MATHEMATICAL FUNCTIONS

GPSS/H supports the following mathematical functions. The term "xpr" refers to the expression used with each. In all cases for trigonometric functions, the xpr *must* be in radians.

ABS(xpr)	Absolute value. The mode for xpr is retained.
ACOS(xpr)	Arc cosine.
ASIN(xpr)	Arc sine.
ATAN(xpr)	Arc tangent.
COS(xpr)	Cosine of angle.
EXP(xpr)	e to the power xpr; xpr is real.
FIX(xpr)	Convert xpr to fixed point.
FLT(xpr)	Convert xpr to floating point.
LOG(xpr)	Natural log of xpr; xpr is real.
SIN(xpr)	Sine of angle.
TAN(xpr)	Tangent of angle.

ARITHMETIC EXPRESSIONS IN GPSS/H

It was mentioned in Chapter 5 that SNAs can be used in operands in arithmetic expressions. The following operations are used in GPSS/H:

+	unary and binary addition
−	unary and binary subtraction
/	division
*	multiplication
@	modular division

The above operations are all familiar to us with the possible exception of modular division. This is defined as division after which only the remainder is kept. For example, 7 @ 4 is 3 since the remainder is 3, 9 @ 10 is 9 (0 with a remainder 9), etc. Thus, one could have

```
         ADVANCE   N(BLOCKA)*Q(FIRST)+Q(LAST)*3.5
```

Since the arithmetic operations in GPSS/H are so similar to those encountered in other programming languages, not much more will be said of them at this time. However, care must be taken to guard against round-off (actually, truncation) error when integer division is used. For example, ADVANCE 5/2 will result in a transaction being placed on the FEC for 2 time units (in integer division, 5/2 = 2), but ADVANCE 5./2 will place the transaction on the FEC for 2.5 time units. Notice that *GPSS/H requires only one of the numbers to be floating point to produce a floating-point result.* In the case where two integers are used in division and a floating-point result is desired, the function FLT is used.

The use of SNAs will greatly expand one's ability to build meaningful simulation models. As additional blocks are introduced, it will become even more apparent how useful they are in writing simulation models. Most of the programs from now on will make use of SNAs. Several examples are given next. Some might appear to be quite fanciful, but they illustrate the extreme power and flexibility of the language.

```
TRUCK    GENERATE  ,,,4
         ADVANCE   60*N(TRUCK)-60
```

Four transactions will leave the GENERATE block at time $t = 0$. The first is put on the future events chain for a time of 60*N(TRUCK)-60. The block count when the first transaction has left is 1. Therefore, the time on the FEC is 0. The second transaction will be put on the FEC for 60 time units, the third for 120 time units, and the fourth for 180 time units. The effect of this ADVANCE block is to delay the entry of each of the transactions after the first by a period of 60 time units multiplied by N.

```
RAMPA    SEIZE     TOMMY
         -----
         -----
         ADVANCE   N(RAMPA)*2
```

Transactions entering the ADVANCE block will be put on the FEC for a time equal to 2 times the number of transactions that have entered the block with the label RAMPA. The preceding ADVANCE block also could have been written

```
◇         ADVANCE    FC(TOMMY)*2
```

because FC(TOMMY) gives the number of times the facility TOMMY has been captured.

```
◇         TERMINATE  W(BLOCKC)
```

The counter given by the START *A* statement is decremented by the amount equal to the current block count at the block with the label BLOCKC.

```
◇         ADVANCE    2.5*AC1
```

The transaction will be put on the FEC for a time equal to 2.5 times the absolute clock value.

```
◇         ADVANCE    QA(LINE)
```

The transaction will be placed on the FEC for a time equal to the integer portion of the average queue content of the queue named LINE.

```
◇         ADVANCE    Q(STORE)
```

A transaction entering the ADVANCE block will be placed on the FEC for a time equal to the length of the queue STORE.

Example 15.1

Customers arrive at Joe's barbershop every 15 ± 6.5 minutes. This distribution is constant throughout the day. If no customers are waiting, Joe will tend to take his time cutting hair. As customers arrive and fill up the shop, Joe will speed up his hair cutting. The time it takes Joe to cut hair is given by the following:

People in Queue	Time to Give Haircut (minutes)
0	18
1	16
2 or 3	14
4 or 5	13
more than 5	12

Simulate the operation of Joe's barbershop for 5 straight shifts of 480 minutes each. Determine how busy Joe is for an 8-hour shift. The program to do the simulation is as follows:

```
◇         SIMULATE
◇TIME     FUNCTION   Q(WAIT),D7
0,18/1,16/2,14/3,14/4,13/5,13/6,12
◇         GENERATE   15,6.5        PEOPLE ARRIVE
◇         QUEUE      WAIT          IN WAITING AREA
◇         SEIZE      JOEB          ENGAGE JOE FOR HAIRCUT
```

```
◇       DEPART     WAIT          LEAVE WAITING AREA
◇       ADVANCE    FN(TIME)      CUT HAIR
◇       RELEASE    JOEB          FREE THE BARBER
◇       TERMINATE                LEAVE THE SHOP
◇       GENERATE   480*5         SIMULATE FOR 5 SHIFTS
◇       TERMINATE  1             END OF SIMULATION
◇       START      1
◇       PUTPIC     LINES=5,FLT(FR(JOEB))/10,FC(JOEB)/20,_
                   QM(WAIT),QA(WAIT),FT(JOEB)
JOE WAS BUSY  ***.**% OF THE TIME
THE NUMBER OF HAIRCUTS/DAY WAS ****.**
THE MAXIMUM NO. OF PEOPLE WAITING WAS ***
THE AVERAGE NUMBER IN THE QUEUE WAS ***.**
AVERAGE TIME FOR A HAIRCUT WAS ****.**
        END
```

The results of the simulation are

```
JOE WAS BUSY  99.87% OF THE TIME
THE NUMBER OF HAIRCUTS/DAY WAS       31.90
THE MAXIMUM NO. OF PEOPLE WAITING WAS 5
THE AVERAGE NUMBER IN THE QUEUE WAS   2.02
AVERAGE TIME FOR A HAIRCUT WAS       15.03
```

As can be seen, Joe is kept quite busy since his time to give a haircut is often slower than the arrival rate of his customers. The maximum content of the queue was 5, and the average time to give the haircuts was 15.03. Notice that the function FLT was used to convert the FC(JOEB) from an integer to a floating point. If this was not done, the result would have been 31.00 and not 31.90.

EXERCISES, CHAPTER 15

1. What will the following blocks or statements do?

```
◇          ADVANCE    Q(FIRST)*Q(SECOND)/3    (a)
◇          GENERATE   10,2,,,FR(MACH1)        (b)
◇          GENERATE   5,1,,,FRN1*100          (c)
◇DRILL     FUNCTION   FR(MACH),D4             (d)
250,5/600,8/700,9/1000,12
◇TOP       FUNCTION   N(BLOCKA),L5            (e)
,5/,6/,9/,10/,15
```

2. Would the following block compile correctly? What would it do in the program?

```
◇          TRANSFER   ,Q(WAIT)+1
```

SOLUTIONS, CHAPTER 15

1. a. The transaction is placed on the FEC for a time given by the product of the queue length at FIRST times the queue length of SECOND. This product is divided by 3. The result is truncated to an integer.

   b. Transactions are generated every 10 ± 2 time units. Each will have a priority given by the fractional utilization in parts per thousand of the facility MACH1.

c. Transactions are generated every 5 ± 1 time units. Each is given a priority given from 0 to 99. (To have priorities from 1 to 100, the number 1 needs to be added to FRN1*100.)

d. When the function DRILL is called, it will return a value of 5, 8, 9, or 12 depending on the fractional utility of the facility MACH.

e. The function TOP is a list function and will return either 5, 6, 9, 10, or 15 depending on the total block count of the block BLOCKA. This is the same function as one with the integer 1, 2, 3, 4, and 5 in the first position.

2. The block will compile correctly but, if the value of Q(WAIT) was 0 when a transaction arrived, it would be routed to block 1. This is most probably a GENERATE block, so a run time error will occur.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 16

The *TEST* Block

So far transactions have moved through the various systems sequentially from block to block. The only way we had to route them to different blocks was via the TRANSFER block. There are many times in a model when the programmer will want to route a transaction to one block or another depending on some aspect of the system. There also will be times when the programmer will want to keep transactions from moving forward until a specific condition is met. One way to do this is by means of the TEST block.

It is possible to do a test on two SNAs and then route the transaction to one of two blocks depending on the result of the test. Examples of where a TEST block might be used arise frequently during a simulation. Some possible examples where a TEST block might be used are

1. If the queue at a shop is greater than 5, arriving customers do not enter.
2. After 12 hours of working, a machine is shut down for maintenance for ½ hour.
3. At 5:00, the barber locks the door on his shop, but customers already in are still served.

GPSS/H can make tests to determine (1) how long the queue is, (2) how long the machine has been in use, and (3) whether it's before or after 5 o'clock. GPSS/H also can perform a test on two SNAs and hold the transaction at the block doing the test until the test is true. Examples of the usefulness of this approach include:

1. A ship cannot enter the harbor until 1 of 2 tugboats is free to guide it into the berth.
2. A part will not move from one machine unless the next machine has been used less than 75% of the time.
3. If it is between noon and 12:30 p.m., no customer can enter a repair facility.

4. Once a machine has finished making 500 parts, it is taken out of service for repairs and maintenance. This downtime lasts for 2 hours. Parts arriving have to wait until the repairs and maintenance are finished.

The use of TEST blocks will greatly expand our programming ability. There are two basic forms of the TEST block, described next.

THE TEST BLOCK IN REFUSAL MODE

This form of the TEST block is as follows:

```
◇          TEST R     A,B
```

where *R* is a *relational operator* that is one of the following:

Symbol	Meaning
L	less than
LE	less than or equal
E	equal
NE	not equal
G	greater than
GE	greater than or equal

The relational operator *must* be placed 1 space after the operation TEST. *A* and *B* are any SNAs to be tested via the relational operator. Some examples of the TEST block are

```
◇(a)       TEST E     Q(TOM),Q(BILL)
◇(b)       TEST NE    R(DOCK),4
◇(c)       TEST L     FR(MACH),400
◇(d)       TEST G     W(BACK1),1
◇(e)       TEST E     N(BLOCKA),N(BLOCKB)
◇(f)       TEST GE    AC1,480
◇(g)       TEST E     Q(ONE)+Q(TWO),5
```

The way the TEST block works when a transaction enters it is that the first SNA is compared with the second by using the relational operator. If the test is true, the transaction moves to the next sequential block. If the test is false, the transaction *must wait in the TEST block until some future time when the test becomes true.* In addition, the transaction waits on the CEC. Thus, in (a), the test is "Is the length of the queue named TOM equal to the length of the queue named BILL?" If the answer is yes, the transaction will move to the next block, but if the answer is no, the transaction will remain in the TEST block until such time that the test is true.

In (b), the remaining storage of DOCK must not be equal to 4 before the transaction can move to the next block. Similarly, for (c), the fractional utilization of the facility MACH must be less than 0.400 or the transaction will reside in the TEST block until that condition is attained. In example (d), the transaction will test to see if the current count of the block BACK1 is greater

than 1. Unless the current count is greater than 1, the transaction will not leave the TEST block.

In (e), the transaction will be held until the total block count of the block labeled BLOCKA is equal to the count of the block labeled BLOCKB. In (f), the AC1 must be greater than 480 before the transaction can move to the next sequential block. In (g), the sum of the queue contents of queues ONE and TWO must equal 5 before the test is true.

Example 16.1

This example should be studied to understand how a TEST block in refusal mode can be used. Joe cuts hair in 15 ± 6 minutes. Customers arrive every 14 ± 8 minutes. Joe has only one chair for them to wait in, so if a customer arrives to find this taken, he will leave. Of the people who leave, 30% will be gone for 30 ± 12 minutes and then return to see if the waiting-area chair is free. (If the chair in the waiting area is not available this second time, he will again leave; 30% of the customers who checked back for the second time will, after 30 ± 12 minutes, return for a third attempt, etc.). The rest will go elsewhere. Joe works from 8:00 to 5:00 with no time off for lunch. At 5:00, Joe locks the door, but will finish cutting the hair of anyone in his shop. Simulate for a typical day.

Solution

```
         SIMULATE
         STORAGE    S(WAITAREA),1
COMEIN   GENERATE   14,8                  CUSTOMERS ARRIVE
         TEST L     N(TIME),1             IS IT PAST 5 O'CLOCK YET?
BACK     TRANSFER   BOTH,,AWAY            IS THERE A CHAIR TO WAIT IN?
INSHOP   ENTER      WAITAREA              TAKE A CHAIR TO WAIT IN
         SEIZE      JOEB                  ENGAGE JOE
         LEAVE      WAITAREA              LEAVE WAITING-AREA CHAIR
         ADVANCE    15,6                  RECEIVE HAIRCUT
NDONE    RELEASE    JOEB                  FREE JOE
         TERMINATE                        LEAVE THE SHOP
AWAY     TRANSFER   .7,,GONE              70% LEAVE
         ADVANCE    30,12                 30% WAIT
         TRANSFER   ,BACK                 RETURN TO SHOP
GONE     TERMINATE                        LEAVE SYSTEM
TIME     GENERATE   480                   5 O'CLOCK COMES
         TEST E     N(NDONE),N(INSHOP)    IS SHOP EMPTY?
         TERMINATE  1                     YES, JOE CAN GO HOME
         START      1
         PUTPIC     LINES=6,AC1,N(COMEIN),FR(JOEB)/10.,_
                N(NDONE),N(GONE),N(AWAY)
 SIMULATED TIME SHOP WAS OPEN   ****.**
 NUMBER TO COME TO SHOP ***
 JOE WAS BUSY           ***.**%
 NUMBER OF HAIRCUTS     ***
 NUMBER TO GO AWAY AND NOT TRY TO COME BACK      ***
 NUMBER TO GO AWAY AND TRY TO COME BACK          ***
         END
```

There are several TEST blocks in the program. The first (TEST L) is used to stop any customer transactions from entering the shop after 5:00. The simulation starts at time $t = 0$ and, with 1 minute as the basic time unit, 5:00 will be given by time $t = 480$. At time $t = 480$, no more customers will be allowed in the shop. This block is

```
◇        TEST L    N(TIME),1
```

At simulated time $t = 480$, a transaction leaves the following block:

```
◇TIME    GENERATE  480
```

This event makes the block count 1 for the block labeled TIME. The test result of the TEST block is then false, and any transactions that enter it will be held up and not be allowed to proceed. This procedure corresponds to not allowing any customers to enter the shop after 5:00. Note that an alternate block to use would have been

```
◇        TEST L    AC1,480
```

The second TEST block (TEST E) is the timer transaction segment of the program. At time $t = 480$, the timer transaction arrives to shut off the program. This corresponds to 5:00. However, the transaction first enters the TEST block

```
◇        TEST E    N(NDONE),N(INSHOP)
```

INSHOP is the label for the ENTER block, which corresponds to a customer entering the shop. N(NDONE) corresponds to the total number of customers who have left the shop. It is the label for the RELEASE JOEB block. Suppose that, at simulated time $t = 480$, N(INSHOP) was 33 and N(NDONE) was 31. The effect of the second TEST block would then be to hold the timer transaction until all of the customers in the shop had been given haircuts by Joe.

The output from the simulation is as follows:

```
SIMULATED TIME SHOP WAS OPEN   496.72
NUMBER TO COME TO SHOP  33
JOE WAS BUSY            91.52%
NUMBER OF HAIRCUTS      30
NUMBER TO GO AWAY AND NOT TRY TO COME BACK      2
NUMBER TO GO AWAY AND TRY TO COME BACK          2
```

Joe was busy 91.52% of the time. A total of 30 customers were able to enter the shop and receive haircuts. Joe worked until time 496.72 or 16.72 minutes past the normal closing time. The program should be rerun by using a RMULT to see the effect of using different random numbers in the simulation.

Whenever a TEST block in refusal mode is used in a program, great care must be exercised that the transaction does not remain in the block forever if this is not the programmer's desire. There is another caution in using this block that we have not been too concerned with up to this time. Whenever a transaction is in a blocked condition at a TEST block, it remains on the current events chain. Whenever the processor does a rescan, this block must be tested. This constant testing can be quite costly in terms of execution time. In some cases,

there will be ways to avoid using such inefficient blocks. The TEST block is both convenient and easy to understand. However, if it is possible to avoid using it, alternative programming should be used. Some other blocks that might be used in its place will be introduced in Chapter 21. In some cases, however, there is no method available other than the TEST block.

TEST BLOCK IN NORMAL MODE

The other form of the TEST block has a *C* operand. The form of it is

```
          TEST R    A,B,C
```

where *C* is the label of the block to which the transaction is routed if the test is *false*. Thus,

```
          TEST E    Q(TOMMY),Q(SALLY),DOWN
```

will test the length of the queue TOMMY and the queue SALLY. If they are equal, the transaction will go to the next sequential block. If they are unequal, the transaction will go to the block named DOWN. For programmers who are used to the logic of Fortran, the way GPSS/H works for the TEST block is going to seem to be quite the opposite to what one would expect. Thus, *great care is required when using this block.*

Example 16.2

In a manufacturing process, parts come to a machine for forming. The inter-arrival rate is 14 ± 7.5 minutes. There are 2 machines available for forming. The first can form in 16 ± 5.4 minutes, and the other takes considerably longer: 24 ± 8 minutes. In fact, this second machine is in such poor condition that it is not used until the first machine is utilized to its fullest. The faster machine cannot be used more than 85% of the time, however, or it may over-heat. Parts enter the room where both machines are located and are formed on the faster machine until it reaches the 85% utilization, at which time the slower machine is also used until the utilization is again below 85%. Build a GPSS/H model to represent the system. Simulate for 100 straight shifts of 8 hours (480 minutes).

Solution

```
          SIMULATE
          GENERATE  14,7.5
          QUEUE     WAIT
          TEST LE   FR(MACH1),850,DOWN1
          SEIZE     MACH1
          DEPART    WAIT
          ADVANCE   16,5.4
DONE1     RELEASE   MACH1
          TERMINATE
DOWN1     SEIZE     MACH2
          DEPART    WAIT
          ADVANCE   24,8
DONE2     RELEASE   MACH2
          TERMINATE
```

```
◇        GENERATE   480*100
◇        TERMINATE  1
◇        START      1
◇        PUTPIC     LINES=6,FR(MACH1)/10.,FR(MACH2)/10.,QM(WAIT),_
                    QA(WAIT),N(DONE1)/100.,N(DONE2)/100.
 UTIL. OF MACHINE 1  ***.**%
 UTIL. OF MACHINE 2  ***.**%
 MAX. QUEUE          ***
 AVERAGE QUEUE       ***.**
 DONE BY MACHINE 1   ***.**    (PER SHIFT)
 DONE BY MACHINE 2   ***.**    (PER SHIFT)
◇        END
```

The results of the simulation are as follows:

```
UTIL. OF MACHINE 1   85.01%
UTIL. OF MACHINE 2   45.06%
MAX. QUEUE           4
AVERAGE QUEUE        0.60
DONE BY MACHINE 1    25.37    (PER SHIFT)
DONE BY MACHINE 2     9.02    (PER SHIFT)
```

Machine 1 was busy for the maximum allowable time of 85.01%. The reason for the slightly higher percentage is that a transaction can use the machine when the percent utilization is less than 85%, and this additional usage increases the utilization to greater than 85%. The secondary machine was used only 45.06%. The maximum queue was 4, and the average queue was 0.60. Machine 1 produced 25.37 units per shift, and machine 2 produced 9.02 units per shift.

EXERCISES, CHAPTER 16

1. What do the following blocks do?

```
◇(a)     TEST E     Q(WAIT),3
◇(b)     TEST NE    Q(STOP1)*Q(STOP1),4
◇(c)     TEST G     AC1,5000
>(d)     TEST E     N(BLOCKA),N(AWAY)
◇(e)     TEST L     N(BLOCKC)35,DDDD
◇(f)     TEST GE    FR(MACH1),500
◇(g)     TEST E     S(TUGS1),R(TUGS2),BYBYE
```

2. A truck and single shovel system involves the following times:

Operation	Time (minutes)
load a truck	2 ± 0.3
travel to crusher	5 ± 1
dump into crusher	1 ± 0.2
return to shovel	4.5 ± 0.7

The mine currently has 6 trucks working. Only 1 truck can dump at a time. All the loads are ore, and none of the equipment ever breaks down. The mine works for 450 minutes per 480-minute shift, after which the

drivers bring their trucks to the shovel and then leave for the day. The mine works 3 shifts per day. Simulate for 100 shifts to determine the production.

3. In exercise 2, suppose that the actual working time is 400 minutes per shift. Determine how this fact changes the production per shift.
4. For the mine in exercise 2, it is discovered that 20% of the loads are waste and need to be hauled to a waste area. Travel time there is obtained by sampling from a normal distribution with a mean of 6.5 minutes and a standard deviation of 1 minute. Dumping has no restriction in that a truck can dump as soon as it is at the waste area. Return time to the shovel is obtained by sampling from a normal distribution with a mean of 5.3 minutes and a standard deviation of 1.1 minutes. Determine the production of ore and waste per shift.
5. A car repair shop has room for only 2 vehicles to wait while it repairs other cars. Cars that arrive when the shop is full leave and do not return. There are two repair bays, so there can be as many as 4 cars at the shop. Cars arrive every 30 ± 12.5 minutes, and it takes 58 ± 20 minutes to repair each car. The shop is open 24 hours a day. Simulate for 500 days to determine how many cars arrive when the shop is full. Suppose that there could be room for another car to wait. Will this additional space result in much of a change in how many cars arrive when the repair shop is full?

SOLUTIONS, CHAPTER 16

1. a. The transaction is delayed in the TEST block until the length of the queue at WAIT is equal to 3. Once this is true, it passes to the next sequential block.

   b. The transaction is delayed in the TEST block if the product of the queues, STOP1 and STOP2, are equal numbers other then 4; else it moves to the next sequential block.

   c. The value of the absolute clock, AC1, must be greater than 500 before the transaction moves to the next sequential block.

   d. The number of transactions that have entered the block with the label BLOCKA is compared to the number that have entered the block with the label BLOCKB. If they are equal, the transaction moves to the next sequential block; else it is routed to the block with the label AWAY.

   e. The number of transactions that have entered the block with the label BLOCKC must be less than 35 in order for the transaction to move to the next sequential block; else it is routed to the block with the label DDDD.

   f. The fractional utilization of the facility MACH1 must be greater than or equal to 500 before the transaction moves to the next sequential block.

   g. The storage used as storage TUGS1 must be equal to the remaining storage of the storage TUGS2 in order for the transaction to move to the next sequential block; else the transaction is routed to the block with the label BYBYE.

2. The program to do the simulation is:

```
        SIMULATE
        GENERATE    ,,,6
UPTOP   QUEUE       WAIT
        SEIZE       SHOVEL
        DEPART      WAIT
        ADVANCE     2,.3
        RELEASE     SHOVEL
        ADVANCE     5,1
        SEIZE       CRUSHER
        ADVANCE     1,.2
        RELEASE     CRUSHER
        ADVANCE     4.5,.7
        TRANSFER    ,UPTOP
        GENERATE    480*100
        TERMINATE   1
        START       1
        PUTPIC      LINES=3,FR(SHOVEL)/10.,FC(CRUSHER)/100.0
        SOLUTION TO EXERCISE 16.2
  SHOVEL WAS BUSY ***.**% OF THE TIME
  PRODUCTION PER SHIFT IS    ***.**  LOADS DUMPED
        END
```

The results of the simulation are:

```
      SOLUTION TO EXERCISE 16.2
SHOVEL WAS BUSY  91.18% OF THE TIME
PRODUCTION PER SHIFT IS    204.96  LOADS DUMPED
```

3. The changed program is:

```
        SIMULATE
        GENERATE    ,,,6
UPTOP   QUEUE       WAIT
        SEIZE       SHOVEL
        DEPART      WAIT
        ADVANCE     RVNORM(1,2,.3)
        RELEASE     SHOVEL
        TRANSFER    .2,,WASTE
        ADVANCE     RVNORM(1,5,1)
        SEIZE       CRUSHER
        ADVANCE     RVNORM(1,1,.2)
        RELEASE     CRUSHER
        ADVANCE     RVNORM(1,4.5,.7)
        TRANSFER    ,UPTOP
WASTE   ADVANCE     RVNORM(1,6.5,1)
        ADVANCE     RVNORM(1,1,.2)
        ADVANCE     RVNORM(1,5.3,1)
        TRANSFER    ,UPTOP
        GENERATE    450*100
        TERMINATE   1
        START       1
        PUTPIC      LINES=3,FR(SHOVEL)/10.,FC(CRUSHER)/100.0
        SOLUTION TO EXERCISE 16.3
  SHOVEL WAS BUSY ***.**% OF THE TIME
  PRODUCTION PER SHIFT IS    ***.**  LOADS DUMPED
        END
```

The results of the simulation are:

```
          SOLUTION TO EXERCISE 16.3
SHOVEL WAS BUSY  91.18% OF THE TIME
PRODUCTION PER SHIFT IS    182.21  LOADS DUMPED
```

4. The program to simulate this is:

```
          SIMULATE
          STORAGE    S(REPAIR),2
 COMEIN   GENERATE   30,12.5
          TEST L     Q(WAIT),2,AWAY
          QUEUE      WAIT
          ENTER      REPAIR
          DEPART     WAIT
          ADVANCE    58,20
          LEAVE      REPAIR
          TERMINATE
 AWAY     TERMINATE
          GENERATE   480*3*500
          TERMINATE  1
          START      1
          PUTPIC     LINES=4,N(COMEIN),N(AWAY),SR(REPAIR)/10.
          RESULTS OF SIMULATION....
   NUMBER OF CARS TO COME TO SHOP (500 DAYS)  ******
   NUMBER OF CARS TO LEAVE AS SHOP IS FULL    ******
   UTIL. OF REPAIR SHOP                          ***.**%
          END
```

The results of the simulation are:

```
          RESULTS OF SIMULATION....
NUMBER OF CARS TO COME TO SHOP (500 DAYS)  23997
NUMBER OF CARS TO LEAVE AS SHOP IS FULL      406
UTIL. OF REPAIR SHOP                          95.17%
```

Adding another space for a car to wait will change the number of cars that leave to 157, which is quite significant.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 17

GPSS/H Supplied Functions

In Chapter 14, functions were introduced. By using piecewise linear approximations, it is possible to approximate any continuous function. Two functions that are very commonly used in simulations are the Poisson (exponential) distribution and the normal (Gaussian) distribution. These arise in simulation studies of a great many systems. For example, the interarrival rates of telephone calls often is given by the exponential distribution, the times to travel from point A to point B by truck is normally distributed, the time for a ship to return to a port is exponential, the time between storms is exponential, etc. It is assumed that both of these functions are well known to students of simulation. Since these functions are so commonly referred to in simulation studies, they are built into GPSS/H, and sampling from them is quite easy.

THE POISSON DISTRIBUTION

The Poisson distribution is a one-parameter distribution being completely specified by its mean value. The built-in function to be used in sampling from it is given by

```
◇          RVEXPO(s,m)
```

where s is an integer that indicates which random-number stream is used and m is the mean of the exponential distribution. Recall that GPSS/H has nearly an infinite number of these streams, but one normally uses only small numbers such as 1, 2, 3, etc. (Note: RVEXPO comes from random variate exponential distribution.)

Examples of using the Poisson distribution function are

```
◇(a)       ADVANCE    RVEXPO(1,12.3)
◇(b)       GENERATE   RVEXPO(1,3.4)
```

In (a), the transaction is placed on the FEC for a time given by sampling from the exponential distribution with a mean of 12.3. In (b), transactions are generated at times given by sampling from the exponential distribution with a mean of 3.4.

THE NORMAL DISTRIBUTION

The normal distribution is a two-parameter distribution and is specified by the mean and standard deviation. The GPSS/H built-in function to sample from the normal distribution is given by

```
◇        RVNORM(s,m,d)
```

where, again, *s* is an integer that indicates which random-number stream is used, *m* is the mean of the normal distribution, and *d* refers to the standard deviation. RVNORM comes from random variate normal distribution. Examples of using the normal distribution function are

```
◇(a)     ADVANCE    RVNORM(1,20,2.3)
◇(b)     GENERATE   RVNORM(1,30,5.5)
```

In (a), the transaction is placed on the FEC for a time that is obtained by sampling from a normal distribution with a mean of 20 and a standard deviation of 2.3. In (b), transactions are generated according to the normal distribution with a mean of 30 and a standard deviation of 5.5.

GPSS/H samples from a distribution bounded by 44 standard deviations above and below the mean, so, while it is theoretically possible to obtain samples that are negative, such an occurrence is rare.

THE TRIANGULAR DISTRIBUTION

GPSS/H has another built-in function that represents the triangular distribution. This distribution is one that looks like a triangle with one side on the *x*-axis. The side extends from a minimum value to a maximum value. The most likely value is the mode. A triangular distribution with a minimum value of 10, a maximum value of 100, and a mode of 20 will be skewed to the left. A triangular distribution with minimum value of 10, maximum value of 100, and a mode of 80 will be skewed to the right. If the mode of this distribution was 55, the distribution would be symmetrical. To sample from a triangular distribution in GPSS/H, one uses the built-in function

```
◇        RVTRI(s,minimum,mode,maximum)
```

where again *s* is an integer that indicates which random-number stream is used. RVTRI comes from random variate triangular distribution. Thus,

```
◇        ADVANCE   RVTRI(2,0,3,10)
```

will sample from the triangular distribution having a minimum value of 0, a mode of 3, and a maximum value of 10.

OTHER BUILT-IN FUNCTIONS

The other built-in functions are given below. In every case, *s* is an integer that indicates which random-number stream is used, i.e., $s = 1$ implies the use of random-number stream 1 (RN1), etc. Unless specified as an integer, every parameter in the function is real. For the exact definition of the function as well as the meaning of the parameters, see any detailed textbook on statistics.

Function	Form and Explanation
Beta	RVBETA(s,α_1,α_2), where α_1 is shape parameter 1, α_2 is shape parameter 2, and α_1 and α_2 > 0.0
Binomial	RVBIN(s,no._of_trials,probability), where no. of trials is an integer >0 and probability is in the interval (0,1.0)
Discrete Uniform	RVDUNI(s,lower_endpoint,upper_endpoint), where lower and upper endpoints are integers; lower endpoint ≤ upper endpoint
M-Erlang	RVERL(s,*m*,β), where *m* is an integer (the order of the distribution) and β is mean value
Extreme Value A	RVEVA(s,γ,β), where γ (location) has no limits and β (scale) > 0.0
Gamma	RVGAMA(s,α,β), where α (shape) > 0.0 and β (scale) > 0.0
Geometric	RVGEO(s,probability), where probability is in interval (0.0,1.0)
Inverse Gausian	RVIGAU(s,α,β), where α (shape) > 0.0 and β (scale) > 0.0
Inverted Weibull	RVIWEIB(s,α,β,γ), where α (shape) > 0.0, β (scale) > 0.0, and γ (location) has no limit
Bounded-Johnson	RVJSB(s,α_1,α_2), where α_1 has no limit and α_2 > 0
Unbounded-Johnson	RVJSB(s,α_1,α_2), where α_1 has no limit and α_2 > 0
Laplace	RVLAP(s,γ,β), where γ (location) has no limit and β (scale) > 0.0
Logistic	RVLGTC(s,γ,β), where γ (location) has no limit and β (scale) > 0.0
Log-Laplace	RVLLP(s,α,β), where α (shape) > 0.0 and β (scale) > 0.0
Lognormal	RVLNOR(s,mean,variance), where mean > 0.0 and variance > 0.0
Negative Binomial	RVNBIN(s,successes,probability), where "successes" is an integer >0 and probability is −∞ to <1.0
Poisson	RVPSSN(s,mean), where mean > 0.0
Pearson Type V	RVPT5(s,α,β), where α (shape) > 0.0 and β (scale) > 0.0
Pearson Type VI	RVPT6(s,α,β), where α (shape) > 0.0 and β (scale) > 0.0
Random Walk	RVRWK(s,α,β), where α (shape) > 0.0 and β (scale) > 0.0
Uniform	RVUNI(s,mean,spread), where mean has no limit; spread has no limit but must be less than or equal to mean
Weibull	RVWEIB(s,α,β), where α (shape) > 0.0 and β (scale) > 0.0

USING EXPONENTIAL AND NORMAL DISTRIBUTIONS OTHER THAN THE BUILT-IN ONES

Before the exponential and normal distributions were built in, one had to sample from a 24-piecewise linear approximation to them.

The exponential distribution can be put in a form so that given a random sample in the interval [0,1), a corresponding interarrival time is given by

$$T_{sample} = T_{mean} [(\ln 1) / (1 - RN)]$$

where
T_{mean} = mean value of distribution
RN = random number

The 24-piecewise approximation for the distribution (ln 1)/(1 – RN) to be used in sampling from the exponential distribution is as follows:

```
◇EXPDIS  □FUNCTION  □RN1,C24
0,0/.1,.104/.2,.222/.3,.355/.4,.509/.5,.69/.6,.915/.7,1.2/.75,1.38
.8,1.6/.84,1.83/.88,2.12/.9,2.3/.92,2.52/.94,2.81/.95,2.99/.96,3.2
.97,3.5/.98,3.9/.99,4.6/.995,5.3/.998,6.2/.999,7/.9998,8
```

Normally, one samples from the exponential distribution in either a GENERATE or an ADVANCE block. Some examples of how this can be done are as follows:

```
◇(a)     □GENERATE  □100,FN(EXPDIS)
◇(b)     □ADVANCE   □225,FN(EXPDIS)
```

In (a), a transaction is generated by sampling from the exponential distribution with a mean of 100. In (b), a transaction is placed on the FEC for a time specified by sampling from the exponential distribution with a mean of 225.

Recall that, when either a GENERATE or ADVANCE block has a *function* as the *B* operand, the effect is to *multiply* the value in the *A* operand by the value of the function. This is true *whenever* a function is in the *B* operand, *not* only for the exponential distribution. Thus, if you had

```
◇        □ADVANCE   □FN(ONE),FN(TWO)
```

and the value of FN(ONE) is 20 and that of FN(TWO) is 8, the transaction is put on the FEC for 160 time units, *not* for a time obtained by sampling from the distribution 20 ± 8.

The normal distribution is a two-parameter distribution. These parameters are the mean and standard deviation. A *standard* normal distribution is one whose mean is 0 and standard deviation is 1. To sample from such a distribution, one first samples from the standard normal distribution and then converts the result to the nonstandard distribution. This procedure is as follows:

$$\text{Sample value} = [(\text{std dev}) \times (\text{value from SNP})] + \text{mean}$$

The SNP is the value drawn from the standard normal population. The piecewise continuous function used in sampling from the SNP is

```
◇SNORM  □FUNCTION  □RN1,C25
0,-5/.00003,-4/.00135,-3/.00621,-2.5/.02275,-2
.06681,-1.5/.11507,-1.21/.15866,-1/.21186,-.8/.27425,-.6
.34458,-.4/.42074,-.2/.5,0/.57926,.2/.65542,.4
.72575,.6/.78814,.8/.84134,1/.88493,1.2/.93319,1.5
.97725,2/.99379,2.5/.99865,3/.99997,4/1,5
```

Examples of this are

```
◇(a)     ADVANCE    3.2*FN(SNORM)+20.7
◇(b)     ADVANCE    10*FN(SNORM)+100
◇(c)     GENERATE   2.2*FN(SNORM)+20.8
```

In (a), the transaction is placed on the FEC for a time given by sampling from the normal distribution with a mean of 20.7 and a standard deviation of 3.2. In (b), the transaction is placed on the FEC for a time given by sampling from the normal distribution with a mean of 100 and a standard deviation of 10. In (c), transactions are generated at times obtained from sampling from the normal distribution with a mean of 20.8 and a standard deviation of 2.2.

Notice that the function SNORM, used in sampling from the normal distribution, can return a value as low as −5. Suppose one had

```
◇        ADVANCE    2.1*FN(SNORM)+10
```

and just such a value was returned. The resulting time is $(-2.1 \times 5) + 10$ or −0.5 time units. This result is meaningless as one cannot go back in time, and an execution error would result. It is necessary to guard against such problems by always ensuring that the standard deviation is less than ⅕ (20%) of the mean. Alternatively, one can add other GPSS/H code to test for negative times and filter them out.

EXERCISES, CHAPTER 17

1. A careful study of a widget manufacturing facility showed that each widget is made from start to finish by a single person. This worker takes a raw form and turns it into a widget. After this is done, the worker takes the widget to a single finishing machine where it is polished and boxed for shipment. There are assumed to be an infinite number of raw forms available so that the widget maker always will be able to begin work on a new one as soon as one is done being boxed. The distribution of times for forming a widget is as follows:

Assembly Time (minutes)	Relative Frequency
25	0.01
26	0.03
27	0.05
28	0.10
29	0.18
30	0.26
31	0.18
32	0.10
33	0.05
34	0.03
35	0.01

The distribution of time to finish a widget is as follows:

Finishing Time (minutes)	Relative Frequency
6	0.05
7	0.25
8	0.40
9	0.25
10	0.05

Each widget manufactured earns the company a profit of $26.25. The plant has an overhead of $125 per day, and each worker earns $96 per day. The workers work 8-hour shifts. For the sake of this exercise, you can assume that the workers work continuously and that there is 1 shift per day. Determine the optimum number of workers to have making widgets.

2. The data for making a widget have been changed. Further careful study by you indicates that the assembly time is normally distributed with a mean of 30 and a standard deviation of 20%. The finishing time is also normally distributed with a mean of 8 and a standard deviation of 20%. Does the optimum number of workers change much?
3. Change the data in exercise 2 to be exponentially distributed with a mean of 30 for assembling and 8 for finishing. How does the answer change as the simulation is run with 4, 5, 6, etc., assemblers?
4. A manufacturing system consists of two waiting lines and two servers. Only 4 units can wait at station 1 and 2 at station 2. If another unit comes along when the waiting space at station 1 is full, it leaves and a penalty is incurred. Arrivals are exponentially distributed with a mean of 0.4 time units. Service times are exponentially distributed at both stations with means of 0.25 and 0.5, respectively. Model this system to see how efficient it is. Simulate for 5000 time units.
5. Change the station working times in exercise 4 to have means of 0.35 for station 1 and 0.40 for station 2. Note that their sum is still 0.75 as it was before. Is the system improved?
6. Repeat exercise 4, but now with waiting space for the stations allocated as 3 and 3.
7. Repeat exercise 6, but now with the service times changed as in exercise 2.
8. Repeat exercises 4 to 7, but with the interarrival and service times following an M-Erlang distribution with an order of 2. Recall that an M-Erlang distribution is one obtained by a Poisson distribution but with the mean divided by the order. Thus, if one had a Poisson distribution with a mean of 5, an M-Erlang distribution with an order of 4 would be 4 Poisson distributions, each with mean 1.25 (5 divided by 4).
9. Most queueing theory problems have no exact solutions. A few, however, do, and these can be studied to compare the simulation solution with the expected result obtained by using a formula. One such example is the exercise presented next.

A car-wash facility has 1 bay to wash cars. The cars arrive with an average interarrival time of 5 minutes. The washing time is 4 minutes. If a car arrives and there is no waiting space, it will leave and not return (or if it does return, it is considered as a "new" car). Determine the behavior of the system for 1, 2, and 3 waiting spaces.

The exact solution for this problem gives as the fractions served:

$$\text{fraction served} = 1 - \left(\frac{1-x}{1-x^{m+1}}\right)x^m$$

where x is the utilization factor, which is the ratio of the mean service time to the mean interarrival time, and m is the number of waiting spaces. Thus, for a single waiting space and $x = 4/5$ (as given in the setup), the formula gives 0.738 as the theoretical fraction served over an infinite time.

Determine how many simulated time units are needed for the simulation result to approach this number.

10. A particular worker tends to work at a progressively slower rate as the 8-hour day goes by. During the first 2 hours, it takes him 12 minutes to perform a service; during the next 2 hours, his average service time is 15 minutes; during the fifth, sixth, and seventh hours, each service takes him an average of 17 minutes; and service started during the eighth hour requires an average of 20 minutes. With a simulated time unit of 0.1 minute, define a discrete function to model this person's service time.
11. Change exercise 10 to a continuous function by assuming the following: at time $t = 0$, the service time is 12; by the end of the second hour, it is 15; at the end of the fourth hour, it is 17; by the end of the seventh hour, it is 20; and finally, at the end of the eighth and last hour, it is 21.

SOLUTIONS, CHAPTER 17

1. The program to do the simulation is:

```
       SIMULATE
MAKEIT FUNCTION   RN1,D10
.01,25/.04,26/.09,27/.19,28/.38,29/.64,30/.82,31/.92,32
.97,33/1,34
 FINISH  FUNCTION  RN1,D5
.05,6/.3,7/.7,8/.95,9/1,10
 WORKER GENERATE   ,,,6
 UPTOP  ADVANCE    FN(MAKEIT)
        SEIZE      FINISH
        ADVANCE    FN(FINISH)
        RELEASE    FINISH
        TRANSFER   ,UPTOP
        GENERATE   480*500
        TERMINATE  1
        START      1
        PUTPIC     LINES=4,N(WORKER),FR(FINISH)/10.,FC(FINISH)/500.,_
                     26.25*FC(FINISH)/500.-125-N(WORKER)*96
     NUMBER OF WORKERS          ****
     UTIL. OF FINISHING MACHINE  ***.**%
     WIDGETS MADE PER SHIFT      ***.**
     PROFIT                      ***.**
        END
```

The results of the simulation are:

```
NUMBER OF WORKERS              3
UTIL. OF FINISHING MACHINE   62.55%
WIDGETS MADE PER SHIFT       37.51
PROFIT                      571.81

NUMBER OF WORKERS              4
UTIL. OF FINISHING MACHINE   82.20%
WIDGETS MADE PER SHIFT       49.32
PROFIT                      786.81

NUMBER OF WORKERS              5
UTIL. OF FINISHING MACHINE   97.79%
WIDGETS MADE PER SHIFT       58.64
PROFIT                      934.90

NUMBER OF WORKERS              6
UTIL. OF FINISHING MACHINE   99.99%
WIDGETS MADE PER SHIFT       60.00
PROFIT                      873.92
```

The optimum number of workers to have is 5.

2. The changes to the program are minimal. The results of the simulation are summarized:

Workers	Profit
3	$540.09
4	$720.08
5	$848.81
6	$860.35
7	$775.59

Now the optimum number of workers to have is 6.

3. The results of the simulation are:

Workers	Profit
3	$483.00
4	$605.79
5	$684.71
6	$713.30
7	$689.12

As can be seen, the number of workers needed remains as 6, but the profit goes down dramatically.

4. The program to do the simulation is:

```
        SIMULATE
COMEIN  GENERATE   RVEXPO(1,.4)
        TEST L     Q(WAIT1),4,AWAY1
        QUEUE      WAIT1
        SEIZE      SERVE1
        DEPART     WAIT1
        ADVANCE    RVEXPO(1,.25)
        RELEASE    SERVE1
        TEST L     Q(WAIT2),1,AWAY2
        QUEUE      WAIT2
        SEIZE      SERVE2
        DEPART     WAIT2
        ADVANCE    RVEXPO(1,.5)
        RELEASE    SERVE2
        TERMINATE
AWAY1   TERMINATE
AWAY2   TERMINATE
        GENERATE    5000
        TERMINATE   1
        START       1
        PUTPIC      LINES=4,N(COMEIN),FR(SERVE1)/10.,QA(WAIT1),_
                    FR(SERVE2)/10.,QA(WAIT2),N(AWAY1),N(AWAY2)
NUMBER TO ENTER SYSTEM            ***
UTIL. OF SERVER 1  ***.**%  AVERAGE NO. IN QUEUE AT SERVER 1     *.**
UTIL. OF SERVER 2  ***.**%  AVERAGE NO. IN QUEUE AT SERVER 2     *.**
NO. TO NOT USE FIRST SERVER *** NO. TO NOT USE SECOND SERVER ****
        END
```

The results of the simulation are:

```
NUMBER TO ENTER SYSTEM            12405
UTIL. OF SERVER 1  59.18%   AVERAGE NO. IN QUEUE AT SERVER 1     0.66
UTIL. OF SERVER 2  73.38%   AVERAGE NO. IN QUEUE AT SERVER 2     0.40
NO. TO NOT USE FIRST SERVER 451  NO. TO NOT USE SECOND SERVER  4763
```

The number of parts that could not use either server is: 5214

5. The results from making these changes are:

```
NUMBER TO ENTER SYSTEM            12464
UTIL. OF SERVER 1  78.33%  AVERAGE NO. IN QUEUE AT SERVER 1      .41
UTIL. OF SERVER 2  62.41%  AVERAGE NO. IN QUEUE AT SERVER 2     0.29
NO. TO NOT USE FIRST SERVER 1497  NO. TO NOT USE SECOND SERVER 3041
```

The number of parts that could not use either server is: 4538

6. The results from making these changes are:

```
NUMBER TO ENTER SYSTEM            12561
UTIL. OF SERVER 1  59.09%  AVERAGE NO. IN QUEUE AT SERVER 1     0.50
UTIL. OF SERVER 2  87.46%  AVERAGE NO. IN QUEUE AT SERVER 2     1.52
NO. TO NOT USE FIRST SERVER 847   NO. TO NOT USE SECOND SERVER 3078
```

The number of parts that could not use either server is: 3925

7. The results from making these changes are:

```
NUMBER TO ENTER SYSTEM            12484
UTIL. OF SERVER 1  74.81%   AVERAGE NO. IN QUEUE AT SERVER 1       .00
UTIL. OF SERVER 2  74.46%   AVERAGE NO. IN QUEUE AT SERVER 2      0.95
NO. TO NOT USE FIRST SERVER 1914   NO. TO NOT USE SECOND SERVER 1264
```

The number of parts that could not use either server is: 3178

8. The number of parts that are unable to use either server for the data in the four examples (4, 5, 6, and 7) are:

Example	Number to Leave
4	4,649
5	3,749
6	3,292
7	2,386

In all cases, the number to leave without using either server is less than when Poisson distributions were assumed. This is because the Erlang distribution tends to cut off the long tail found in the Poisson distribution. Thus, extreme values are not found as much with the Erlang distribution as with the Poisson distribution.

9. The program to do the simulation is easily written. It is:

```
       SIMULATE
       RMULT     123456
ARRIVE GENERATE  RVEXPO(1,5)
       TEST L    Q(WAIT),1,AWAY
       QUEUE     WAIT
       SEIZE     WASH
       DEPART    WAIT
       ADVANCE   RVEXPO(1,4)
       RELEASE   WASH
       TERMINATE
AWAY   TERMINATE
       GENERATE  480
       TERMINATE 1
       START     1
       PUTPIC    LINES=4,N(ARRIVE),N(AWAY),1.-FLT(N(AWAY))/N(ARRIVE),_
                 FR(WASH)/10.
NUMBER OF CARS TO ARRIVE:    ****
NUMBER TO GO AWAY            ****
FRACTION SERVED                  .***
UTIL. OF CAR WASH             ***.**%
       END
```

The results of the simulation for 480 time units are:

```
NUMBER OF CARS TO ARRIVE:   95
NUMBER TO GO AWAY           29
FRACTION SERVED              .695
UTIL. OF CAR WASH           64.95%
```

Running the program for 10*480 time units yields:

```
NUMBER OF CARS TO ARRIVE:     949
NUMBER TO GO AWAY             261
FRACTION SERVED                  .725
UTIL. OF CAR WASH              60.61%
```

Running the program for 100*480 time units yields:

```
NUMBER OF CARS TO ARRIVE:     9575
NUMBER TO GO AWAY             2548
FRACTION SERVED                   .734
UTIL. OF CAR WASH               59.30%
```

Finally, running the program for 1000*480 time units yields:

```
NUMBER OF CARS TO ARRIVE:    95580
NUMBER TO GO AWAY            25191
FRACTION SERVED                   .736
UTIL. OF CAR WASH               59.15%
```

It would appear that running the simulation for 100*480 time units gives a satisfactory result.

10.

```
      WORK  FUNCTION   AC1,D4
200,12/400,15/700,17/800,20

      WORK  FUNCTION    AC1,C5
0,12/200,15/400,17/700,20/800,21
```

If the simulation runs for more than 800 time units, it is possible to have AC1@800 as the first operand in the FUNCTION statement.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 18

Parameters, the LOOP Block, and the EQU Statement

PARAMETERS ATTACHED TO EACH TRANSACTION

As each transaction travels from block to block, it carries with it several things. For example, we already know that transactions can have different levels of priority. In addition, there are other ways to make each transaction different.

Each transaction possesses a set of abstract things known as *parameters*. These are carried with the transaction as it moves through the simulation and can be modified during the program. The values of these are not normally a part of the output report but can be used during the program by the programmer. Transactions can be viewed conceptually as "stick people," and it may be helpful to think of parameters as "pockets on the pants" of the stick people. Just as you can place items in pants' pockets, you can put numbers (or, less commonly, names) inside each of the pockets to differentiate between the transactions. Each parameter has a number from 1 to 255. You can give parameter number 12 the value 4, parameter number 7 the value −47, etc. The transaction will carry these values with it as it moves through the blocks. How you do this will be explained below.

Each transaction can have 4 different kinds of parameters. There can be up to 255 of each, although it is rare that one would use more than a few in a typical program. Parameters are designated by two letters to show the parameter type. The 4 types of parameters in GPSS/H (and their designations in GPSS/H code) are as follows:

1. *Halfword parameters* (PH) may be assigned integer values in the range −32,768 to +32,767.
2. *Fullword parameters* (PF) may be assigned integer values in the range −2,147,483,648 (= -2^{31}) to +2,147,483,647 (= $+2^{31} - 1$).
3. *Byte-word parameters* (PB) may be assigned integer values in the range −128 (= -2^{7}) to +127 (= $+2^{7} - 1$).

4. *Floating-point* (decimal) *parameters* (PL) may be assigned floating-point values whose maximum (or minimum) size is machine dependent, but can be as large as 10^{308} (or as small as 10^{-308}) as far as the GPSS/H processor is concerned.

Parameters can be thought of as a collection of SNAs or values that the transaction owns. *The form of a parameter when used as an SNA is the two type-indicating letters followed by a number to indicate which parameter of that type is to be evaluated* (recall that GPSS/H evaluates any SNA in a line of code before doing anything else with the line of code). *The form of a parameter when used to refer to a specific, previously defined value is a number followed by the parameter type.* The values of these parameters are normally numbers, although one also can give them names. For the purposes of this book, the parameters used will have only number values.

As an example of the use of parameters, in the GENERATE block, every transaction initially receives 12 halfword parameters by default. Thus, although we didn't know it, all of our transactions so far had 12 of these halfword parameters. The number of halfword parameters can be increased or decreased, and other types of parameters can be attached to a transaction, in the GENERATE block in operands *F* through *I*. In the GENERATE block, halfword parameters are defined by *n*PH, fullword parameters by *n*PF, byte-word parameters by *n*PB, and floating-point parameters by *n*PL, where the *n* indicates the number of available parameters of that type. *It makes no difference where in operands F through I you indicate the number of each type of parameter, so the following two lines of GPSS/H code are equivalent:*

```
◇        GENERATE  100,3,,,,3PH,4PF,5PB,6PL
◇        GENERATE  100,3,,,,5PB,6PL,3PH,4PF
```

Some examples of parameter definitions in GENERATE blocks follow:

```
◇(a)     GENERATE  12,2,,,,4PH
◇(b)     GENERATE  ,,,5,,12PF,20PH
◇(c)     GENERATE  ,,,12,,5PF
◇(d)     GENERATE  12,4,,,,0PH,1PF
◇(e)     GENERATE  100,3,,,,3PH,4PF,5PB,6PL
◇(f)     GENERATE  ,,,10,1,12PL
◇(g)     GENERATE  100,,,,,20PH,50PB
```

Block (a) generates transactions with 4 halfword parameters. Block (b) generates transactions with 12 fullword parameters and 20 halfword parameters. Block (c) generates transactions with 5 fullword parameters. Block (d) generates transactions with no halfword parameters and 1 fullword parameter. Block (e) generates transactions with 3 halfword parameters, 4 fullword parameters, 5 byte-word transactions, and 6 floating-point transactions. Block (f) generates transactions with 12 floating-point transactions. Block (g) generates transactions with 20 halfword transactions and 50 byte-word transactions.

It is important to remember that, once you specify any parameter types and numbers via the F–I operands of the GENERATE block, you no longer have the 12 halfword parameters by default. Thus,

```
◇        GENERATE  ,,,1,,1PL
```

generates a single transaction with only 1 floating-point parameter and no halfword parameters. If, later in the program, you tried to use the following ASSIGN block (this kind of block is described next), an error would result:

```
◇        ASSIGN    1,1,PH
```

Owing to storage constraints, it is best to use halfword or byte-word parameters unless the numbers used as parameter values are beyond the ranges of these parameter types. In addition, although it may not seem obvious, it is preferable to specify that all the transactions in a program have the same number and type of parameters. For example, you may have the main GENERATE block as

```
◇        GENERATE  ,,,5,,20PH
```

Later in the program, the timer transaction enters via

```
◇        GENERATE  480*100            TIMER TRANSACTION ARRIVES
```

It is preferable to have this as

```
◇        GENERATE  480*100,,,,,20PH  TIMER TRANSACTION ARRIVES
```

even though it is going to be terminated immediately.

Similar to other SNAs such as Q*n*, FR*n*, and SC*n*, parameters that are SNAs (i.e., PH*n*, PF*n*, PB*n*, and PL*n*) can be used as operands in other blocks besides the ASSIGN block (described in the next section). For example, consider the following lines of code:

```
◇(a)     ADVANCE   PF4
◇(b)     TEST E    PH1,PH4,DOWN1
◇(c)     QUEUE     PH1
◇(d)     ENTER     TUGS,PB2
```

In (a), the transaction will be placed on the FEC for a time given by the transaction's fullword parameter 4. In (b), a test is made to see if the first and fourth halfword parameters are equal (TEST E is covered in Chapter 16). If so, the transaction moves sequentially to the next block. It they are not equal, the transaction is routed to the block with the label DOWN1. In (c), the transaction joins the queue given by its first halfword parameter. In (d), the transaction will enter the storage TUGS and use a storage as specified by its second byte-word parameter.

In addition, consider the following:

```
◇TIMES   FUNCTION  PH1,D4
1,100/2,125/3,150/4,175
```

Now, when a transaction enters the block

```
◇        ADVANCE   FN(TIMES)
```

it will be placed on the FEC for a duration of 100, 125, 150, or 175 time units depending on the value of its first halfword parameter.

THE ASSIGN BLOCK

Initially–i.e., when a transaction is created in a GENERATE block–the values of all the transaction's parameters are zero. The value(s) of a transaction's parameter(s) can be modified via the immediately following ASSIGN block(s):

```
◇        ASSIGN    A,B,C
```

where operand *A* is the parameter number, which must be between 1 and 255 or may be any SNA whose value falls in that range; operand *B* is the value (either given by a number or an SNA) to be assigned to the parameter; and operand *C* is the parameter type (PH, PF, PB, or PL) (the parameter type can be omitted for certain cases, but it is not considered good programming to do so in general). For example, if the transaction is given only the 12 halfword parameters by default, as has been the case in preceding chapters in this book, it would be acceptable to omit the parameter type. Thus, when a transaction leaves the block

```
◇        ASSIGN    1,5,PH
```

the halfword parameter 1 will have the value of 5. The following code

```
◇        GENERATE  ,,,3
◇        ASSIGN    2,100,PH
```

generates 3 transactions, and the value of their second halfword parameter is set to 100. The result of

```
◇VALUE   FUNCTION  RN1,D3
.2,1/.5,2/1,3
◇        GENERATE  ,,,1
◇        ASSIGN    2,FN(VALUE),PH
```

is that the transaction's second halfword parameter will have the value of 1 for 20% of the time, the value of 2 for 30% of the time, and the value of 3 for the rest of the time. An example might be a mine in which 20% of the trucks are type 1, 30% are type 2, and the remainder are type 3. Later in the program, these parameters can be used to differentiate the trucks. For example, suppose that the loading rates for the 3 truck types are given by the functions TYPE1, TYPE2, and TYPE3. A simulation of the loading of the fleet of trucks might be given by

```
◇LOADIT  FUNCTION  PH2,M3
1,FN(TYPE1)/2,FN(TYPE2)/3,FN(TYPE3)
◇LOAD1   FUNCTION  RN1,C20
◇        -----
◇LOAD2   FUNCTION  RN1,C24
◇        -----
```

```
◇LOAD3    □FUNCTION  □RN2,C25
◇         □-----
◇         □-----
◇         □ADVANCE   □FN(LOADIT)
```

Since any of the operands of the ASSIGN block can be an SNA, it is possible to have the following:

```
◇         □ASSIGN    □PH1,3,PH
```

In this block, which parameter is going to be assigned the value 3 depends on what value is already in the transaction's halfword parameter 1 (PH1). If PH1 contains a value of 4, then halfword parameter 4 (4PH) will be assigned the value 3. If PH1 contains a value of 8, then halfword parameter 8 (8PH) will be assigned the value 3.

The following code containing the SNA N(TIMES) (which tracks the total number of transactions to have entered the ASSIGN block) in the *B* operand

```
◇TIMES    □GENERATE  □,,,5
◇         □ASSIGN    □1,N(TIMES),PH
```

creates 5 transactions. The first will have a 1 assigned to its halfword parameter 1, the second a 2 assigned to its halfword parameter 1, the third a 3, etc. This code pair is a method of generating a series of transactions with a single GENERATE block and giving the transactions sequential numbers in halfword parameter 1. The block

```
◇         □ASSIGN    □Q(WAIT)+1,1.23,PL
```

will assign the value of 1.23 to the floating-point parameter whose number is given by the value of Q(WAIT)+1.

Giving Parameters Names

As mentioned, it is also possible to designate the parameter by a name instead of a number, as in the following:

```
◇         □ASSIGN    □TOM,10,PH
```

Later, when reference to the halfword parameter named TOM is made, it is done as follows:

```
◇         □ADVANCE   □PH(TOM)
```

The transaction will be put on the FEC for a time of 10, since the value assigned to the halfword parameter TOM is 10. The preference here, however, is to use parameters given by numbers rather than by name.

The ASSIGN Block in Increment or Decrement Mode

You can add to (or subtract from) the value of a parameter by putting a plus (or minus) before the first comma in the operands. The following ASSIGN block will take the value in halfword parameter 4 and add 5 to it:

```
◇         □ASSIGN    □4+,5,PH
```

The following ASSIGN block will subtract 6 from the value in halfword parameter 3:

```
◇         □ASSIGN    □3-,6,PH
```

The following ASSIGN block will add the length of the queue WAIT to the transaction's first halfword parameter. If the queue length was 4 and the value of PH1 was 12, its new value would be 16.

```
◇         □ASSIGN    □1+,Q(WAIT),PH
```

The following ASSIGN block will add the average waiting time of the nonzero entry transactions for the queue WAIT1 to the transaction's seventh floating-point parameter:

```
◇         □ASSIGN    □7+,QX(WAIT1),PL
```

Example 18.1

In Chapter 12, we had an example (number 12.4) of a hardware store with shoppers selecting items from each of 4 possible aisles. Each shopper who went through the store took 45 ± 12 seconds to check out. Shoppers who went directly to the checkout counter took 20 ± 8 seconds. In actual practice, the time to check out is a function of how many items each person has in the shopping cart. Suppose the number of items selected by each person who goes down each aisle is given by the following:

Aisle	No. of Items Selected
1	3 ± 2
2	4 ± 3
3	3 ± 1
4	5 ± 4

At the checkout counter, all people select additional items as follows:

No. of Additional Items	Relative Probability
0	0.30
1	0.25
2	0.45

Checkout time is 3.5 seconds per item. Shoppers arrive in a Poisson stream with a mean of 82.5 seconds. Simulate for 100 shifts of 8 hours each. Determine the number of carts to have in the store so that a shopper will always be guaranteed of having a cart. How busy is the checkout person? Modify the program written previously to include these changes.

Solution

```
◇         □SIMULATE
◇AISLE1   □FUNCTION  □RN1,C2              □SHOPPING IN AISLE 1
0,1/1,6
```

```
◇AISLE2  ▯FUNCTION  ▯RN1,C2            ▯SHOPPING IN AISLE 2
0,1/1,8
◇AISLE3  ▯FUNCTION  ▯RN1,C2            ▯SHOPPING IN AISLE 3
0,2/1,5
◇AISLE4  ▯FUNCTION  ▯RN1,C2            ▯SHOPPING IN AISLE 4
0,1/1,10
◇WAIT    ▯FUNCTION  ▯RN1,D3
.3,0/.55,1/1,2
◇        ▯STORAGE   ▯S(CARTS),1000      ▯PROVIDE 1000 CARTS
◇        ▯GENERATE  ▯RVEXPO(1,82.5)     ▯CUSTOMERS ARRIVE
◇        ▯TRANSFER  ▯.12,,COUNTER       ▯12% GO TO COUNTER
◇        ▯ENTER     ▯CARTS              ▯REST TAKE A CART
◇        ▯TRANSFER  ▯.2,,AISLE2         ▯80% GO TO AISLE 1
◇        ▯ASSIGN    ▯1,FN(AISLE1),PH    ▯SELECT ITEMS IN AISLE 1
◇        ▯ADVANCE   ▯125,70             ▯SHOP IN AISLE 1
◇AISLE2  ▯TRANSFER  ▯.25,,AISLE3        ▯75% GO TO AISLE 2
◇        ▯ASSIGN    ▯1+,FN(AISLE2),PH   ▯SELECT ITEMS IN AISLE 2
◇        ▯ADVANCE   ▯140,40             ▯SHOP IN AISLE 2
◇AISLE3  ▯TRANSFER  ▯.15,,AISLE4        ▯85% GO TO AISLE 3
◇        ▯ASSIGN    ▯1+,FN(AISLE3),PH   ▯SELECT ITEMS IN AISLE 3
◇        ▯ADVANCE   ▯150,65             ▯SHOP IN AISLE 3
◇AISLE4  ▯TRANSFER  ▯.10,,CHECK         ▯90% GO TO AISLE 3
◇        ▯ASSIGN    ▯1+,FN(AISLE4),PH   ▯SELECT ITEMS IN AISLE 4
◇        ▯ADVANCE   ▯175,70             ▯SHOP IN AISLE 3
◇CHECK   ▯ASSIGN    ▯1+,FN(WAIT),PH     ▯SELECT ITEMS AT COUNTER
◇        ▯QUEUE     ▯LINE               ▯STAND IN LINE
◇        ▯SEIZE     ▯CHECKER            ▯READY TO CHECK OUT
◇        ▯DEPART    ▯LINE               ▯LEAVE THE QUEUE
◇        ▯ADVANCE   ▯PH1*3.5            ▯CHECK OUT
◇        ▯RELEASE   ▯CHECKER            ▯FREE THE CHECKER
◇        ▯LEAVE     ▯CARTS              ▯GET RID OF CART
◇        ▯TERMINATE                     ▯LEAVE THE STORE
◇COUNTER ▯ASSIGN    ▯1,FN(WAIT),PH      ▯SELECT ITEMS AT COUNTER
◇        ▯QUEUE     ▯LINE               ▯STAND IN LINE
◇        ▯SEIZE     ▯CHECKER            ▯READY TO CHECK OUT
◇        ▯DEPART    ▯LINE               ▯LEAVE THE QUEUE
◇        ▯ADVANCE   ▯PH1*3.5            ▯CHECK OUT
◇        ▯RELEASE   ▯CHECKER            ▯FREE THE CHECKER
◇        ▯TERMINATE                     ▯LEAVE THE STORE
◇        ▯GENERATE  ▯3600*8*100         ▯SIMULATE FOR 20 DAYS
◇        ▯TERMINATE ▯1                  ▯TIMER TRANSACTION ARRIVES
◇        ▯START     ▯1
◇        ▯PUTPIC    ▯LINES=7,N(COMEIN),N(TAKECART),_
                     N(COUNTER),FR(CHECKER)/10.,QM(LINE)_
                     QA(LINE)SM(CARTS)
 NUMBER TO COME TO SHOP               *****
 NUMBER TO USE AISLES                 *****
 NUMBER TO GO DIRECTLY TO COUNTER     *****
 UTIL. OF CHECKER AT CHECKOUT          ***.**%
 MAXIMUM QUEUE AT CHECKOUT             ****
 AVERAGE QUEUE AT CHECKOUT             ****.**
 MAXIMUM CARTS IN USE AT ONE TIME      ****
◇        ▯END
```

The output from the simulation is

```
NUMBER TO COME TO SHOP              34973
NUMBER TO USE AISLES                30841
NUMBER TO GO DIRECTLY TO COUNTER     4132
UTIL. OF CHECKER AT CHECKOUT           49.94%
MAXIMUM QUEUE AT CHECKOUT               8
AVERAGE QUEUE AT CHECKOUT               0.31
MAXIMUM CARTS IN USE AT ONE TIME       18
```

The results of the simulation show that 34,973 people entered the hardware store. Of these, 30,841 went shopping down the various aisles. The remaining 4,132 went directly to the checkout counter. The checkout person was busy 49.94% of the time, and the average queue at the checkout was 0.31. Notice that, even though the average queue length was short, there was at least once a queue length of 8. The maximum number of carts in use in the shop at one time was 18. It would appear that having 20 carts available would be sufficient to guarantee that no person would come and find them all taken.

General Form of the ASSIGN Block

There is a more general form of the ASSIGN block that is not used much anymore since the exponential distribution now is built in and is referenced by a single function call. However, the more general form is presented here for the sake of being complete:

```
◇          ASSIGN    A,B,C,D
```

Operands *A* and *B* have their usual meaning. *C*, however, is the name or number of a function. *D* is the type of parameter that *A* is. If *C* is omitted, then *D* takes its place, and we have the ASSIGN block presented previously. If one used all 4 operands, the effect is as follows:

1. The function as specified by the *C* operand is evaluated. If the value returned is a decimal, the value is truncated if the parameter is an integer type.
2. This value is then *multiplied* by the number in the *B* operand.
3. The result of the multiplication in step 2 is placed in the transaction's parameter as specified by the *A* operand.

For example,

```
◇          ASSIGN    3,6,5,PH
```

The function defined with the label 5 is evaluated. Suppose the result is 2. This is multiplied by 6 and the new result–namely, 12–is placed in the transaction's third halfword parameter. In the following block,

```
◇          ASSIGN    1,3,FIRST,PL
```

the function FIRST is evaluated. Suppose the number returned is 2.9543. This is multiplied by 3. The result, 8.8629, is placed in the transaction's first floating-point parameter. If the ASSIGN block had been

```
◇        ASSIGN    1,3,FIRST,PH
```

the result returned would have been 8 and not 8.8629 because values are truncated to integers.

Notice the form in the two preceding lines of code. If you had written

```
◇        ASSIGN    1,3,FN(FIRST),PH
```

a run-time error would probably result unless there happened to be a function with the number (i.e., as a label) given by the value returned by referencing the function FIRST because GPSS/H evaluates the function FIRST before doing the ASSIGN specification. If, for example, FN(FIRST) returns the value 7, then the function with the label 7 is evaluated. If no such function exists, a run-time error occurs.

In Chapter 14, it was mentioned that the block

```
◇        ADVANCE   FN(ONE),FN(TWO)
```

would place the transaction on the FEC for a time of FN(ONE) × FN(TWO), *not* for a time given by FN(ONE) ± FN(TWO). If this latter result is needed, the way to accomplish it is with the following three lines of code:

```
◇        ASSIGN    1,FN(ONE),PH
◇        ASSIGN    2,FN(TWO),PH
◇        ADVANCE   PH1,PH2
```

THE LOOP BLOCK

GPSS/H does not have general DO loops as blocks. These are common in other languages such as Fortran and Pascal. In Chapter 23, however, we shall see that it is possible to have DO loops that are used in control statements. These DO loops are very useful in running programs multiple times with selected variables changed.

GPSS/H does have a block that acts similarly to the DO loop, but it is restricted. This block is the LOOP block, and it acts in connection with a transaction's specified parameter. The general form of it is

```
◇        LOOP      A,B
```

where operand *A* is a parameter and operand *B* is the label of the block *at the start of the loop.* An example of such a block is

```
◇        LOOP      1PH,BACK1
```

The way the LOOP block works is as follows: The transaction's parameter, as specified in the *A* operand, is used to control the looping. (Note that the first iteration of the loop has already been done by the time that the transaction reaches the LOOP block.) When the transaction enters the LOOP block, the value given by the *A* operand is decremented by 1, and the result is compared with 0. If the value is 0, the transaction is routed to the next sequential block. If the value is greater than 0, the transaction is routed to the block whose

label is given in the *B* operand of the LOOP block. The destination block is always *before* the LOOP block. Some examples of LOOP blocks are

```
◇(a)    LOOP    3PH,BACK1
◇(b)    LOOP    PH1,UPTOP
◇(c)    LOOP    5PF,OVER
```

In (a), suppose that the value of the transaction's third halfword parameter is 6. Then the looping is done for values of 5, 4, 3, 2, and 1. In (b), the transaction's first halfword parameter is used. Suppose its value is 4, and the value of halfword parameter 4 is 5. Looping is done for 4, 3, 2, and 1. In (c), the looping depends on the value of the transaction's fullword parameter 5.

Looping is done *only* by a decrement of 1 and cannot be done by increments. As restrictive as this block may seem, there are many uses for it.

THE EQU STATEMENT (COMPILER DIRECTIVE)

You can use parameters as operands of other blocks such as the QUEUE and SEIZE blocks. In fact, this is often done. Thus, one could have blocks such as

```
◇       QUEUE   PH2
◇       QUEUE   PH1+PH4
◇       SEIZE   PH4
```

Use of such code can greatly compress the number of lines of GPSS/H. For example, suppose that there are 3 types of ships entering a harbor. Type 1 requires 1 tugboat to berth it, type 2 requires 3 tugboats, and type 3 requires 4 tugboats. Rather than have three nearly identical segments of code, you could set each ship type to have a different number in one of its parameters, say halfword parameter 5. Thus, type 1 ships might have a 1 in halfword parameter 5, type 2 ships have a 3, and type 3 ships have a 4. Then you could use the block

```
◇       ENTER   TUGS,PH5
```

where TUGS has been defined as the STORAGE to represent the number of tugboats available. In addition, you could have the ships enter separate queues according to ship type by using the block

```
◇       QUEUE   PH5
```

If you wanted the ships to be in a global queue, it is tempting to write

```
◇       QUEUE   WAIT
◇       QUEUE   PH5
```

Unfortunately, there is a problem. If it is assumed that there are no other queues in the program or that the queue WAIT is the first queue encountered by the compiler, the queue WAIT is assigned to the first queue during compiling. Now, when a type 1 ship enters queue 1, this is not only QUEUE 1 but also QUEUE WAIT. The statistics will be incorrect. One way around this is to have the queue WAIT renamed as, say,

```
◇       QUEUE   10
```

This approach will work, but it is better to use mnemonics that represent names that are more meaningful than just numbers. The EQU compiler directive allows programmers to use names for operands but also to assign them specific numbers. The general form of this statement is

```
◇LABEL   ▯EQU      ▯A,B
```

The label is the mnemonic that you want to use. Operand *A* is the integer number that you want to assign to the entity. Operand *B* is the family name of the entity. These are

Entity	Family Name
facility	F
function	Z
parameter	PH, PB, PF, or PL
queue	Q
random number	RN
storage	S

Some examples of the EQU statement are discussed next.

```
◇MACH1   ▯EQU      ▯10,F  ▯(a)
◇HALT    ▯EQU      ▯4,Q   ▯(b)
```

In (a), the facility MACH1 is specified as being the tenth facility. In (b), the queue HALT is specified as being the fourth queue.

```
◇WAIT    ▯EQU      ▯7,Q   ▯(c)
```

In (c), the queue WAIT is specified as being the seventh queue of all of the possible queues GPSS/H has. Example 18.2 uses a similar statement, but there is a caveat. In a program, if you had

```
◇        ▯QUEUE    ▯WAIT
◇        ▯QUEUE    ▯PH1
```

and PH1 was between 1 and 6, there would be no problem in having the queue WAIT specified as queue 7, but if PH1 could be 7 or more, then a run-time error could arise.

Finally, in the following code, both queue HALT and facility HALT are designated as the tenth queue and tenth facility, respectively.

```
◇HALT    ▯EQU      ▯10,Q,F
◇        ▯-----
◇        ▯-----
◇        ▯QUEUE    ▯HALT
◇        ▯SEIZE    ▯HALT
```

Example 18.2

Three types of ships use a harbor: type 1, type 2, and type 3. Ships of type 1 and type 2 enter the harbor and are guided into a dock by either 1 or 2 tugboats. Type 1 ships need one tugboat and type 2 ships require two tugboats.

Type 3 ships require three tugboats. When the ships leave the docks, type 1 and type 2 ships sail to another port and eventually return in a closed circuit or cycle. Type 3 ships enter the harbor every 14 ± 7 hours and then leave for good. The number of ships of type 1 and of type 2 and the unloading and reloading times for all ships in the harbor are as follows:

Ship Type	Number	Time in Dock (hours)	Time to Cycle (hours)
1	12	24 ± 6.6	180 ± 25
2	14	28 ± 8.6	206 ± 27
3	—	27 ± 7.7	—

When a ship enters the harbor, it must wait until a berth is free. There are 3 berths available. There are 3 tugboats available. When a berth is free, and there are ships waiting in a queue, the captain of the first ship in the queue checks to see if enough tugboats are available. If so, the tugboat takes 1 hour to dock the ship. After unloading and reloading, the captain of the ship again checks to see if enough tugboats are available. If so, it takes 0.15 hours for the tugboat to move the ship far enough away to free the berth. It then takes 1 ± 0.2 hours to complete unberthing and free the tug(s).

Write a GPSS/H model to simulate the harbor facility. When the ship transactions are put into the system, have them spaced out by 24 hours each, and have the ships of type 1 and type 2 enter the system via their respective ADVANCE blocks so that they are initially at the beginning of their circuit sailing away from the harbor. Form separate queues for each type of ship as well as a global queue. The program is to have all the ships in the main segment rather than to have 3 separate segments. Simulate for 2 years of 365 days, 24 hours per day, of operation.

Solution

```
◇        □SIMULATE
◇        □STORAGE   □S(DOCK),3/S(TUGS),3 □PROVIDE BERTHS, DOCKS
◇WHICH   □FUNCTION  □PH1,D3
1,FIRST/2,SECOND/3,THIRD
◇WAIT    □EQU       □10,Q                □DEFINE WAIT AS QUEUE NO. 10
◇SHIPA   □GENERATE  □,,,12,,1PH,2PL      □PROVIDE TYPE 1 SHIPS
◇        □ASSIGN    □1,1,PH              □NUMBER THEM 1
◇        □ASSIGN    □1,24,PL             □MEAN UNLOAD & LOAD TIME
◇        □ASSIGN    □2,6.6,PL            □SPREAD
◇        □ADVANCE   □N(SHIPA)*24         □SPACE OUT THE SHIPS
◇        □TRANSFER  □,FIRST              □SEND TO SEA
◇SHIPB   □GENERATE  □,,,14,,1PH,2PL      □PROVIDE TYPE 2 SHIPS
◇        □ASSIGN    □1,2,PH              □NUMBER THEM 2
◇        □ASSIGN    □1,28,PL             □MEAN UNLOAD & LOAD TIME
◇        □ASSIGN    □1,8.6,PL            □SPREAD
◇        □ADVANCE   □N(SHIPB)*24         □SPACE OUT THE SHIPS
◇        □TRANSFER  □,SECOND             □SEND TO SEA
◇        □GENERATE  □14,7,,,,1PH,2PL     □TYPE 3 SHIPS ARRIVE
◇        □ASSIGN    □1,3,PH              □NUMBER THEM 3
◇        □ASSIGN    □1,27,PL             □MEAN UNLOAD & LOAD TIME
◇        □ASSIGN    □1,7.7,PL            □SPREAD
```

```
◇HARBOR  ⎕QUEUE     ⎕WAIT              ⎕JOIN GLOBAL QUEUE
◇        ⎕QUEUE     ⎕PH1               ⎕JOIN INDIVIDUAL QUEUES
◇        ⎕ENTER     ⎕DOCK              ⎕IS A DOCK FREE?
◇        ⎕ENTER     ⎕TUGS,PH1          ⎕ARE TUGBOATS FREE?
◇        ⎕DEPART    ⎕WAIT              ⎕LEAVE THE GLOBAL QUEUE
◇        ⎕DEPART    ⎕PH1               ⎕LEAVE INDIVIDUAL QUEUE
◇        ⎕ADVANCE   ⎕1                 ⎕TUGBOATS BERTHS SHIP
◇        ⎕LEAVE     ⎕TUGS,PH1          ⎕FREE THE TUGBOATS
◇        ⎕ADVANCE   ⎕PL1,PL2           ⎕UNLOAD & LOAD A SHIP
◇        ⎕ENTER     ⎕TUGS,PH1          ⎕ENGAGE TUGBOATS
◇        ⎕ADVANCE   ⎕.15               ⎕LEAVE DOCK
◇        ⎕LEAVE     ⎕DOCK              ⎕FREE DOCK
◇        ⎕ADVANCE   ⎕1,.2              ⎕FINISH UNBERTHING
◇        ⎕LEAVE     ⎕TUGS,PH1          ⎕FREE TUGBOATS
◇        ⎕TRANSFER  ⎕,FN(WHICH)        ⎕TRANSFER SHIPS
◇FIRST   ⎕ADVANCE   ⎕180,25            ⎕TYPE 1 SHIPS AT SEA
◇        ⎕TRANSFER  ⎕,HARBOR           ⎕BACK TO HARBOR
◇SECOND  ⎕ADVANCE   ⎕206,27            ⎕TYPE 2 SHIPS AT SEA
◇        ⎕TRANSFER  ⎕,HARBOR           ⎕BACK TO HARBOR
◇THIRD   ⎕TERMINATE                    ⎕TYPE 3 SHIPS LEAVE
◇        ⎕GENERATE  ⎕24*365*2          ⎕SIMULATE FOR 2 YEARS
◇        ⎕TERMINATE ⎕1
◇        ⎕START     ⎕1,NP
◇        ⎕PUTPIC    ⎕LINES=10,FILE=SYSPRINT,(S(DOCK)+R(DOCK),_
 S(TUGS)+R(TUGS),QA(WAIT),QA1,QA2,QA3,SR(DOCK)/10,SR(TUGS)/10)
0        |====================================================|
         |  NUMBER OF DOCKS WAS                         ***     |
         |  NUMBER OF TUGBOATS WAS                      ***     |
         |  AVERAGE NUMBER OF SHIPS IN GLOBAL QUEUE      **.**  |
         |  AVERAGE NO. TYPE 1 SHIPS IN QUEUE            **.**  |
         |  AVERAGE NO. TYPE 2 SHIPS IN QUEUE            **.**  |
         |  AVERAGE NO. TYPE 3 SHIPS IN QUEUE            **.**  |
         |  UTILIZATION OF DOCK                         **.**%  |
         |  UTILIZATION OF TUGBOATS                     **.**%  |
         |====================================================|
◇        ⎕CLEAR
◇        ⎕STORAGE   ⎕S(TUGS),4
◇        ⎕START     ⎕1,NP
◇        ⎕PUTPIC    ⎕LINES=10,FILE=SYSPRINT,(S(DOCK)+R(DOCK),_
 S(TUGS)+R(TUGS),QA(WAIT),QA1,QA2,QA3,SR(DOCK)/10,SR(TUGS)/10)
0        |====================================================|
         |  NUMBER OF DOCKS WAS                         ***     |
         |  NUMBER OF TUGBOATS WAS                      ***     |
         |  AVERAGE NUMBER OF SHIPS IN GLOBAL QUEUE      **.**  |
         |  AVERAGE NO. TYPE 1 SHIPS IN QUEUE            **.**  |
         |  AVERAGE NO. TYPE 2 SHIPS IN QUEUE            **.**  |
         |  AVERAGE NO. TYPE 3 SHIPS IN QUEUE            **.**  |
         |  UTILIZATION OF DOCK                         **.**%  |
         |  UTILIZATION OF TUGBOATS                     **.**%  |
         |====================================================|
◇        ⎕CLEAR
◇        ⎕STORAGE   ⎕S(TUGS),3/S(DOCK),4
◇        ⎕START     ⎕1,NP
◇        ⎕PUTPIC    ⎕LINES=12,(S(DOCK)+R(DOCK),_
 S(TUGS)+R(TUGS),QA(WAIT),QA1,QA2,QA3,SR(DOCK)/10,SR(TUGS)/10)
```

```
|=====================================================|
|  NUMBER OF DOCKS WAS                        ***     |
|  NUMBER OF TUGBOATS WAS                     ***     |
|  AVERAGE NUMBER OF SHIPS IN GLOBAL QUEUE     **.**  |
|  AVERAGE NO. TYPE 1 SHIPS IN QUEUE           **.**  |
|  AVERAGE NO. TYPE 2 SHIPS IN QUEUE           **.**  |
|  AVERAGE NO. TYPE 3 SHIPS IN QUEUE           **.**  |
|  UTILIZATION OF DOCK                        **.**%  |
|  UTILIZATION OF TUGBOATS                    **.**%  |
|=====================================================|
◇           END
```

The program should be carefully studied. Notice the use of the EQU compiler directive:

```
◇WAIT    EQU      10,Q
```

This statement sets the queue WAIT equal to the tenth queue in the program. If this directive had not been given, then when a type 1 ship transaction entered the block

```
◇        QUEUE    PH1
```

it *also* would have been entering the queue WAIT. The output from the program is as follows:

```
|=================================================|
| NUMBER OF DOCKS WAS                      3      |
| NUMBER OF TUGBOATS WAS                   3      |
| AVERAGE NUMBER OF SHIPS IN GLOBAL QUEUE  1.37   |
| AVERAGE NO. TYPE 1 SHIPS IN QUEUE        0.36   |
| AVERAGE NO. TYPE 2 SHIPS IN QUEUE        0.47   |
| AVERAGE NO. TYPE 3 SHIPS IN QUEUE        0.54   |
| UTILIZATION OF DOCK                     92.01%  |
| UTILIZATION OF TUGBOATS                 28.28%  |
|=================================================|
|=================================================|
| NUMBER OF DOCKS WAS                      3      |
| NUMBER OF TUGBOATS WAS                   4      |
| AVERAGE NUMBER OF SHIPS IN GLOBAL QUEUE  1.20   |
| AVERAGE NO. TYPE 1 SHIPS IN QUEUE        0.30   |
| AVERAGE NO. TYPE 2 SHIPS IN QUEUE        0.42   |
| AVERAGE NO. TYPE 3 SHIPS IN QUEUE        0.48   |
| UTILIZATION OF DOCK                     90.66%  |
| UTILIZATION OF TUGBOATS                 21.14%  |
|=================================================|
|=================================================|
| NUMBER OF DOCKS WAS                      4      |
| NUMBER OF TUGBOATS WAS                   3      |
| AVERAGE NUMBER OF SHIPS IN GLOBAL QUEUE  0.28   |
| AVERAGE NO. TYPE 1 SHIPS IN QUEUE        0.08   |
| AVERAGE NO. TYPE 2 SHIPS IN QUEUE        0.11   |
| AVERAGE NO. TYPE 3 SHIPS IN QUEUE        0.10   |
| UTILIZATION OF DOCK                     68.96%  |
| UTILIZATION OF TUGBOATS                 28.40%  |
|=================================================|
```

The results of the program indicate that, with 3 docks and 3 tugs, the docks are used 92.01% of the time and the tugboats are busy 28.28% of the time. The average number of ships waiting in the global queue is 1.37. When another tugboat is added, the changes are minor: the docks are still busy more than 90%, namely, 90.66% and the tugboats are busy 21.14%. The average queue of all the ships is 1.20. When another dock is added, the changes are quite dramatic: the docks are now used only 68.96% of the time and the tugboats are busy 28.40% of the time, but the average global queue is reduced to .28 ships.

EXERCISES, CHAPTER 18

1. A transaction has values 1 and 2 in halfword parameters 1 and 2, respectively.

```
◇FIRST    □FUNCTION  □PH1,D2
1,6/2,10
◇SECOND   □FUNCTION  □PH2,D2
1,2/2,3
```

State what will happen when the transaction enters each of the following blocks:

```
◇(a)      □ADVANCE   □PH1*60
◇(b)      □ADVANCE   □FN(FIRST)*FN(SECOND)
◇(c)      □QUEUE     □PH2
◇(d)      □ENTER     □TUGS,PH1
◇(e)      □LEAVE     □SHIP,PH1*PH2
◇(f)      □ADVANCE   □FN(FIRST),PH2
◇(g)      □ADVANCE   □10,FN(FIRST)
```

Also state the effect of each of the following three blocks:

```
◇(h)      □ASSIGN    □3,PH1,PH
◇         □ASSIGN    □4,PH2,PH
◇         □ADVANCE   □PH3*PH4

◇(i)      □ASSIGN    □3,PH1,PH
◇         □ASSIGN    □4,PH2,PH
◇         □ADVANCE   □PH4,PH3
```

2. Assume that the following entities hold the following values at a particular time in a program:

Entity	Value
F(MACH1)	1
Q(WAIT1)	2
Q(WAIT2)	3
FR(MACH1)	578
S(TUGS)	5
SC(TUGS)	123
R(TUGS)	2

A transaction has the value of its first 6 halfword parameters as follows. All other parameter values are 0.

Halfword Parameter Number	Value
1	3
2	-1
3	-2
4	5
5	4
6	6

State what happens if the transaction were to enter each of the following *independent* blocks:

```
(a)   ASSIGN   4,F(MACH1),PH
(b)   ASSIGN   4+,F(MACH1),PH
(c)   ASSIGN   PH4,F(MACH1),PH
(d)   ASSIGN   PH4-,F(MACH1),PH
(e)   ASSIGN   1,PH1*PH5,PH
(f)   ASSIGN   1-,PH1*PH5,PH
(g)   ASSIGN   Q(WAIT1),PH3,PH
(h)   ASSIGN   PH5,FR(MACH1)/S(TUGS),PH
(i)   ASSIGN   1,Q(WAIT1)+Q(WAIT2),PH
(j)   ASSIGN   S(TUGS),SC(TUGS),PH
(k)   ASSIGN   S(TUGS)-PH1,PH6,PH
(l)   ASSIGN   PH5+,SC(TUGS)-F(MACH1),PH
```

3. A construction job is using 1 shovel and 10 trucks–6 of type 1 and 4 of type 2. The shovel can load only 1 truck at a time. Each truck is loaded in 1.2 ± 0.5 minutes. Both types of trucks then travel to a junction in the same time, namely, a time given by the normal distribution with a mean of 5 minutes and a standard deviation of 0.75 minutes. At the junction, trucks of type 1 travel to a dump area in 2 ± 1.2 minutes. Type 2 trucks travel to a different dump area in 2.5 ± 0.8 minutes. Both types of trucks take 1 ± 0.2 minutes to dump. Type 1 trucks return to the shovel in 5.5 ± 2 minutes, and type 2 trucks return to the shovel in 3.5 ± 1.2 minutes. Simulate for 100 continuous shifts of 480 minutes each. You are to use the same ADVANCE blocks for both truck types.
4. Consider the following code:

```
        GENERATE ,,,1
        ASSIGN   1,5,PH
        ASSIGN   5,2,PH
BACK    ASSIGN   2+,PH1,PH
        LOOP     1PH,BACK
        TERMINATE 1
        START    1
```

What will be the value of the transaction's second parameter at the end of the program?
5. Suppose that the LOOP block was LOOP PH1,BACK in exercise 4. What would the value of the transaction's second parameter be at the end of the program?

SOLUTIONS, CHAPTER 18

1. a. It will be put on the FEC for 60 time units.
   b. It will be put on the FEC for 18 time units.
   c. It will enter queue number 2.
   d. It will use 1 of the storages TUGS.
   e. It will free 2 of the storages, SHIP.
   f. It will be placed on the FEC for a time of 6 ± 2.
   g. It will be placed on the FEC for a time of 10*1 time units, NOT 10 ± 1.
   h. The effect of the 3 blocks is to place the transaction on the FEC for 2 time units.
   i. The transaction will be placed on the FEC for 2 ± 1 time units.
2. a. Halfword parameter 4 has the value 1.
   b. Halfword parameter 4 has the value 6.
   c. Halfword parameter 5 has the value 1.
   d. Halfword parameter 5 has the value 4.
   e. Halfword parameter 1 has the value 12.
   f. Halfword parameter 1 has the value −9.
   g. Halfword parameter 2 has the value −2.
   h. Halfword parameter 4 has the value 115. (integer division)
   i. Halfword parameter 1 has the value 5.
   j. Halfword parameter 5 has the value 123.
   k. Halfword parameter 2 has the value 6.
   l. Halfword parameter 4 has the value 122.
3. The program to do the simulation is as follows:

```
          SIMULATE
 TODUMP   FUNCTION   PH1,M2
1,FN(ONE)/2,FN(TWO)
 ONE      FUNCTION   RN1,C2
0,.8/1,3.2
 TWO      FUNCTION   RN1,C2
0,1.7/1,3.3
 TOSHOVEL FUNCTION   PH1,M2
1,FN(THREE)/2,FN(FOUR)
 THREE    FUNCTION   RN1,C2
0,3.5/1,7.5
 FOUR     FUNCTION   RN1,C2
0,2.3/1,4.7
          GENERATE   ,,,6
          ASSIGN     1,1,PH
          TRANSFER   ,DDDD
          GENERATE   ,,,4
          ASSIGN     1,2,PH
 DDDD     QUEUE      WAIT
          SEIZE      SHOVEL
          DEPART     WAIT
```

```
        ADVANCE    1.2,.5
        RELEASE    SHOVEL
        ADVANCE    RVNORM(1,5,.75)
        ADVANCE    FN(TODUMP)
        ADVANCE    1,.2
        ADVANCE    FN(TOSHOVEL)
        TRANSFER   ,DDDD
        GENERATE   480*100
        TERMINATE  1
        START      1
        PUTPIC     LINES=3,FC(SHOVEL)/100.,FR(SHOVEL)/10
        RESULT OF SIMULATION.....

TRUCKS LOADED PER SHIFT ***.**    UTIL. OF SHOVEL  ***.**%
        END
```

The results of the simulation are:

```
        RESULT OF SIMULATION.....

TRUCKS LOADED PER SHIFT 321.36    UTIL. OF SHOVEL   80.52%
```

4. The value of the second parameter will be 15.
5. The value of the second parameter will be 15. This value is not obtained in the same manner as in Exercise 14.4 but is the result of adding 5 + 5 + 5. In Exercise 14.5 the value of 15 was obtained from adding 5 + 4 + 3 + 2 + 1.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 19

Tables in GPSS/H

It is possible in GPSS/H to make tables of *any* SNA values. A table might be the length of a queue, the value of a parameter, the percentage of time a machine is working, etc. Often, the time a transaction has been in the system is tabulated or how long it took to go from one point in the program to another point is of interest. GPSS/H makes these tables in the form of histograms. For example, in studying people entering a hardware store and using the shopping carts, the simulation might give the maximum number of carts in use as 14. The mean number of carts in use at any time might be 4.5. It would be instructive to see how often 14 carts were in use as well as the distribution of the usage of the carts during the simulation. These data can be easily acquired via GPSS/H programs.

In making up the tables, GPSS/H also computes the sample mean of the data, the sample distribution, the number of samples that fall into each of the ranges, and the percentage of values in the sample that fall into each of the ranges in the series. The computations are done automatically.

Recall that a histogram has intervals that record the number of times a variable falls within each interval. Since a person doing a simulation often is concerned with tabulating data, GPSS/H provides a very simple method of doing this. In fact, to make a histogram for any SNA takes only 2 lines of code! One of these is a block and the other a statement.

THE TABLE STATEMENT

To record data in a table requires the defining of a statement called the TABLE statement. Its general form is

```
◇LABEL    ▯TABLE      ▯A,B,C,D,E
```

where operand *A* is an SNA, operand *B* is the starting data point, operand *C* is the interval width (for all but the starting and ending intervals), operand *D* is

the number of intervals, and operand *E* gives the time span to be used (if any). GPSS/H will count the intervals you specify in the following way:

1. The first interval will be from $-\infty$ to the value specified by the *B* operand.
2. The succeeding $D - 1$ intervals (where *D* refers to the *D* operand) will each be of the width specified by the *C* operand.
3. The last or *D*th interval will be from the value obtained by using the *B, C,* and *D* operands to $+\infty$.

In general, therefore, the beginning interval and ending intervals are the regions from $-\infty$ to the first data point of the histogram as specified in the *B* operand and from the last data point of the histogram to $+\infty$. *These two "extra" intervals, which are included in the interval count indicated by the D operand, are important to keep in mind when you are using the TABLE statement.* For example,

```
◇MYTAB   ▯TABLE      ▯Q(FIRST),0,3,4
```

would have 4 intervals as follows: $-\infty$ to 0, 0 to 3, 3 to 6, and 6 to $+\infty$. Some other examples of the TABLE statement are

```
◇FIRST   ▯TABLE      ▯S(CARTS),0,1,20
◇MARK1   ▯TABLE      ▯Q(WAIT),0,1,15
◇NEXT    ▯TABLE      ▯FR(MACH1),0,50,22
```

The table FIRST will record the amount of storage used in the storage CARTS and put the values in a table with intervals from 0 to 18. The interval widths will be 1. The first interval will be from $-\infty$ to 0 (even though there will never be an entry here), the second from 0 to 1, etc., and the last interval (i.e., the twentieth) from 18 to $+\infty$.

The table MARK1 will give the distribution of the number of people who were in the QUEUE called WAIT. The table will have 15 intervals, which include the interval from $-\infty$ to 0 and the interval from 13 to $+\infty$.

The table NEXT gives the utilization of the facility called MACH1. The table will go from 0 to 50, 50 to 100, 100 to 150, etc. (not counting the end intervals). The class interval of 50 was chosen because the utilization is given in parts per thousand.

THE TABULATE BLOCK

To make an entry in a table, you use the TABULATE block:

```
◇        ▯TABULATE  ▯A
```

where the operand *A* is the label of the appropriate TABLE statement. Every time a transaction enters this block, an entry in made in the TABLE that has the label given as operand *A* in the TABULATE block. Thus, the combination of a TABLE statement and a TABULATE block such as

```
◇PEOPLE  □TABLE      □Q(WAIT),0,1,10
◇        □-----      □-----
◇        □-----      □-----
◇
◇        □TABULATE   □PEOPLE
```

would make a table (histogram) of the people in the queue named WAIT.

To illustrate what a table from a program looks like, consider the program to study people using a single facility. People arrive every 12 ± 4 minutes. The service time is 11 ± 7 minutes. If the server is busy, people wait in a queue until the server is free. The program to simulate for 50 days of 480 minutes per day and to obtain a table of times for people who had to wait for service is as follows:

```
◇        □SIMULATE
◇INQUE   □TABLE      □QX(WAIT),0,1,20
◇        □GENERATE   □12,4
◇        □QUEUE      □WAIT
◇        □SEIZE      □MACH1
◇        □TABULATE   □INQUE
◇        □DEPART     □WAIT
◇        □ADVANCE    □11,7
◇        □RELEASE    □MACH1
◇        □TERMINATE
◇        □GENERATE   □480*50
◇        □TERMINATE  □1
◇        □START      □1
◇        □END
```

The block entry count and the table from the output are as follows:

```
BLOCK CURRENT    TOTAL
1                 1997
2                 1997
3                 1997
4                 1997
5                 1997
6          1      1997
7                 1996
8                 1996
9                    1
10                   1
```

```
TABLE     INQUE
ENTRIES IN TABLE  MEAN ARGUMENT  STANDARD DEVIATION  SUM OF ARGUMENTS
      1997.0000        10.4626              1.3307        20893.7941   NON-WEIGHTED
      UPPER  OBSERVED   PERCENT  CUMULATIVE     MULTIPLE   DEVIATION    DEVIATION
      LIMIT FREQUENCY  OF TOTAL  PERCENTAGE    REMAINDER     OF MEAN    FROM MEAN
     0.        1.0000      0.05        0.05        99.95     0.           -7.8625
     1.0000    3.0000      0.15        0.20        99.80     0.0956       -7.1110
     2.0000    3.0000      0.15        0.35        99.65     0.1912       -6.3596
     3.0000    4.0000      0.20        0.55        99.45     0.2867       -5.6081
```

```
  4.0000    1.0000    0.05     0.60      99.40    0.3823    -4.8566
  5.0000    4.0000    0.20     0.80      99.20    0.4779    -4.1051
  6.0000   50.0000    2.50     3.30      96.70    0.5735    -3.3536
  7.0000   16.0000    0.80     4.11      95.89    0.6691    -2.6021
  8.0000    9.0000    0.45     4.56      95.44    0.7646    -1.8506
  9.0000    3.0000    0.15     4.71      95.29    0.8602    -1.0991
 10.0000  268.0000   13.42    18.13      81.87    0.9558    -0.3476
 11.0000  887.0000   44.42    62.54      37.46    1.0514     0.4039
 12.0000  717.0000   35.90    98.45       1.55    1.1469     1.1553
 13.0000   30.0000    1.50    99.95       0.05    1.2425     1.9068
 14.0000    1.0000    0.05   100.00       0.00    1.3381     2.6583
```

As shown in TABLE INQUE, there were 1997 entries. The output stops at the upper limit of 14.0000 as there were no entries beyond this. GPSS/H does not give output if there are no entries. Thus, the *D* operand of the TABLE statement could have been any number greater than 14, and the same output results would be produced, for this example.

There is nothing else to do to make tables of SNAs. This ability to make tables of any SNA so easily and rapidly is often hard for people to believe the first time they are introduced to it.

SNAs ASSOCIATED WITH TABLES

There are several SNAs associated with tables. These are as follows:

TB(LABEL) or TB*j*	Sample mean. For the previous table, labeled INQUE, this is 10.4626, as shown.
TC(LABEL) or TC*j*	Number of observations (= entries). This SNA's value is 1977 for the table INQUE.
TD(LABEL) TD*j*	Standard deviation. For the table INQUE, this value is 1.3307.

THE SNA M1

An SNA that is quite useful when constructing tables–first introduced in Chapter 15–is M1; it gives the time a transaction has been in the system. Whenever a transaction is created, it is marked with the time it enters the system (by using the absolute clock!). At any later time in the simulation, when the SNA M1 is referred to, the compiler takes the time of the absolute clock and subtracts the transaction's entry time from it. This procedure gives the time that the transaction has been in the system; i.e., M1. For example, a transaction entered the system at time 404.56; when the absolute clock reads 625.77, the transaction's M1 is 231.21. Consider the following lines of code:

```
◇TIMES    □TABLE      □M1,0,25,30
◇         □-----
◇         □-----
◇         □TABULATE   □TIMES
```

The table TIMES will give the tabulation of times that the transactions are in the system from the time they entered up to the time that they encounter the block TABULATE TIMES. The times will be in intervals of 25, and there will be 30 such intervals.

THE MARK BLOCK

Suppose you want a tabulation of times for the transaction to go from point A to point B in a system. It is first necessary to make a record of the absolute clock time when the transaction arrives at point A by making a record of the absolute clock value in a parameter of the transaction. Which parameter this value is to be stored in must be specified by using the MARK block. The general form of the block is

```
◇        MARK      A
```

where the *A* operand is the number of the chosen parameter followed by its type (PH, PF, PL, or PB), e.g., 4PH. Note that the parameter as used in this block is *not* an SNA. Also note that the form MARK (number of parameter)$(parameter type) will also work. Thus, the following will work but is not used in this book:

```
◇        MARK      5$PL
```

When a transaction leaves the MARK block, the effect is to put the absolute clock time in the transaction's parameter specified by the number in the *A* operand. Since the absolute clock value is a real number, if the parameter number does not refer to a floating-point parameter, the value is truncated to an integer. When this happens, a message such as the following appears:

"IN STATEMENT 9 - WARNING 393 - Clock value (floating point) will be truncated to an integer value."

Thus, if the absolute clock was at 1234.567 and you had:

```
◇        MARK      4PH
```

then halfword parameter 4 would have the value of 1234. Such truncation is normally not desired and, so, when using this block, it is recommended that floating-point parameters be used to avoid truncation error. For example, you could have

```
◇        GENERATE  ,,,1,,5PL
◇        -----     -----
◇        MARK      5PL
```

If you had omitted the PL in the MARK block, you would get an error message. Don't forget that, since you are now specifying in the GENERATE block that the transaction has 5 floating-point parameters, it does not have *any* halfword parameter.

In order to tabulate the time it takes for a transaction to go from point A to point B, the SNA that is used in the *A* operand of the TABLE statement in conjunction with the MARK block is MP*xn*, where M refers to time (as in the SNA M1), P*x* indicates the parameter type (*x* = H, B, F, or L), and *n* is the parameter number (the parameter type and number must match the usage in the associated MARK block; see Appendix B for more information). In the TABLE statement, the effect of this SNA is to subtract its value from the current absolute clock value. Referring to the same example, if the clock value

was now 2344.777, the value of MPL4 would be 1110.21. Examples of this SNA are as follows:

```
◇FIRST    □TABLE     □MPL1,0,50,20
◇SECOND   □TABLE     □MPL2,0,40,22
◇THIRD    □TABLE     □MPL3,0,30,25
          □GENERATE  □30,12,,,,12PH,3PL
           -----
          □MARK      □1PL
           -----
          □MARK      □2PL
           -----
          □MARK      □3PL
           -----
          □TABULATE  □FIRST
          □TABULATE  □SECOND
          □TABULATE  □THIRD
```

The preceding code will give 3 tables of the simulated times it takes the transactions to travel from the 3 MARK blocks to the 3 TABULATE blocks in the program.

ADDITIONAL TABLES

There are several tables described next that GPSS/H can make for you with only a minor change to the program.

IA Mode Table

Whenever a GENERATE block is used, the distribution of interarrival times is required as data. Often, the interarrival (IA) times at points *interior* to a model are required. For example, at a particular point in a model, whenever a transaction arrives, a record is made of the arrival time, say, 456.78 time units. At a later time, say, 666.99 time units, a second transaction arrives at the same point. The interarrival time is then determined as 666.99 - 456.78 or 210.21.

The interarrival time can be tabulated by the following lines of code, for example,

```
◇FIRST    □TABLE     □IA,0,20,25
◇         □-----
◇         □TABULATE  □FIRST
```

The SNA IA in the TABLE statement must be operand *A*.

RT Mode Table

Closely related to the IA mode is the *rate* (RT) of the arrivals at a point, for example, the arrivals per 10 seconds, the arrivals per minute, etc. A table showing arrival rates is constructed as follows:

```
◇SECOND   □TABLE     □RT,0,15,25,10
```

Notice that there is an *E* operand in the TABLE statement. Recall that, in a TABLE statement, the *E* operand gives the time span to be used. In the above example, the table will give arrivals per 10 time units. To make an entry in the table, the following block is used in conjunction with the TABLE statement labeled SECOND:

```
◇          TABULATE   SECOND
```

QTABLE Mode

The average residence time in a queue is often required. The average time is given in the ordinary output. The QTABLE gives the table of average times in the queue. Amazingly enough, this table requires only a single line of code:

```
◇LABEL     QTABLE     A,B,C,D
```

where operand *A* is a queue label and operands *B*, *C*, and *D* are the same as in the TABLE statement. For example, in the code

```
◇LINE1     QTABLE     WAIT,0,600,20
```

whenever a transaction leaves the queue WAIT, an entry is made in the QTABLE labeled LINE1; the entry is the time that the transaction spent in the queue.

EXERCISES, CHAPTER 19

1. At stage A, parts enter a system every 15 ± 6 seconds. They join a queue before the first machine, which can work on only 1 part at a time. The process takes 13 ± 7 seconds. The parts then move to stages B, C, and D before leaving the system. However, after stage D, 10% of the parts go back for complete reworking and must go through the system again. Determine (a) the average time for the parts to go completely through the system, (b) the average time for the parts to go from A to C, (c) the average time for the parts to go from A to D, and (d) the average time for the parts to go from B to finish.
2. Shoppers enter a small grocery store in a Poisson stream with a mean of 75 seconds. Each takes a shopping cart. Each person will purchase from 0 to 25 items, uniformly distributed. It takes 3.2 seconds per item at the checkout counter, and there is only 1 checkout clerk. The shop is open 10 hours each day. Determine a histogram of the number of carts in use for a 20-day period.
3. At a repair shop there are 3 bays to repair trucks. There are 3 types of trucks that come for repairs: type 1, type 2, and type 3. The percentage of each that arrives are 35% for type 1, 40% for type 2, and 25% for type 3. A truck arrives every 16 ± 6 minutes for repairs. There are two types of repairs: normal and long. The time to repair the trucks if the service type is normal is

Type	Time in Minutes (normal distribution)	
	Mean	Standard Deviation
1	30	6.0
2	35	5.5
3	50	8.4

Of the total trucks that come for repairs, 20% require a longer time to finish. The repair time for all three types of these trucks is the same, 80 ± 30 minutes. However, the long-repair-time trucks are given a lower priority than trucks requiring normal service. Determine how long each truck is in the repair shop.

SOLUTIONS, CHAPTER 19

1. The program to do the simulation is given below:

```
        SIMULATE
        STORAGE    S(MACHB),2/S(MACHD),3
TOTTIME TABLE      M1,10,5,30
TIMATOC TABLE      MP1PL,10,5,30
TIMATOD TABLE      MP2PL,10,5,30
TIMBTOX TABLE      MP3PL,10,5,30
        GENERATE   15,6,,,,10PL
BACK    MARK       1PL
        MARK       2PL
        QUEUE      WAITA
        SEIZE      MACHA
        DEPART     WAITA
        ADVANCE    13,7
        RELEASE    MACHA
        MARK       3PL
        QUEUE      WAITB
        ENTER      MACHB
        DEPART     WAITB
        ADVANCE    26,8
        LEAVE      MACHB
        TABULATE   TIMATOC
        QUEUE      WAITC
        SEIZE      MACHC
        DEPART     WAITC
        ADVANCE    12,8
        RELEASE    MACHC
        TABULATE   TIMATOD
        QUEUE      MACHD
        ENTER      MACHD
        DEPART     MACHD
        ADVANCE    40,12
        LEAVE      MACHD
        TRANSFER   .10,,BACK
        TABULATE   TOTTIME
        TABULATE   TIMBTOX
        TERMINATE
```

```
           GENERATE   3600*8,,,,,10PL
           TERMINATE  1
           START      1
           PUTPIC     LINES=4,TB(TIMATOC),TB(TIMATOD),TB(TIMBTOX),_
                      TB(TOTTIME)
   TIME FROM A TO C       *****.**
   TIME FROM A TO D       *****.**
   TIME FROM B TO FINISH  *****.**
   TIME IN SYSTEM         *****.**
            END
```

The result of the simulation is:

```
TIME FROM A TO C         262.96
TIME FROM A TO D          89.63
TIME FROM B TO FINISH    110.94
TIME IN SYSTEM           195.46
```

2. The program to do the simulation is as follows:

```
           SIMULATE
           STORAGE    S(CARTS),1000
 ITEMS     FUNCTION   RN1,C2
0,0/1,26
 CARTSIN   TABLE      S(CARTS),1,1,20
           GENERATE   RVEXPO(1,75)
           ENTER      CARTS
           ADVANCE    300,300
           ASSIGN     1,FN(ITEMS),PH
           QUEUE      WAIT
           SEIZE      CHECKER
           DEPART     WAIT
           ADVANCE    3.2*PH1
           RELEASE    CHECKER
           TABULATE   CARTSIN
           LEAVE      CARTS
           TERMINATE
           GENERATE   3600*10*20
           TERMINATE  1
           START      1
           PUTPIC     LINES=3,SM(CARTS),TB(CARTSIN),FR(CHECKER)/10.
  MAXIMUM NUMBER OF CARTS USED   ****
  AVERAGE NUMBER OF CARTS IN USE ****.**
  UTIL. OF CHECKER                  ***.**%
           END
```

The results of the simulation are:

```
MAXIMUM NUMBER OF CARTS USED     15
AVERAGE NUMBER OF CARTS IN USE    5.99
UTIL. OF CHECKER                 54.17%
```

3. The program to do the simulation is as follows:

```
           SIMULATE
           STORAGE      S(BAYS),3
 TYPE      FUNCTION     RN1,D3
.35,1/.75,2/1,3
```

```
 WHICH     FUNCTION     PH1,M4
1,RVNORM(1,30,6)/2,RVNORM(1,35,5.5)/3,RVNORM(1,50,8.4)/4,FN(BIGFIX)
 BIGFIX    FUNCTION     RN1,C2
0,50/1,110
 TIMEIN    TABLE        M1,10,5,30
 COMEIN    GENERATE     16,6,,,1
           ASSIGN       1,FN(TYPE),PH
           TRANSFER     .20,,MAJOR
 BACKUP    QUEUE        WAIT
           ENTER        BAYS
           DEPART       WAIT
           ADVANCE      FN(WHICH)
           LEAVE        BAYS
           TABULATE     TIMEIN
           TERMINATE
 MAJOR     PRIORITY     0
           ASSIGN       1,4,PH
           TRANSFER     ,BACKUP
           GENERATE     480*100
           TERMINATE    1
           START        1
           PUTPIC       LINES=3,N(COMEIN),SR(BAYS)/10.,TB(TIMEIN)
  NUMBER OF TRUCKS TO COME FOR SERVICE (100) SHIFTS ****
  UTIL. OF THE SERVICE BAYS                           ***.**%
  AVERAGE TIME IN THE SYSTEM FOR ALL TRUCKS           ***.**
           END
```

The results of the simulation are as follows:

```
NUMBER OF TRUCKS TO COME FOR SERVICE (100) SHIFTS 2972
UTIL. OF THE SERVICE BAYS                         93.18%
AVERAGE TIME IN THE SYSTEM FOR ALL TRUCKS         59.84
```

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 20

Savevalues

Up to this point, all of the SNAs we have used were supplied by one of the blocks or were internal to GPSS/H. Often the programmer will want to have his or her own user-supplied SNAs. Parameters have been used to store various numbers, but these are not printed out after a program is run. There are two ways to provide such user-supplied SNAs. The original way will be presented here; the other in Chapter 23. Although the methods have a lot in common, there are several significant differences.

GPSS/H provides for user-defined SNAs and calls them *savevalues* (values that are "saved"). These values are printed out in the output report, except for zero values. GPSS/H has 4 different types of savevalues:

1. *Halfword savevalues* are integers in the range −32,768 to +32,767. Their family name is XH.
2. *Fullword savevalues* are integers in the range -2^{31} (= −2,147,483,648) to $+2^{31} - 1$ (= +2,147,483,647). Their family name is XF. A fullword savevalue is referenced in this book by XF(name) if a name is used or XF*n*, where *n* is a number, if a number is used.
3. *Byte-word savevalues* are integers in the range −128 to +127. Their family name is XB.
4. *Floating-point savevalues* are decimal values whose family name is XL. Their size depends on the computer, but can be as large (or small) as 10^{-308} to 10^{308}.

To reference a savevalue, it is necessary to do so by using the family name with parentheses around the user-supplied name. Thus,

```
         ADVANCE    XF(SPEED)
         GENERATE   XL(AVG),XL(SPREAD)
         QUEUE      XH(WAIT)
         ASSIGN     1,XL(FIRST),PL
         SEIZE      XB(MACH1)
         RELEASE    XH2
```

are all examples of using savevalues as operands of various blocks.

If the second letter in the family name is omitted, the GPSS/H processor assumes that the savevalue is a fullword savevalue (such omission is not recommended, however). For example,

```
◇         ASSIGN    1,XF(TEST),PH
```

and

```
◇         ASSIGN    1,X(TEST),PH
```

are the same. In both cases, the value of the savevalue TEST will be given to the transaction's first halfword parameter. In this book, however, the exact type of the savevalue will be fully specified; thus, reference to savevalues as X(TEST) will not be done.

An old way of referencing savevalues that still works in GPSS/H is to use the dollar sign ($). Thus, XL$NUMB and XL(NUMB) are the same. The dollar sign method of referencing savevalues will not be used here as it is a holdover from the original versions of GPSS.

THE SAVEVALUE BLOCK

All savevalues are initially 0. (How to give them initial values different from 0 will be considered in the section on the INITIAL statement at the end of this chapter.) During the running of a program, it is often desired to change their values. This is done by the SAVEVALUE block. Its general form is

```
◇LABEL    SAVEVALUE A,B,C
```

where operand *A* is the name the programmer chooses for the savevalue; the name can be a "word" (must start with a letter and be no longer than 8 characters), a number, or any SNA. Operand *B* is the value to be assigned to the savevalue and can be a number or an SNA. Operand *C* is the family name of the savevalue (XH, XF, XB, or XL). If operand *C* is omitted, the savevalue is a fullword savevalue by default. The label for this block is omitted unless the block is cross-referenced.

Some examples of the SAVEVALUE block are

```
◇(a)      SAVEVALUE JIM,2,XF
◇(b)      SAVEVALUE TOMMY,-100,XH
◇(c)      SAVEVALUE JANE,32,XF
◇(d)      SAVEVALUE 1,4,XF
◇(e)      SAVEVALUE ANNA,25.63,XL
◇(f)      SAVEVALUE NEXT,25/2,XL
◇(g)      SAVEVALUE OTHER,25.0/2,XL
◇(h)      SAVEVALUE PH4,12,XH
```

In (a), the fullword savevalue JIM is set equal to 2. In (b), the halfword savevalue TOMMY is set equal to −100. In (c), the fullword savevalue JANE is set equal to 32. In (d), the fullword savevalue 1 is set equal to 4. In (e), the floating-point savevalue ANNA is set equal to 25.63. In (f), the value of NEXT

is set equal to 12.0000. It is *not* 12.5000 because *division is integer division unless specified by using at least one floating-point number in the operand list.* In (g), the division is floating-point division because of the decimal point in the 25.0 value (GPSS/H converts the 2 to a floating-point number). Here the value of OTHER is 12.5000. (There are 4 decimal places because this is the format that GPSS/H prints out for savevalues, not because this is the number of digits actually stored). In (h), it is not known what halfword savevalue will have the value of 12. This assignment will depend on the number stored in the transaction's fourth halfword parameter. If that number is 5, then savevalue 5 will have the value 12.

The use of a number for the name of a variable may seem confusing since, in other languages, variable names normally must be alphanumeric and must start with a letter. For example, in Fortran, one might have statements defining variables such as JIM = 2, TOMMY = -100.0, and JANE = 32. These 3 Fortran statements have the same effect as the first 3 examples (a–c) of the SAVEVALUE block. Corresponding Fortran statements for examples (e), (f), and (g) might be ANNA = 25.63, NEXT = 25/2, and OTHER = 25.5. But there is no equivalent Fortran statement for example (d), i.e., in Fortran you cannot say 1 = 4.

The use of numbers for names of savevalues will be avoided in this book wherever possible. (However, there are times when this feature is very handy, especially when using a parameter for the first operand of the savevalue.) For example, consider the SAVEVALUE

```
◇        □SAVEVALUE □PH1,3,PH
```

Here the savevalue to be given the value of 3 will depend on the transaction's first halfword parameter. If it happened to be 4, then the savevalue 4 is given the value 3. Referencing values by means of another number may seem strange, but there are times when this feature is very useful. *This technique is possible because the equals sign (=) in computer programming does not mean "the left-side value is equal to the right-side value" but, rather, means "the old value is replaced by the new value."*

THE FIX AND FLT MODE CONVERSION

Suppose you want to have the value of savevalue FIRST be equal to Q(ONE) plus Q(TWO), the sum of which is divided by Q(THREE), and you want an exact floating-point result. It would *not* be correct to write

```
◇        □SAVEVALUE □FIRST,(Q(ONE)+Q(TWO))/Q(THREE)XL
```

Since the length of a queue (Q) is an integer, the quotient will represent integer division. In order to have floating-point division, one could write the following:

```
◇        □SAVEVALUE □JUNK1,Q(ONE)+Q(TWO),XL
◇        □SAVEVALUE □JUNK2,Q(THREE),XL
◇        □SAVEVALUE □FIRST,XL(JUNK1)/XL(JUNK2),XL
```

This code is a bit awkward. GPSS/H offers two mode-conversion operators—FIX and FLT—to convert to fixed-point (integer) or floating-point mode. They work identically to the mode converters found in other languages, such as Fortran, where the corresponding mode converters are IFIX and FLOAT. In GPSS/H, one might have

```
FLT(Q(ONE))
FIX(XL(TEST))
FLT(FC(MACHA))
```

Thus, one could have written the desired savevalue as

```
◇        □SAVEVALUE □FIRST,(Q(ONE)+Q(TWO))/FLT(Q(THREE)),XL
```

Notice that it was *not* necessary to convert all 3 fixed-point queue values.

Savevalues can be used in increment or decrement mode, just as the ASSIGN block was used. Thus,

```
◇(a)     □SAVEVALUE □LOAD+,25,XF
◇(b)     □SAVEVALUE □COST-,XF(PRICE),XF
◇(c)     □SAVEVALUE □PILE+,FN(TRUCK),XF
◇(d)     □SAVEVALUE □PH1+,PH2,XF
◇(e)     □SAVEVALUE □NEXT+,FN(LAST)+FN(FIRST),XL
◇(f)     □SAVEVALUE □TOM-,Q(ONE)/3,XL
◇(g)     □SAVEVALUE □TOM-,Q(ONE)/3.0,XL
```

In (a), the savevalue LOAD is incremented by 25. In (b), the savevalue COST is decremented by whatever the savevalue PRICE is. In (c), the savevalue PILE is incremented by reference to the function TRUCK. In (d), the savevalue specified by the transaction's first halfword parameter is incremented by the value in its second halfword parameter. In (e), the value of the savevalue is given by reference to the sum of the functions LAST and FIRST; this sum is added to the current value of the savevalue NEXT. In (f), the savevalue TOM is decremented by the length of the queue ONE divided by 3 (integer division!). In (g), the savevalue TOM is decremented by the length of the queue ONE divided by 3.0 (floating-point division). In the first 4 examples, the savevalues are fullword savevalues. In the other 3, the savevalues are floating point. In another language, such as Fortran, corresponding statements might be

```
LOAD = LOAD + 25
COST = COST - PRICE
PILE = PILE + F(TRUCK)
X(1) = X(1) + X(2)
NEXT = NEXT + F(LAST) + F(FIRST)
TOM  = TOM - QONE/3
TOM  = TOM - QONE/3.0
```

A common error in programming is to omit the family name XH, XF, XB, or XL with the savevalue in parentheses when referencing it. If the family name is omitted, the value of the savevalue is taken to be 0 no matter what it actually is. *This type of error is most insidious as it can be extremely hard to detect and no run-time error takes place.* Thus, if you had intended to write

```
◇        □TEST□E    □XF(VALUE),1,AWAY
```

but instead wrote:

```
◇        □TEST□E    □VALUE,1,AWAY
```

The test would *always be false*. There would be no run-time error.

THE INITIAL STATEMENT

As has been indicated, the values of all savevalues are set equal to 0 by the processor when a program begins. Often, however, a programmer wants savevalues to be initially set to nonzero values. This is done with the INITIAL statement. The general form of the INITIAL statement is

```
◇        □INITIAL   □A,B,C,D,...
```

where each operand (as many as necessary) contains a pair of items: Item 1 is the family name of the savevalue followed by the specific name of the savevalue in parentheses or just the number of the savevalue without parentheses; item 2 is the value to be assigned to the specific savevalue. The two parts of each operand are separated by a comma (no space), and the various operands are separated by slashes (/).

For students who have studied Fortran, the INITIAL statement in GPSS/H is analogous to the DATA statement. An example of the INITIAL statement is

```
◇        □INITIAL   □XF(FIRST),100/XH(TEST),-340/XF1,10000
```

An alternate way (a holdover from the early days of GPSS) to write the above is with dollar signs:

```
◇        □INITIAL   □XF$FIRST,100/XH$TEST, -340/XF1,10000
```

A shorthand form can be used for multiple initialization, as follows:

```
◇        □INITIAL   □XH1-XH10,3/XH(PLACE),125/XL(TOWN),1234.5432
```

This statement sets the halfword savevalues 1 through 10 equal to 3. The halfword savevalue PLACE is set equal to 125, and the floating-point savevalue TOWN is set equal to 1234.5432.

EFFECT OF RESET AND CLEAR ON SAVEVALUES

A RESET statement does *not* affect the savevalues, but a CLEAR statement *sets all savevalues to 0.* If the program is to be rerun with the original initialized (i.e., nonzero) values, there are two things that can be done: (1) Reinitialize all the values with a new INITIAL statement (or statements) or (2) use a form of the CLEAR statement called the *selective* CLEAR. The selective CLEAR is simply the CLEAR statement followed by a list of savevalues that are *not* to be set to zero. The general form of the selective CLEAR statement is

```
◇        □CLEAR     □A,B,C,D,...
```

where each operand (as many as necessary) is a savevalue that must retain its value in the succeeding simulation. Thus,

```
◇         CLEAR      XH(TOM),XF(JOHN),XH(PLACE),XF7
```

will clear all the savevalues in the program *except* for TOM, JOHN, PLACE, and 7.

Example 20.1

A common problem in inventory control is known as the "newspaper boy's problem," which concerns a newsboy who sells his papers on the street corner, as opposed to one who delivers papers from house to house. The demand for newspapers is uncertain each day but, by building up past records, the newsboy can determine a probability distribution of expected sales. The newsboy must go in the morning to the main newspaper office and purchase papers. If he does not have enough to sell (demand greater than supply), he must buy papers from a newsstand. These cost him more than if he were able to purchase them himself at the newspaper office in the morning, but he still makes a profit by doing so. If he has too many, the main office will purchase back his unsold stock for a token amount. Suppose that the following data held:

cost per paper:	$0.36
selling price:	0.55
cost per paper later in the day:	0.45
refund per unsold paper:	0.15

The expected sales are given by the following data from past sales of newspapers:

No. of Papers Sold	Relative Probability
50	0.05
55	0.08
60	0.14
65	0.20
70	0.15
75	0.13
80	0.10
85	0.08
90	0.05
95	0.02

Determine the number of papers the newsboy should obtain each day from the main office to maximize his expected profit each day. Simulate for 200 days of sales. Use supply amounts from 50 to 95 in increments of 5. Use the PUTPIC statement to print out only the number of papers supplied and the resulting expected profit each day. Use a LOOP block to run the program several times.

Solution

The program to do the simulation is as follows:

```
◇        □SIMULATE
◇        □RMULT     □123                     □RANDOM NUMBER SEED
◇        □INITIAL   □XH(SUPPLY),50           □INITIALIZE SUPPLY AMOUNT
◇SELL    □FUNCTION  □RN1,D10                 □HOW MANY TO SELL?
.05,50/.13,55/.27,60/.47,65/.62,70/.75,75/.85,80/.93,85/.98,90/1,95
◇        □GENERATE  □,,,1                    □DUMMY TRANSACTION
◇UPTOP   □ASSIGN    □1,FN(SELL),PH           □DETERMINE SALES FOR DAY
◇        □TEST□GE   □XH(SUPPLY),PH1,DOWN     □IS THIS GREATER THAN SUPPLY?
*
*  IF SUPPLY GREATER THAN DEMAND, DETERMINE PROFIT
*
◇        □SAVEVALUE □PROFIT+,19*PH1-(XH(SUPPLY)-PH1)*15,XL
◇        □ADVANCE   □1                       □ONE DAY PASSES
◇        □TRANSFER  □,UPTOP                  □BACK FOR ANOTHER DAY
*
*  BELOW IS FOR THE CASE IN WHICH THE SUPPLY IS NOT ENOUGH
*
◇DOWN    □SAVEVALUE □PROFIT+,55*PH1-36*XH(SUPPLY)_
                -45*(PH1-XH(SUPPLY)),XL
◇        □ADVANCE   □1                       □ONE DAY PASSES
◇        □TRANSFER  □,UPTOP                  □BACK FOR ANOTHER DAY
◇        □GENERATE  □200                     □SIMULATE FOR 200 DAYS
◇        □SAVEVALUE □MONEY,XL(PROFIT)/200,XL □DETERMINE AVERAGE PROFIT
◇        □TERMINATE □1                       □END OF SIMULATION
◇        □START     □1                       □BEGIN PROGRAM
◇        □END
```

From the output of the program, the following table was constructed:

No. of Papers to Stock	Expected Profit (cents)
50	1142
55	1180
60	1210
65	1222*
70	1207
75	1176
80	1127
85	1067
90	1000
95	929

As can be seen from the asterisk in the table, the number of papers to stock each day is 65. This will result in an expected profit per day of $12.22.

EXERCISES, CHAPTER 20

1. Determine the value of the savevalue for each of the following. All savevalues were initially 0. Assume the following values:

   Q(ONE) is 1
   F(MACH) is 0
   PH1 is 3
   S(TUGS) is 4

```
◇(a)    SAVEVALUE  JOE,PH1+F(MACH),XH
◇(b)    SAVEVALUE  JIM-,S(TUGS),XF
◇(c)    SAVEVALUE  PH1,50/20,XH
◇(d)    SAVEVALUE  2+,5.0/3.0,XH
◇(e)    SAVEVALUE  TOMMY,Q(ONE)-PH1,XL
◇(f)    SAVEVALUE  BILLY,PH1*S(TUGS),XB
```

2. A storage bin can hold 10,000 tons of coal. The amount removed each day varies from 600 to 900 tons (uniform distribution). The stock is checked each morning. When the stock reaches 5,000 tons or less, an order is placed for 1,000 tons. This order takes 7 days to arrive. Simulate for 1,000 days and determine the number of stockouts that occur. (A stockout occurs when there is not enough coal to satisfy the demand for the day.) Initially there are 9,000 tons in the bin.
3. Although not obvious, exercise 2 can be done by using the concept of storages for the storage bin. Redo it by using a storage of 10,000 for the bin. You will need to initialize the program by means of a dummy transaction that has no other purpose than to initialize the storage of the coal in the bin.

SOLUTIONS, CHAPTER 20

1. a. The value of the savevalue JOE is 3.
   b. The value of the savevalue JIM is −4.
   c. The value of the third savevalue is 2 (integer division).
   d. The value of the second savevalue is 1 (integer conversion).
   e. The value of the savevalue TOM is −2.
   f. The value of the savevalue BILL is 12.
2. The program to do the simulation is as follows:

```
         SIMULATE
         INITIAL    XH(STOCK),9000
STOCKDMD TABLE      XH(STOCK),0,500,500
DEMAND   FUNCTION   RN1,C2
0,600/1,900
         GENERATE   1,,,,1
         TEST L     XH(STOCK),5000,ITSOK
         ADVANCE    7
         SAVEVALUE  STOCK+,1000,XH
  ITSOK  TERMINATE
         GENERATE   1
```

```
          SAVEVALUE USED,FN(DEMAND),XH
          TEST G    XH(STOCK),XH(USED),STOCKOUT
          SAVEVALUE STOCK-,XH(USED),XH
          TABULATE  STOCKDMD
          TERMINATE 1
 STOCKOUT SAVEVALUE TIMESOUT+,1,XH
          TERMINATE 1
          START     1000
          PUTPIC    XH(TIMESOUT)
OUTAGES   ***
          END
```

The output is:

OUTAGES 35

3. One such program is as follows:

```
          SIMULATE
          STORAGE   S(STOCK),10000
 STOCKDMD TABLE     S(STOCK),0,500,500
 DEMAND   FUNCTION  RN1,C2
0,600/1,900
          GENERATE  1,,,,1    ONE DAY GOES BY
          TEST L    S(STOCK),5000,ITSOK  IS LEVEL BELOW 5000?
          ADVANCE   7                  WAIT A WEEK
          ENTER     STOCK,1000         ADD COAL TO BIN
  ITSOK   TERMINATE
          GENERATE  1              ONE DAY GOES BY
          ASSIGN    1,FN(DEMAND),PH      WHAT WILL DEMAND BE?
          TEST G    S(STOCK),PH1,STOCKOUT   IS THERE ENOUGH COAL?
          LEAVE     STOCK,PH1          YES, TAKE IT
          TABULATE  STOCKDMD           TABULATE COAL IN BIN
          TERMINATE 1
 STOCKOUT SAVEVALUE TIMESOUT+,1,XH     STOCKOUT HAS OCCURRED
          TERMINATE 1
          GENERATE  ,,,1,2             INITIALIZE COAL IN BIN
          ENTER     STOCK,9000
          TERMINATE
          START     1000
          PUTPIC    XH(TIMESOUT)
 OUTAGES  ***
          END
```

The output is the same as for Exercise 2:

OUTAGES 35

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 21

Logic Switches and Gates

LOGIC SWITCHES

In many simulations, transactions are frequently in a blocked condition in which a certain condition or conditions are true (or false). A given transaction may be forced to remain blocked until the blocking condition becomes false (or true) or may be routed to another block. This blocking can be done by using a TEST block. However, the TEST block is the most inefficient block in GPSS/H. A much better way to block or route a transaction is the GATE block. A GATE block is always used with another condition that (1) allows the transaction to pass to the next sequential block, (2) delays the transaction until the blocking condition changes from true to false (or vice versa), or (3) routes the transaction to another block. Before the GATE block is described further, however, a block that is used primarily with the GATE block will be introduced—the LOGIC block.

THE LOGIC BLOCK

Consider the following program code:

```
◇         TEST E      XH(LOCK),0
◇         -----       -----
◇         -----       -----
◇         GENERATE    ,,,1
◇BACK     ADVANCE     RVEXPO(1,125)
◇         SAVEVALUE   LOCK,1,XH
◇         ADVANCE     RVEXPO(1,12.5)
◇         SAVEVALUE   LOCK,0,XH
◇         TRANSFER    ,BACK
```

When the program begins, the value of the halfword savevalue LOCK is 0 (as are all savevalues unless an INITIAL statement is used to specify nonzero initial values) so the transactions that enter the TEST block will pass through to the next sequential block. The transaction created in the GENERATE block is put on the FEC chain for a time given by sampling from the

exponential distribution with a mean of 125. After this transaction is returned to the CEC, the value of the savevalue LOCK is set equal to 1 (any other value could have been selected for the savevalue for this example). The transaction is then put on the FEC for a time given by sampling from the exponential distribution with a mean of 12.5. Now, any transactions that enter the TEST block will be delayed until the value of the savevalue LOCK becomes equal to 0. This delay might represent a traffic light turning red, a break at a factory for lunch, a breakdown of an assembly line, etc.

Rather than using savevalues and TEST blocks in this manner, there is a better way to handle such conditions in GPSS/H by using switches that are either "on" or "off." The "on" condition is known as "set," and the "off" condition is known as "reset." These switches are turned on and off by the LOGIC block. Its form is as follows:

```
◇        LOGIC R   A
```

where *R* is a relational operator whose value is S for set, R for reset, or I for invert and operand *A* is the name or number of the switch. When a transaction enters a logic block, the effect is as follows:

1. If the logic relationship is S, the switch is put in a set position.
2. If the logic relationship is R, the switch is put in a reset position.
3. If the logic relationship is I, the switch is inverted, i.e., if it was set, it becomes reset or vice versa.

Examples of LOGIC blocks are as follows:

```
         LOGIC S   HALT
◇        LOGIC R   1
◇        LOGIC I   WAIT
```

Notice that logic switches can be named either symbolically or numerically. The usual rules apply for selecting names or numbers for these switches, namely, they start with a letter and can be up to 8 characters in length.

When a program begins, all switches are in a reset position. It is possible to have the switches in a set position by means of the INITIAL statement, a form of which is

```
◇        INITIAL   LS(SWITCHA)/LS(SWITCHB)/LS1
```

An example of this statement is

```
◇        INITIAL   LS(HALT)/LS(PATH)/LS1
```

Alternatively, in place of the parentheses, one can use a single dollar sign. The above line of code could have been written as

```
◇        INITIAL   LS$HALT/LS$PATH/LS1
```

If you have multiple switches all given by numbers, there is a shorthand way to put them in a set position:

```
◇        INITIAL   LS1-5/LS9-12
```

This statement would put logic switches 1 through 5 in a set position and also switches 9 through 12 in a set position. Although rarely needed, LR may also be used in the INITIAL statement to reset logic switches.

Some sample program code might be

```
◇         GENERATE   ,,,1               DUMMY TRANSACTION
◇UPTOP    ADVANCE    RVEXPO(1,200)      MACHINE WORKING
◇         LOGIC S    STOPIT             MACHINE DOWN
◇         ADVANCE    RVNORM(1,20,3.5)   MACHINE BEING FIXED
◇         LOGIC R    STOPIT             MACHINE FIXED
◇         TRANSFER   ,UPTOP             BACK TO WORK
```

This code might be used to represent a machine that works for a certain time and then is shut down when repairs and/or maintenance is performed. The time between breakdowns is given by sampling from the exponential distribution with a mean of 200 time units. When the machine is down, it is fixed or otherwise maintained. This work takes a time that is normally distributed with mean of 20 time units and a standard deviation of 3.5 time units. Notice how the dummy transaction keeps looping in the program segment and alternately sets the logic switch from set to reset. Although it would not be as clear, it would have been all right to have the 2 LOGIC blocks replaced by just

```
◇         LOGIC I    STOPIT
```

THE GATE BLOCK

Logic switches generally need another block in the main program in order to be of any use. This block is often the GATE block. This block works as its name implies, much like a gate in the path of the transactions. When the gate is open, transactions pass through; when it is closed, they either wait until the gate becomes open or they are routed elsewhere.

There are two forms of the GATE block. The first is refusal mode. The general form is

```
◇         GATE R     A,B
```

where *R* is a relational operator such as LS ("logic switch set") or LR ("logic switch reset"), operand *A* is the name or number of a logic switch, and operand *B* is the label of a block to which a transaction will be routed if the logic switch is in a set position. (Operand *B* is omitted in some cases.) An example of this block might be

```
◇         GATE LS    HALT
```

When a transaction arrives at this block, the GPSS/H processor tests to see if the logic switch HALT is in a set position. If so, the transaction moves to the next sequential block. If the logic switch HALT is in a reset position, the transaction remains in the previous block. It also remains on the CEC. However, an internal flag is turned to the "on" position, and the transaction is not scanned again until such time as the switch is turned to an "off" position. The internal flag is turned off when the LOGIC block is entered by another transaction.

If a machine in a simulation is to be periodically shut down, one might use a GATE block as follows:

```
◇          QUEUE     WAIT
◇          GATE LR   STOPIT
◇          SEIZE     MACH1
◇          DEPART    WAIT
◇          ADVANCE   RVEXPO(1,25)
◇          RELEASE   MACH1
```

Whenever the logic switch STOPIT is in a set position, the transaction is kept in the block QUEUE WAIT.

GATE blocks can also be used with facilities and storages as follows:

```
◇          GATE R   A
```

where operand *A* specifies a facility or storage entity such as a machine or a ship's berth and relational operator *R* can be one of the following:

R	Gate Block Tests for
FU	facility in use
FNU	facility not in use
FS	facility can be seized
FNS	facility cannot be seized
SF	storage full
SNF	storage not full
SE	storage empty
SNE	storage not empty

(There are other relationships that can be used with a GATE block, but they will not be used in this book.) Some examples of using facilities and storages in GATE blocks are

```
◇(a)       GATE FU    MACH1
◇(b)       GATE SNE   TUGS
◇(c)       GATE FNU   SHOP
◇(d)       GATE SF    BERTH
```

In (a), the transaction will be held in the previous block if the facility MACH1 is *not* in use. In (b), the transaction is delayed if the storage TUGS is in any situation other than empty. Thus, if the storage of TUGS is 4 and 1 is being used, the transaction will be held up. In (c), the transaction is delayed if the facility SHOP is being used. Finally, in (d), the transaction is held up if the storage BERTH is not full.

THE GATE BLOCK IN CONDITIONAL TRANSFER MODE

When a GATE block has a *B* operand, this is the label of a block. If the GATE is closed, the transaction will be transferred to the block with this label. Thus,

```
◇          GATE LR   HALTIT,AWAY
```

will send the transaction to the block AWAY whenever the logic switch HALTIT is in a set position.

Example 21.1

A contractor is excavating for a large shopping center. He has 5 trucks, numbered 1 to 5, that haul excavated dirt away to a dump. There is a single shovel to load the trucks. The trucks travel in a circuit: load, haul, dump, and return. All times for these activities are found to be normally distributed, as follows:

	Time (minutes)	
Truck Activity	**Mean**	**Standard Deviation**
Loading	2.5	0.35
Traveling to dump	7.5	1.26
Dumping	1.75	0.2
Returning to shovel	5.6	1.1

Only 1 truck can be loaded at a time, but there is no such restriction on dumping. The various times for the trucks are the same regardless of the truck.

The trucks periodically break down and/or must be serviced. Each has a different reliability. The downtimes for the trucks follow the exponential distribution, and the time to repair a truck follows the normal distribution, as shown:

Truck No.	Downtime Distribution	Repair-Time Distribution
1	400	(20,3.5)
2	425	(35,6.9)
3	550	(55.5,12)
4	345	(40,7)
5	300	(50,8)

The downtimes are given as the means for the exponential distribution and the repair times are given as (mean, standard deviation) for the normal distribution. Write the GPSS/H program to simulate the situation by using GATE blocks and LOGIC switches to cause the trucks to be down periodically. Even though the breakdowns of the trucks can be any place in the system, it is sufficient to have the trucks tested for breakdowns after they dump.

Solution

The program is written such that halfword parameter 1 is used to store a number from 1 to 5 to represent the 5 different trucks. When a truck dumps, it is sent to 1 of 5 different GATE blocks depending on what type of truck it is. If a truck is down, the corresponding LOGIC switch will be in a set position, and it will be held up at this point (the transaction will remain in the TRANSFER block). There are 5 program segments to alternately put the logic switches in set and reset positions. Notice how the two segments work. For example, for the first truck, the lines of code for when it is tested to see if it is down and the segment to shut it down are written side by side:

```
◇BLOCKA  □BUFFER                                   □GENERATE  □,,,1,10
◇        □GATE□LR  □FIRST           ◇□BACK1       □ADVANCE   □RVEXPO(1,400)
◇        □TRANSFER □,DOWN           ◇□TIMEA       □LOGIC□S   □FIRST
                                                   □TEST□E    □W(BLOCKA)1
                                                   □ADVANCE   □RVNORM(1,5.6,1.1)
                                                   □LOGIC□R   □FIRST
                                                   □TRANSFER  □,BACK1
```

When a truck transaction is to be tested to see if it is down, it first goes to a BUFFER block. This causes a rescan of the CEC. The dummy transaction that will alternately set and reset the logic switch FIRST has a priority of 10. Thus, in case of a time tie, it will be moved forward first. Let us suppose that the dummy transaction has left the ADVANCE RVEXPO(1,400) block so that the truck is to be down. The switch FIRST is placed in a set position. However, before the transaction can enter the ADVANCE RVNORM(1,5.6,1.1) block to represent the truck being down, it is held up in the block TEST E W(BLOCKA)1. This is so that the truck will, indeed, be delayed. For example, suppose the truck breakdown occurred just after the truck was loaded, this breakdown lasted for 8 minutes, and then the truck took 9 minutes to reach BLOCKA. If the dummy transaction was not delayed, it would immediately enter the ADVANCE RVNORM(1,5.6,1.1) block, and when the truck arrived at BLOCKA, the logic switch FIRST would no longer be in a set position, and the truck therefore would not experience any delay.

The effect of testing for failures in this manner introduces a slight error in that the trucks will continue to operate when they are to be down until they reach the various GATE blocks to delay them. This error can be overcome in several ways: Have more such GATE blocks in the program, which will tend to increase the lines of code, or adjust the statistical distributions to compensate for the error introduced.

The program listing follows. In Chapter 31, the technique of using macros is presented. Macros greatly reduce the length of the programming code.

```
◇         □SIMULATE
◇WHERE    □FUNCTION  □PH1,L5
1,BLOCKA/2,BLOCKB/3,BLOCKC/4,BLOCKD/5,BLOCKE
◇TIMES    □GENERATE  □,,,5
◇         □ASSIGN    □1,N(TIMES),PH
◇UPTOP    □QUEUE     □WAIT
◇         □SEIZE     □SHOVEL
◇         □DEPART    □WAIT
◇         □ADVANCE   □RVNORM(1,2.5,.35)
◇         □RELEASE   □SHOVEL
◇         □ADVANCE   □RVNORM(1,7.5,1.2)
◇         □ADVANCE   □RVNORM(1,1.75,.2)
◇         □TRANSFER  □,FN(WHERE)
◇BLOCKA   □GATE□LR   □FIRST
◇         □TRANSFER  □,DDWN
◇BLOCKB   □GATE□LR   □SECOND
◇         □TRANSFER  □,DDWN
◇BLOCKC   □GATE□LR   □THIRD
```

```
◇          □TRANSFER  □,DDWN
◇BLOCKD    □GATE□LR   □FOURTH
◇          □TRANSFER  □,DDWN
◇BLOCKE    □GATE□LR   □FIFTH
◇DDWN      □ADVANCE   □RVNORM(1,5.6,1.1)
◇          □TRANSFER  □,UPTOP
◇          □GENERATE  □,,,1
◇BACK1     □ADVANCE   □RVEXPO(1,400)
◇TIMEA     □LOGIC□S   □FIRST
◇          □ADVANCE   □RVNORM(1,20,3.5)
◇          □LOGIC□R   □FIRST
◇          □TRANSFER  □,BACK1
◇          □GENERATE  □,,,1
◇BACK2     □ADVANCE   □RVEXPO(1,425)
◇TIMEB     □LOGIC□S   □SECOND
◇          □ADVANCE   □RVNORM(1,35,6.9)
◇          □LOGIC□R   □SECOND
◇          □TRANSFER  □,BACK2
◇          □GENERATE  □,,,1
◇BACK3     □ADVANCE   □RVEXPO(1,550)
◇TIMEC     □LOGIC□S   □THIRD
◇          □ADVANCE   □RVNORM(1,55,10)
◇          □LOGIC□R   □THIRD
◇          □TRANSFER  □,BACK3
◇          □GENERATE  □,,,1
◇BACK4     □ADVANCE   □RVEXPO(1,344)
◇TIMED     □LOGIC□S   □FOURTH
◇          □ADVANCE   □RVNORM(1,40,7)
◇          □LOGIC□R   □FOURTH
◇          □TRANSFER  □,BACK4
◇          □GENERATE  □,,,1
◇BACK5     □ADVANCE   □RVEXPO(1,300)
◇TIMEE     □LOGIC□S   □FIFTH
◇          □ADVANCE   □RVNORM(1,50,8)
◇          □LOGIC□R   □FIFTH
◇          □TRANSFER  □,BACK5
◇          □GENERATE  □480*1000
◇          □TERMINATE □1
◇          □START     □1
◇          □PUTPIC    □LINES=10,FC(SHOVEL)/1000.,_
                       FR(SHOVEL)/10.,N(TIMEA),_
                       N(TIMEB),N(TIMEC),_
                       N(TIMED),N(TIMEE)
0 RESULTS OF SIMULATION FOR 1000 SHIFTS
 LOADS DUMPED PER SHIFT     ****.**
 UTILIZATION OF SHOVEL       ***.**%
 TIMES TRUCK 1 DOWN         ****
 TIMES TRUCK 2 DOWN         ****
 TIMES TRUCK 3 DOWN         ****
 TIMES TRUCK 4 DOWN         ****
 TIMES TRUCK 5 DOWN         ****
◇          □END
```

The output is as follows:

```
RESULTS OF SIMULATION FOR 1000 SHIFTS
LOADS DUMPED PER SHIFT     124.09
UTILIZATION OF SHOVEL       64.62%
TIMES TRUCK 1 DOWN        1104
TIMES TRUCK 2 DOWN        1031
TIMES TRUCK 3 DOWN         770
TIMES TRUCK 4 DOWN        1167
TIMES TRUCK 5 DOWN        1340
```

EXERCISES, CHAPTER 21

1. In Example 21.1, the shovel was busy only 64.62% of the time. Assume that the trucks never failed or, if one did fail, a replacement was immediately available. Determine the utilization of the shovel under this condition.
2. What GATE block can be used to replace the following TEST blocks:

```
(a)   TEST NE   R(TUGS),0
(b)   TEST E    F(MACH1),1
(c)   TEST E    S(BOATS),0
(d)   TEST E    R(BOATS),0
```

3. Write the GATE block in refusal mode so that it will be used to hold the transaction unless the condition is true: (1) The logic switch STOP1 is in a set position. (2) The facility MACH1 is being used. (3) The logic switch STOP1 is in a reset position. (4) The storage of TUGS is all taken.
4. At a storage bin, 1 ton of ore arrives every 1 minute in a Poisson stream. The bin can hold 75 tons. If the bin is full when the ore arrives, it will spill and have to be loaded onto an ore car later. A truck comes along every 60 ± 5 minutes. The truck has room for 100 ± 50 tons of coal. The time to load the truck is insignificant. Simulate for 500 days and determine the average spillage per day (1 day = 3 times 480 minutes).
5. In exercise 21.4, increase the bin by units of 5 up to 90 tons. Determine the spillage per day. Suppose you can speed up the trucks, and they now arrive every 50 ± 5 minutes. Determine the spillage per day for bin sizes of 75 tons and 80 tons.
6. Trucks come to a repair facility every 20 ± 10 minutes. There are 3 repair areas and repairs take 55 ± 20 minutes. The facility also does other repairs, and, on an average of 100 minutes (exponentially distributed), the repair shop works on other projects. These take 10 minutes (also exponentially distributed). Trucks waiting for the repair shop, or trucks that arrive during this time, remain at the shop until it is ready for them. Any trucks being repaired continue to be repaired. Determine the utilization of the shop, the number of trucks repaired in 1 day, and the maximum number of trucks that were waiting.

SOLUTIONS, CHAPTER 21

1. The program to do the simulation is obtained by removing the segments that cause the trucks to be down. It is not necessary to remove the GATE blocks as the logic switches are always reset.

   The results of the simulation are that the average number of loads dumped each day are 133.00 and the utilization of the shovel is 69.25%

2. a. `GATE SNE TUGS`
   b. `GATE FNS MACH`
   c. `GATE SNE BOATS`
   d. `GATE SNE BOATS`

3. a. `GATE LS STOP1`
   b. `GATE FNS MACH1`
   c. `GATE SE TUGS`
   d. `GATE SF TUGS`

4. The program to do the simulation is as follows:

```
          SIMULATE
          STORAGE     S(BIN),80
 INCAR    FUNCTION    RN1,C2
0,50/1,150
 INBIN    TABLE       S(BIN),20,1,75
          GENERATE    RVEXPO(1,1)   ONE TON OF COAL COMES
          GATE SNF    BIN,AWAY
          ENTER       BIN,1
          TERMINATE
 AWAY     SAVEVALUE   OVERFLOW+,1,XH
          TERMINATE
          GENERATE    50,5
          TABULATE    INBIN
          ASSIGN      1,FN(INCAR),PH
          TEST GE     PH1,S(BIN),NOTSO
          LEAVE       BIN,S(BIN)
          TERMINATE
 NOTSO    LEAVE       BIN,PH1
          TERMINATE
          GENERATE   480*3*100
          TERMINATE  1
          START      1
          PUTPIC     LINES=2,TB(INBIN),XH(OVERFLOW)/100.
 AVG. AMOUNT IN BIN WHEN TRUCK COMES  **.**
 TONS OVERFLOWED PER DAY             ***.**
          END
```

   The results of the simulation are:

```
AVG. AMOUNT IN BIN WHEN TRUCK COMES  61.37
TONS OVERFLOWED PER DAY               7.63
```

5. The results of the simulation for different bin sizes are:

Bin Size	Avg. Ore in Bin When Truck Comes	Avg. Spillage
80	60.41	3.41
85	60.52	1.59
90	60.58	0.76

If the truck now arrives every 50 ± 5 minutes, the results are:

Bin Size	Avg. Ore in Bin When Truck Comes	Avg. Spillage
75	49.91	0.17
80	49.93	0.01

It would appear that it is much better to increase the truck speed than the bin size.

6. The program to simulate this is as follows:

```
     SIMULATE
     STORAGE    S(REPAIR),3
     GENERATE   20,10
     QUEUE      WAIT
     GATE LR    SHOPOPEN
     ENTER      REPAIR
     DEPART     WAIT
     ADVANCE    55,20
     LEAVE      REPAIR
     TERMINATE
     GENERATE   ,,,1
BACK ADVANCE    RVEXPO(1,100)
     LOGIC S    SHOPOPEN
     ADVANCE    RVEXPO(1,10)
     LOGIC R    SHOPOPEN
     TRANSFER   ,BACK
     GENERATE   480*3*500
     TERMINATE  1
     START      1
     PUTPIC     LINES=5,SC(REPAIR)/500.,SR(REPAIR)/10.,N(BACK)/500.,_
                QM(WAIT),QA(WAIT)
TRUCKS REPAIRED PER DAY        ***.**
UTIL. OF REPAIR SHOP           ***.**%
NUMBER OF TIMES DOWN PER DAY   ***.**
MAXIMUM QUEUE                  ***
AVERAGE QUEUE LENGTH            **.**
     END
```

The results of the simulation are:

```
TRUCKS REPAIRED PER DAY        71.82
UTIL. OF REPAIR SHOP           91.59%
NUMBER OF TIMES DOWN PER DAY   13.12
MAXIMUM QUEUE                   8
AVERATE QUEUE LENGTH            0.62
```

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 22

Other Forms of the TRANSFER Block

In Chapter 7, three forms of the TRANSFER block were discussed: the unconditional TRANSFER, the conditional TRANSFER, and the TRANSFER BOTH modes. The first two of these are by far the most commonly used. However, there are other forms of the TRANSFER block that can be quite handy when they are needed. Each will be discussed here along with possible applications.

THE TRANSFER BLOCK IN PICK MODE

This form of the TRANSFER block will select a block to which to transfer the transaction at random from a number of possible blocks. The transaction will unconditionally go to this block. Each of the blocks to be selected will have the same probability of being selected. Thus, if there are 3 blocks, each will be selected with a probability of 0.3333; for 4 blocks, the probability of picking a particular one is 0.250, etc. The form of the block is

```
◇        TRANSFER   A,B,C
```

where operand *A* is the word "PICK" and operands *B* and *C* are block labels. The block with the label in operand *B* must appear in the program code before the block with the label in operand *C*. Each block between these two blocks is considered to be in the range of the transfer. This restriction means that, in general, *only TERMINATE and TRANSFER blocks are between the blocks labeled in operands* B *and* C. Consider the following code:

```
◇        TRANSFER   PICK,FIRST,LAST
◇FIRST   TRANSFER   ,MACH1
◇        TRANSFER   ,MACH2
◇        TRANSFER   ,MACH3
◇        TRANSFER   ,MACH4
◇LAST    TRANSFER   ,MACH5
```

A transaction will be routed with equal probability to one of the blocks labeled MACH1, MACH2, MACH3, MACH4, and MACH5.

Example 22.1

What will the following program do?

```
         SIMULATE
         GENERATE  1                 THIS IS BLOCK 1
         ADVANCE   1                 THIS IS BLOCK 2
         TRANSFER  PICK,FIRST,LAST   THIS IS BLOCK 3
FIRST    TERMINATE                   THIS IS BLOCK FIRST
         TERMINATE                   THIS IS BLOCK 5
         TERMINATE                   THIS IS BLOCK 6
         TERMINATE                   THIS IS BLOCK 7
         TERMINATE                   THIS IS BLOCK 8
LAST     TERMINATE                   THIS IS BLOCK LAST
         GENERATE  1000              THIS IS BLOCK 10
         TERMINATE 1                 THIS IS BLOCK 11
         START     1
         END
```

Solution

The program will generate a transaction every 1 time unit. The transaction will be put on the FEC for 1 time unit. When the transaction returns to the CEC, it will then be transferred to one of 6 TERMINATE blocks with equal probability. The following is selected output from running the program for the 999 transactions that enter the TRANSFER block:

```
RELATIVE CLOCK: 1000.0000   ABSOLUTE CLOCK: 1000.0000
BLOCK CURRENT       TOTAL  BLOCK CURRENT       TOTAL
1                     999  11                      1
2           1         999
3                     998
FIRST                 177
5                     164
6                     146
7                     175
8                     170
LAST                  166
10                      1
```

Each of the TERMINATE blocks between and including FIRST and LAST can be expected to be entered approximately 166 times, as shown in the TOTAL columns of the output.

THE TRANSFER BLOCK IN ALL MODE

The TRANSFERALL mode attempts to route the transaction to a series of blocks by trying each one in sequence. For example, if the blocks relating to a truck-repair shop are BAYA, BAYB, BAYC, and BAYD, the transaction attempts to first enter the block labeled BAYA. If it can, it does so. If not, it tries the block labeled BAYB, etc. If all of the blocks are in refusal mode, the transaction waits until the first block in the series is free.

The form of the TRANSFER block in ALL mode is as follows:

```
         TRANSFER  A,B,C,D
```

where the word ALL must be in operand *A*, operands *B* and *C* are block labels that define the range of the transfer, and operand *D* must be a positive integer that indicates that the blocks to which the transaction is to be transferred are *D* program lines (i.e., blocks) apart. For example,

```
◇         □TRANSFER  □ALL,BEGIN,STOP,4
◇BEGIN    □-----
◇         □-----
◇         □-----
◇         □-----
◇BLOCKX   □-----
◇         □-----
◇BLOCKY   □-----
◇         □-----
◇BLOCKZ   □-----
◇         □-----
◇         □-----
◇         □-----
◇STOP     □-----
```

According to the program outline, the transaction first attempts to enter the block labeled BEGIN because that is the value of operand *B* in the TRANSFER block. If the transaction can do so, it does. If not, it attempts to enter the block with the label BLOCKX because, although the BLOCKX block is not directly specified in the TRANSFER block, it is 4 blocks down from the block START. (The labels of intermediate blocks at appropriate spacing–here, BLOCKX and BLOCKZ–need not be in the program.) This attempt of the transaction to enter a block is continued until all blocks 4 lines apart are considered by the GPSS/H processor up to and including block STOP. If all are in refusal mode, the transaction is held in the TRANSFER block until future scans of the CEC show that one of the blocks has become able to accept the transaction.

Example 22.2

A factory has 3 machines to rework failed parts. Parts arrive every 12 ± 4.5 minutes. Machine A is the best and can finish a part every 14 ± 4.3 minutes. If this machine is free, the part goes there; otherwise, it is sent to machine B, which works in 20 ± 5.7 minutes. If both machines A and B are busy, the part is sent to machine C, which is quite slow, working at 30 ± 3.4 minutes. If all machines are busy, the part waits for the first one to be free. Simulate for 100 days of operation of 24 hours per day.

Solution

```
◇         □SIMULATE
◇         □GENERATE  □12,4.5
◇         □ADVANCE   □0
◇         □TRANSFER  □ALL,AAAA,BBBB,4
◇AAAA     □SEIZE     □MACH1
◇         □ADVANCE   □14,4.3
◇         □RELEASE   □MACH1
◇         □TERMINATE
◇         □SEIZE     □MACH2
◇         □ADVANCE   □20,5.7
```

```
◇         RELEASE    MACH2
◇         TERMINATE
◇BBBB     SEIZE      MACH3
◇         ADVANCE    30,3.4
◇         RELEASE    MACH3
◇         TERMINATE
◇         GENERATE   480*3*100
◇         TERMINATE  1
◇         START      1
◇         PUTPIC     LINES=4,FR(MACH1)/10.,FC(MACH1)/100.,_
                      FR(MACH2)/10.,FC(MACH2)/100.,_
                      FR(MACH3)/10.,FC(MACH3)/100.
 FACILITY       UTIL. PARTSPERDAY
 MACH1          ***.**%      ***.**
 MACH2          ***.**%      ***.**
 MACH3          ***.**%      ***.**
◇         END
```

The results of running the simulation are as follows:

```
FACILITY       UTIL.         PARTS PER DAY
MACH1          68.38%        70.27
MACH2          57.39%        41.24
MACH3          16.82%         8.09
```

The output shows that the first machine is busy 68.38% of the time and is repairing 70.27 parts per day. The second machine is busy 57.39% of the time and is repairing 41.24 parts per day, while the third machine is working only 16.82% of the time and is repairing 8.09 parts per day.

THE TRANSFER BLOCK IN FUNCTION MODE

The TRANSFER PICK transfers the transactions to different blocks with equal probability. Sometimes you want to transfer a transaction to particular blocks with set but different percentage probabilities for each. The TRANSFER block in function mode is used for this procedure. Since this form of the TRANSFER block is so similar to the unconditional TRANSFER block, it has been used in previous chapters in a simple form. However, it will be discussed in detail here. There can be several forms, but the one most frequently used is

```
◇         TRANSFER   ,B
```

where operand *A* is omitted (as shown by the comma) and the *B* operand is FN (for "function") followed by the block label of the line of code defining the function. The function referenced can have blocks in the number pairs in the function definition. For example,

```
◇FIRST    FUNCTION   RN1,D4
.1,BLOCKA/.35,BLOCKB/.8,BLOCKC/1,BLOCKD
◇         -----
◇         -----
◇         TRANSFER   ,FN(FIRST)
```

BLOCKA, BLOCKB, BLOCKC, and BLOCKD are block labels. The transaction will be transferred to BLOCKA 10% of the time, to BLOCKB 25% of the time,

to BLOCKC 45% of the time, and to BLOCKD 15% of the time. This is a very useful form of the TRANSFER block.

Example 22.3

Trucks arrive for service at a repair shop. Interarrival times are 28 ± 6 minutes. Serious repairs are needed for 10% of the trucks, and a special crew is called in to do the work. These repairs take 200 ± 60 minutes to complete. The other services can be broken into two types, type B and type C. Type B service is required by 30% of the trucks, and type C service is needed by the remaining 60%. Both of these types of service are done by the same crew. Type B service takes 45 ± 15 minutes, and type C service takes 20 ± 6 minutes. Simulate for 20 shifts of 8 hours each.

Solution

```
◇         SIMULATE
◇WHICH    FUNCTION   RN1,D3
.1,SERVA/.4,SERVB/1,SERVC
◇         GENERATE   28,6
◇         QUEUE      WAIT
◇         TRANSFER   ,FN(WHICH)
◇SERVA    SEIZE      OTHER
◇         DEPART     WAIT
◇         ADVANCE    200,60
◇         RELEASE    OTHER
◇         TERMINATE
◇SERVB    SEIZE      FIRST
◇         DEPART     WAIT
◇         ADVANCE    45,15
◇         RELEASE    FIRST
◇         TERMINATE
◇SERVC    SEIZE      FIRST
◇         DEPART     WAIT
◇         ADVANCE    20,6
◇         RELEASE    FIRST
◇         TERMINATE
◇         GENERATE   480*20
◇         TERMINATE  1
◇         START      1
◇         PUTPIC     LINES=3,FC(OTHER)/100.,FR(OTHER)/10.,_
                      FC(FIRST)/100.,FR(FIRST)/10,QM(WAIT),QA(WAIT)
◇         END
```

The results from running the program are as follows:

```
TIMES FACILITY OTHER USED/DAY  5.19   UTIL OF FACILITY OTHER  71.62%
TIMES FACILITY FIRST USED/DAY  46.12  UTIL OF FACILITY FIRST  90.13%
 MAXIMUM QUEUE                 10      AVERAGE QUEUE           1.82
```

When using the TRANSFER function mode, it is possible to have a function that returns a number. This number then refers to the number of the program block to which the transaction is to be transferred. (Recall that the GPSS/H

processor numbers all of the *blocks* consecutively during compiling, as shown in the .LIS file.) For example, if the block

```
◇        TRANSFER  ,FN(WHERE)
```

returns the value 12, the transaction is routed to the block in numerical position 12. The problem with using this form of the TRANSFER block is that, if any changes are made to the program such as adding or deleting blocks, all such TRANSFER blocks must also be changed.

It is also possible to include arithmetic operations, such as in the following:

```
◇        TRANSFER  ,FN(THIRD)+5
```

The function THIRD is referenced and a value is returned, say, 14. Then 5 is added to 14, and the transaction is routed to the block in position 19.

THE TRANSFER BLOCK IN PARAMETER MODE

It is possible to transfer to a block number whose value is given by one of the transaction's parameters. The form of this TRANSFER mode is shown by

```
◇        TRANSFER  ,B,C
```

where operand *A* is blank (again shown by the comma), operand *B* is the transaction's parameter that identifies the block to which the transaction will be transferred, and operand *C* (may be omitted) is a positive or negative value that is added to the value stored in the parameter given in operand *B*. Consider the code

```
◇        TRANSFER  ,PH4
```

In this example, the transaction is routed to the block given by the value of the transaction's fourth halfword parameter. If this value is 15, the transaction is routed to block 15; if the value is 20, the transaction goes to block 20, etc.

The following code is an example of use of the *C* operand in such transfer blocks:

```
◇        TRANSFER  ,PH7,3
```

Now, 3 is added to the value stored in the transaction's seventh halfword parameter, and the transaction is routed to the block whose number is given by this total. For example, if the value stored in halfword parameter 7 was 30, the transaction would be routed to block 33. However, because it is possible to have arithmetic in operands, the use of the *C* operand in the TRANSFER block is hardly ever used. For example, the above transfer also could have been coded as

```
◇        TRANSFER  ,PH7+3
```

The simulation in Example 22.3 could have been written by using the TRANSFER block in parameter mode as follows:

```
◇         SIMULATE
◇WHICH    FUNCTION   RN1,D3
.1,4/.4,10/1,16
◇         GENERATE   28,6
◇         ASSIGN     5,FN(WHICH),PH
◇         TRANSFER   ,PH5
◇         QUEUE      WAIT
◇         SEIZE      OTHER
◇         DEPART     WAIT
◇         ADVANCE    200,60
◇         RELEASE    OTHER
◇         TERMINATE
◇         QUEUE      WAIT
◇         SEIZE      FIRST
◇         DEPART     WAIT
◇         ADVANCE    45,15
◇         RELEASE    FIRST
◇         TERMINATE
◇         QUEUE      WAIT
◇         SEIZE      FIRST
◇         DEPART     WAIT
◇         ADVANCE    20,6
◇         RELEASE    FIRST
◇         TERMINATE
◇         GENERATE   480*20
◇         TERMINATE  1
◇         START      1
◇         PUTPIC     LINES=3,FC(OTHER)/100.,FR(OTHER)/10.,_
                      FC(FIRST)/100.,FR(FIRST)/10,QM(WAIT),QA(WAIT)
 TIMES FACILITY OTHER USED/DAY **.**  UTIL OF FACILITY OTHER ***.**%
 TIMES FACILITY FIRST USED/DAY **.**  UTIL OF FACILITY FIRST ***.**%
  MAXIMUM QUEUE                 **      AVERAGE QUEUE          **.**
◇         END
```

The output from this program is identical to that of the previous one.

THE TRANSFER BLOCK IN SUBROUTINE MODE

It is possible to transfer to a subroutine and then return to the main program. This approach might be taken when there are a series of identical blocks that have to be repeatedly referenced. The general form used is

```
◇         TRANSFER   A,B,C
```

The "word" SBR *must* be in operand *A*. Operand *B* contains the block label to transfer the transaction to, i.e., the initial block of the subroutine. Upon entering the TRANSFER SBR block, the transaction is unconditionally routed to the subroutine block. The GPSS/H processor places the block number of the TRANSFER SBR block in the transaction's parameter as specified by operand *C*. *Unlike in other programming languages, when the transaction finishes with the subroutine, it does not automatically return to the main program.* The transaction must be directed back to the block it came from in order to continue moving from block to block. To accomplish returning the transaction to the

main program once the transaction finishes the subroutine, the following form of code can be used:

```
◇           TRANSFER  ,B
```

where the *A* operand is again blank and the *B* operand is the transaction's parameter that was listed in the *C* operand of the original TRANSFER SBR block plus 1. The effect of this line of code is that the transaction is transferred back to the main program at the block below the one that transferred the transaction to the subroutine. For example, if the initial TRANSFER SBR block is

```
◇           TRANSFER  SBR,SUBFIVE,2PH
```

then the code to return the transaction to the main program would be

```
◇           TRANSFER  ,PH2+1
```

THE TRANSFER BLOCK IN SIMULTANEOUS MODE

Another form of the TRANSFER block is the TRANSFER SIM mode. This mode was used more in earlier versions of GPSS when it was more important to write code that took the minimum time for execution. This form of the TRANSFER block makes use of the fact that, every time a transaction is delayed, it has a switch called the SIM indicator that is put in a set position. Whenever a transaction leaves an ADVANCE block, this switch is reset. Also when a transaction leaves the TRANSFER SIM block, the switch is reset. The general form of this block is

```
◇           TRANSFER  SIM,B,C
```

The "word" SIM must be in operand *A*. Operand *B* is the label of the block where the transaction is routed when the switch is in a reset position. It is rarely used because the next sequential block is often the one to route the transaction to. In this regard, TRANSFER SIM is similar to the TRANSFER BOTH block. Operand *C* is the block where the transaction is routed when the switch is in a set position. This block is usually the first of a series of GATE blocks or other blocks used in logic testing.

Consider the following series of blocks that might have to do with a ship ready to enter a harbor. In order for the ship to enter, three conditions must be satisfied simultaneously: (1) There must be a high tide. (2) The single tugboat must be available. (3) A berth must be free.

```
◇BACK1      GATE LS     ONE           IS THERE A HIGH TIDE?
◇           GATE FNU    TUG           IS THE TUG AVAILABLE?
◇           GATE SNF    BERTH         IS A BERTH FREE?
◇           TRANSFER    SIM,,BACK1
```

The first GATE block has to do with the fact that there must be a high tide. The second and third control the availability of the tugboat and the berth.

Suppose that a transaction arrives at the first of these blocks. The transaction's SIM switch is reset. If the transaction is not delayed at any of the three GATE blocks, it simply leaves the TRANSFER SIM block and enters the next sequential block.

Alternatively, suppose that there was no high tide. The transaction waits in the block before BACK1 until high tide. Since it has now been delayed, its SIM switch is placed in a set position. If the other two GATE blocks allow it to pass through, it arrives at the TRANSFER SIM block with its SIM indicator in a set position. While in the TRANSFER SIM block, the transaction's SIM switch is placed in a reset position, and then it is routed to try to enter the BACK1 block again, as given by the *C* operand. If there is still a high tide, the transaction's switch is reset; if the two succeeding blocks also allow it to pass through, then when it again encounters the TRANSFER SIM block, it passes through to the next sequential block.

Suppose that another transaction enters the series of three blocks at a later time. The tide is high but the tugboat is busy. The transaction is held at the second GATE block until a tugboat is free. Now, suppose that, while the ship was waiting for the tugboat, the tide changes to a low tide. When the transaction reaches the TRANSFER SIM block, the switch is in a set position because of the delay in waiting for the tugboat. Thus, it will be routed to the first GATE block. But now the tide is low, so the transaction will be delayed here until the tide becomes high. The transaction will continue to cycle through the three GATE blocks and the TRANSFER SIM block until all three conditions as given by the GATE blocks are satisfied.

The above situation could have been simulated by using a single TEST block and Boolean logic (covered in Chapter 27), but this approach causes more execution time than using the three GATE blocks. Whenever possible, the programmer is encouraged to avoid the use of the TEST block because it increases execution time.

EXERCISES, CHAPTER 22

1. What will happen when a transaction enters the following TRANSFER PICK block?

```
◇        ▯TRANSFER  ▯PICK,FIRST,LAST
◇FIRST   ▯TRANSFER  ▯,AAAA
◇        ▯TRANSFER  ▯,BBBB
◇        ▯TRANSFER  ▯,CCCC
◇        ▯TERMINATE
◇        ▯TRANSFER  ▯,DDDD
◇LAST    ▯TRANSFER  ▯,EEEE
```

2. Explain what happens for each of the following TRANSFER function blocks:

```
◇WHERE   ▯FUNCTION  ▯RN1,D3        ▯(a)
.25,BLOCKA/.8,BLOCKB/1,BLOCKC
◇        ▯-----
◇        ▯-----
◇        ▯TRANSFER  ▯,FN(WHERE)
◇WHAT    ▯FUNCTION  ▯PH1,L4        ▯(b)
,AAAA/,BBBB/,CCCC/,DDDD
◇        ▯-----
◇        ▯-----
◇        ▯TRANSFER  ▯,FN(WHAT)
```

3. Ships arrive at a port every 8 ± 2 hours, and there are 3 berths for the ships. A ship will remain in a berth for 21 ± 4 hours. It cannot enter or leave when the following conditions hold:
   a. A storm has closed the harbor. These come up on average every 24 hours (exponentially distributed). They last for an average of 2.5 hours, also exponentially distributed.
   b. Miscellaneous work stoppages occur every 12 ± 2 hours and last for 1 ± 0.5 hours.
   c. The tide is out every 10 ± 3 hours, and this condition remains for 1.5 ±0.5 hours.

   Model the harbor to determine how busy the berths are and the maximum number of ships that had to wait in the queue. Model for 5 years.
4. Run the following program and discuss the output.

```
◇         ▯SIMULATE
◇MYSUB    ▯TEST▯NE    ▯XL(TIMER),AC1,AWAY
◇         ▯BPUTPIC    ▯AC1
◇THE TIME IS NOW ***.**
◇         ▯SAVEVALUE ▯TIMER,AC1,XL
◇AWAY     ▯TRANSFER  ▯,PH1+1
◇         ▯GENERATE  ▯18,6
◇         ▯TRANSFER  ▯SBR,MYSUB,1PH
◇         ▯QUEUE     ▯WAIT
◇         ▯SEIZE     ▯BARBER
◇         ▯DEPART    ▯WAIT
◇         ▯ADVANCE   ▯16,8
◇         ▯RELEASE   ▯BARBER
◇         ▯TRANSFER  ▯SBR,MYSUB,1PH
◇         ▯TERMINATE
◇         ▯GENERATE  ▯60*2
◇         ▯TERMINATE ▯1
◇         ▯START     ▯1
◇         ▯END
```

5. In a factory, parts come along every 20 ± 10 seconds. Machine 1, which takes 35 ± 4 seconds, is used for 50% of the parts; machine 2, which takes 77 ± 3.5 seconds, is used for 25% of the parts; machine 3, which takes 85 ± 5 seconds, is used for 15% of the parts; and machine 4, which takes 150 ± 7 seconds, is used for 10% of the parts. Simulate for 100 days of 1440 minutes each. Determine the utilization of each of the 4 machines. Use a TRANSFER function block.
6. It is possible to write the program in exercise 22.5 by using a TRANSFER block as a parameter. Make the necessary changes to use this approach.

SOLUTIONS, CHAPTER 22

1. The transaction will be routed to any of the 6 blocks between and including the one labeled FIRST and LAST with equal probability. In 5 of these cases, the transaction will be routed to different parts of the program; in the other, it is terminated.

2. The transaction will be routed to the block with the label BLOCKA 25% of the time; it will be routed to the block with the label BLOCKB 55% of the time and it will be routed to the block with the label BLOCKC 20% of the time.
3. The program to do the simulation is given:

```
         SIMULATE
         STORAGE     S(BERTH),3
         GENERATE    8,2
         QUEUE       HARBOR
BACK1    GATE LR     STORM
         GATE LR     STOPAGE
         GATE LR     TIDE
         TRANSFER    SIM,,BACK1
         ENTER       BERTH
         DEPART      HARBOR
         ADVANCE     21,4
BACK2    GATE LR     STORM
         GATE LR     STOPAGE
         GATE LR     TIDE
         TRANSFER    SIM,,BACK2
         LEAVE       BERTH
         TERMINATE
         GENERATE    ,,,1
BACK3    ADVANCE     RVEXPO(1,24)    STORM COMING
         LOGIC S     STORM
         ADVANCE     RVEXPO(1,2.5)
         LOGIC R     STORM
         TRANSFER    ,BACK3
         GENERATE    ,,,1
BACK4    ADVANCE     12,2            WORK STOPPAGE
         LOGIC S     STOPAGE
         ADVANCE     1,.5
         LOGIC R     STOPAGE
         TRANSFER    ,BACK4
         GENERATE    ,,,1
BACK5    ADVANCE     10,3            TIDE OK
         LOGIC S     TIDE
         ADVANCE     1.5,.5
         LOGIC R     TIDE
         TRANSFER    ,BACK5
         GENERATE    24*365*5
         TERMINATE   1
         START       1
         PUTPIC      LINES=2,SR(BERTH)/10.,QM(HARBOR)
UTIL. OF BERTHS  ***.**%
MAXIMUM QUEUE     **
         END
```

The result of running the simulation give the following:

```
UTIL. OF BERTHS  89.83%
MAXIMUM QUEUE     3
```

4. The program puts the simulation clock time on the screen every time it changes. If the subroutine is called and the clock time has not changed, nothing is done. (For this example, every time the subroutine is called, there will be a change in the clock time.
5. The program for this example follows.

```
         SIMULATE
WHERE    FUNCTION   RN1,D4
.5,BLOCKA/.75,BLOCKB/.90,BLOCKC/1,BLOCKD
         GENERATE   20,10
         QUEUE      WAIT
         TRANSFER   ,FN(WHERE)
BLOCKA   SEIZE      MACH1
         DEPART     WAIT
         ADVANCE    35,4
         RELEASE    MACH1
         TERMINATE
BLOCKB   SEIZE      MACH2
         DEPART     WAIT
         ADVANCE    77,3.5
         RELEASE    MACH2
         TERMINATE
BLOCKC   SEIZE      MACH3
         DEPART     WAIT
         ADVANCE    85,5
         RELEASE    MACH3
         TERMINATE
BLOCKD   SEIZE      MACH4
         DEPART     WAIT
         ADVANCE    150,7
         RELEASE    MACH4
         TERMINATE
         GENERATE   480*100*3
         TERMINATE  1
         START      1
         PUTPIC     LINES=4,FR(MACH1)/10.,FR(MACH2)/10.,_
                    FR(MACH3)/10.,FR(MACH4)/10.
  UTIL. OF MACH1   ***.**%
  UTIL. OF MACH2   ***.**%
  UTIL. OF MACH3   ***.**%
  UTIL. OF MACH4   ***.**%
         END
```

The results of running the simulation are:

```
UTIL. OF MACH1   87.21%
UTIL. OF MACH2   77.35%
UTIL. OF MACH3   65.54%
UTIL. OF MACH4   72.26%
```

6. The modified program is:

```
         SIMULATE
WHERE    FUNCTION    RN1,D4
.5,5/.75,10/.90,15/1,20
         GENERATE    20,10
         QUEUE       WAIT
ASSIGN   1,FN(WHERE),PH
         TRANSFER    ,PH1
         SEIZE       MACH1
         DEPART      WAIT
         ADVANCE     35,4
         RELEASE     MACH1
         TERMINATE
         SEIZE       MACH2
         DEPART      WAIT
         ADVANCE     77,3.5
         RELEASE     MACH2
         TERMINATE
         SEIZE       MACH3
         DEPART      WAIT
         ADVANCE     85,5
         RELEASE     MACH3
         TERMINATE
         SEIZE       MACH4
         DEPART      WAIT
         ADVANCE     150,7
         RELEASE     MACH4
         TERMINATE
         GENERATE    480*100*3
         TERMINATE   1
         START       1
         PUTPIC      LINES=4,FR(MACH1)/10.,FR(MACH2)/10.,_
                      FR(MACH3)/10.,FR(MACH4)/10.
   UTIL. OF MACH1    ***.**%
   UTIL. OF MACH2    ***.**%
   UTIL. OF MACH3    ***.**%
   UTIL. OF MACH4    ***.**%
          END
```

Although the program gives identical results to the one used to simulate exercise 18.5, this program is harder to follow since the blocks where the transfer takes place do not have labels.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 23

Ampervariables; DO Loops; GETLIST Statements; IF, GOTO, HERE, and LET Statements

AMPERVARIABLES

It is possible to have variables that change in a GPSS/H program each time it is run. We have done this already by redefining the block that we wanted to be changed. For example, a program that was run once with 4 workers in a factory had

```
WORKERS GENERATE ,,,4
```

After the first run, we might have had statements such as

```
        START    1
        CLEAR
        RMULT    777
WORKERS GENERATE ,,,5
        START    1
```

Now the program is run a second time but with 5 worker transactions being used in the simulation. If it is desired to run the program again but now with 6 workers, it is easy to add the necessary lines of code. However, it is possible in GPSS/H to simplify this procedure further by using the concept of *ampervariables*. These are variables that have their values changed during the running of the program. They are defined by the use of the ampersand as their first character (hence, their name).

GPSS/H allows for 5 types of these ampervariables: integer, real or floating point, 2 character types, and external. Integer ampervariables cannot have a decimal point; real or floating-point ampervariables must have a decimal point; character ampervariables are for character strings; and external ampervariables refer to external functions and subroutines. In the following discussion, only integer, real, and character ampervariables are covered as they are the ones most commonly used. The other two are not used in this book.

All ampervariables must be defined prior to their use. They are defined by a statement, as follows:

```
◇        INTEGER    A,B,C,...
◇        REAL       A,B,C,...
◇        CHAR*n     A,B,C,...
```

where the operands are the list of ampervariables. INTEGER and REAL ampervariables may have as many as 8 alphanumeric characters. Character (CHAR) ampervariables have their maximum (≤255) length specified by the *n*. Thus,

```
◇(a)     INTEGER    &I,&JOE,&K123456,&JJJ,&XYZ
◇(b)     REAL       &ZX,&KLMN,&TRUCKS,&SPEED
◇(c)     CHAR*2     &ANS,&YES,&NO
```

The code labeled (a) defines the integer ampervariables I, JOE, K123456, JJJ, and XYZ. The code labeled (b) defines the real ampervariables ZX, KLMN, TRUCKS, and SPEED. That in (c) defines three character ampervariables of length 2 (note that the length of the name of an ampervariable may be different from the length of the string it represents). In the main program blocks, it is then possible to have the following, for example:

```
◇        QUEUE      &I
◇        ADVANCE    &SPEED
◇        GENERATE   ,,&JJJ
◇        SEIZE      &JOE
◇        ASSIGN     1,&ZX,PL
```

One defines values for ampervariables by the LET statement or the BLET block. These are used as follows:

```
◇        LET        &I=4
◇        LET        &SPEED=45.67
◇        LET        &ANS='Y'
```

Notice that character ampervariables are enclosed in single quotation marks. In Chapter 8 it was mentioned that ASCII extended characters (e.g., the ± symbol) are not normally allowed in the PUTPIC statement. However, recall that one can have the following:

```
◇        CHAR*1     &PLUSMNUS
◇        LET        &PLUSMNUS='±'
```

Then, one can place &PLUSMNUS in a PUTPIC statement and the output will contain the ± symbol.

THE GPSS/H DO LOOP

GPSS/H has DO loops that can be used to greatly shorten the code for control statements. Integer ampervariables are commonly used in connection with DO loops. The form is quite similar to that found in other programming languages:

```
◇        DO        A,B,C
◇        -----
◇        -----
◇        ENDDO
```

where operand *A* is an integer ampervariable and has the form &I=lower limit, operand *B* is the upper limit, and operand *C* is the increment. Explicitly giving the increment is optional. It is not possible to decrement the ampervariable in a DO loop.

Here is how the DO loop works. The integer ampervariable is set equal to its lower limit, and the statements from the DO statement down to the ENDDO statement are executed. The integer ampervariable is then incremented. If the increment is missing (as it often is), the value is assumed to be 1 by default. The statements are then executed up to ENDDO. This looping through the code is continued until the value of the ampervariable is greater than the upper limit. Thus, the lines of code

```
◇        INTEGER   &I
◇        DO        &I=2,10
◇        CLEAR
◇        RMULT     9
◇WORKS   GENERATE  ,,,&I
◇        START     1
◇        ENDDO
```

would run the program first with I = 2 and then with I = 3, etc., up to and including I = 10 for a total of 9 times. There would be 9 reports written.

The lines of code

```
◇        INTEGER   &K
◇        DO        &K=4,10,2
◇        CLEAR
◇        RMULT     777
◇        INITIAL   XH(VALUE),&K+3
◇        START     1
◇        ENDDO
```

would run the program first for values of K = 4 and XH(VALUE) = 7. The second time, the values would be K = 6 (increment is 2) and XH(VALUE) = 9, etc., up to K = 10 and XH(VALUE) = 13.

It is possible to have nested DO loops. Each DO loop must have its own ENDDO statement:

```
◇        INTEGER   &J
◇        DO        &J=2,6
```

```
◇       □CLEAR
◇       □RMULT      □54321
◇TRUCKS □GENERATE   □,,,&J
◇       □DO         □&I=1,3
◇       □STORAGE    □S(TUGS),&I
◇       □START      □1
◇       □ENDDO
◇       □ENDDO
```

The value of &J is first set equal to 2. The block

```
◇TRUCKS □GENERATE   □,,,&J
```

would be

```
◇TRUCKS □GENERATE   □,,,2
```

next, the value of I is 1, and so the STORAGE is

```
◇       □STORAGE    □S(TUGS),1
```

The program is run for these values. The value of I is next incremented to 2, and the program run with the STORAGE of TUGS equal to 2 (&J remains equal to 2). Thus, the main program will be executed 18 times. (&J = 2, 3, 4, 5, 6, and &I = 1, 2, and 3.)

SIMULATION OF A TIRE SUPPLY PROBLEM

Example 23.1

The owner of a garage stocks snow tires for sale during the winter months. He places one order at the end of summer and cannot receive any more tires if the demand is greater than the supply. In addition to a $20.00 service charge per order, tires cost $25.00 times the number of tires ordered. The $20 is a fixed cost no matter how many tires are ordered. The tires sell for $45 each. If any tires are left over, there is a penalty of $5 per tire in "holding" costs. If a person wants to buy a tire and it is no longer in stock, the $20 profit is considered as a loss. From past records, the owner feels that the demand for tires is given by the following relative probability distribution:

Demand for Tires	Relative Probability
100	0.03
105	0.05
110	0.10
115	0.15
120	0.18
125	0.14
130	0.12
135	0.10
140	0.08
145	0.03
150	0.02

Determine the number of tires the garage owner should order to maximize his expected profit. Simulate for 200 winters.

Solution

The way to do the simulation is to first assume a supply amount of a reasonable amount of tires. Suppose this amount is 120. Then by using the owner's data on demand distribution, a demand is simulated by means of Monte Carlo simulation. Suppose this is 100. This means that the store made a profit of (100 × $45) – $20 – ($120 × 25) – [$5 × ($120 – $100)] = $1,380. If the demand had been 130, however, the profit would have been (120 × 45) – $20 – ($120 × 25) – [$20 × ($130 – $120)] = $2,180.

Such computations are done for a large number of possible demands, say 200. The expected profit is then the average of the simulated ones. This result is then taken to be the expected profit (or loss) for the given supply amount. The following program to do the simulation assumes supply amounts of 90 to 150 tires in increments of 5, i.e., amounts of 90, 95, 100, . . . , 150.

GPSS/H is ideal for such inventory problems. The program to do the simulation is given below:

```
◇       ▯SIMULATE
◇       ▯INTEGER   ▯&I,&STOCK,&DMD
◇       ▯REAL      ▯&PROFIT,&RESULT
◇MYOUT  ▯FILEDEF   ▯'TIRES.OUT'
◇       ▯PUTSTRING ▯('▯▯')
◇       ▯PUTSTRING ▯('▯▯')
◇       ▯PUTSTRING ▯(' SIMULATION OF A TIRE SUPPLY PROBLEM')
◇       ▯PUTSTRING ▯('▯▯')
◇       ▯PUTSTRING ▯(' A DEALER PLACES AN ORDER FOR TIRES')
◇       ▯PUTSTRING ▯(' THIS IS DONE ONLY ONE TIME')
◇       ▯PUTSTRING ▯(' THESE COST $20 + $25*(NUMBER ORDERED)')
◇       ▯PUTSTRING ▯(' TIRES SELL FOR $45 EACH')
◇       ▯PUTSTRING ▯(' IF ANY TIRES ARE LEFT OVER, A PENALTY')
◇       ▯PUTSTRING ▯(' OF $5.00 PER TIRE RESULTS')
◇       ▯PUTSTRING ▯(' IF A PERSON WANTS TO BUY A TIRE AND IT')
◇       ▯PUTSTRING ▯(' IS NOT IN STOCK, THE $20 PROFIT IS')
◇       ▯PUTSTRING ▯(' CONSIDERED TO BE A LOSS')
◇       ▯PUTSTRING ▯(' THE DEMAND PROBABILITY DISTRIBUTION IS GIVEN')
◇       ▯PUTSTRING ▯('▯▯')
◇       ▯PUTSTRING ▯(' THE SIMULATION IS IN PROGRESS...')
◇DEMAND ▯FUNCTION  ▯RN1,D11
.03,100/.08,105/.18,110/.33,115/.51,120/.65,125/.77,130
.87,135/.95,140/.98,145/1,150
◇       ▯GENERATE  ▯,,,1
◇BACK1  ▯BLET      ▯&DMD=FN(DEMAND)
◇       ▯TEST▯GE   ▯&STOCK,&DMD,DDDD
◇       ▯BLET      ▯&PROFIT=&PROFIT+&DMD*45.-(20.+25.*&STOCK)-_
                    5.*(&STOCK-&DMD)
◇       ▯ADVANCE   ▯1
◇       ▯TRANSFER  ▯,BACK1
◇DDDD   ▯BLET      ▯&PROFIT=&PROFIT+&STOCK*45.-(20.+25.*&STOCK)-_
                    20.*(&DMD-&STOCK)
```

```
◇         ADVANCE    1
◇         TRANSFER   ,BACK1
◇         DO         &I=90,150,5
◇         CLEAR
◇         RMULT      777777
◇         LET        &STOCK=&I
◇         GENERATE   200
◇         TERMINATE  1
◇         START      1
◇         LET        &RESULT=&PROFIT/200.
^         PUTPIC     LINES=3,FILE=TIRES.OUT,&I,&RESULT
 RESULTS OF TIRE SHOP SIMULATION
 NUMBER OF ITEMS IN STOCK WAS          ***
 WITH THESE, THE EXPECTED PROFIT IS  ****.**
◇         LET        &PROFIT=0
◇         ENDDO
◇         END
```

While the simulation takes place, the following is shown on the screen:

```
A DEALER PLACES AN ORDER FOR TIRES
THIS IS DONE ONLY ONE TIME
THESE COST $20 + $25*(NUMBER ORDERED)
TIRES SELL FOR $45 EACH
IF ANY TIRES ARE LEFT OVER, A PENALTY
OF $5.00 PER TIRE RESULTS
IF A PERSON WANTS TO BUY A TIRE AND IT
IS NOT IN STOCK, THE $20 PROFIT IS
CONSIDERED TO BE A LOSS
THE DEMAND PROBABILITY DISTRIBUTION IS GIVEN

THE SIMULATION IS IN PROGRESS...
```

The output file TIRES.OUT has been created to hold the results of the simulation. Had they been sent to the screen, it would not have been possible to read them as they would scroll by too rapidly. The file TIRES.OUT is as follows:

```
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS          90
  WITH THESE, THE EXPECTED PROFIT IS  1102.50
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS          95
  WITH THESE, THE EXPECTED PROFIT IS  1302.50
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         100
  WITH THESE, THE EXPECTED PROFIT IS  1502.50
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         105
  WITH THESE, THE EXPECTED PROFIT IS  1693.75
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         110
  WITH THESE, THE EXPECTED PROFIT IS  1864.00
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         115
  WITH THESE, THE EXPECTED PROFIT IS  2009.75
```

```
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         120
  WITH THESE, THE EXPECTED PROFIT IS  2110.00
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         125
  WITH THESE, THE EXPECTED PROFIT IS  2131.50
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         130
  WITH THESE, THE EXPECTED PROFIT IS  2112.75
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         135
  WITH THESE, THE EXPECTED PROFIT IS  2050.25
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         140
  WITH THESE, THE EXPECTED PROFIT IS  1954.50
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         145
  WITH THESE, THE EXPECTED PROFIT IS  1818.50
RESULTS OF TIRE SHOP SIMULATION
  NUMBER OF ITEMS IN STOCK WAS         150
  WITH THESE, THE EXPECTED PROFIT IS  1673.75
```

From the output, it is found that the optimum number of tires to order is 125, which will result in an expected profit of $2,131.50. However, if 120 or 130 are ordered, the expected profit is only slightly less. If desired, the program can be rerun by using supply amounts from 120 to 130 in increments of 1. (Note: This example is modified from one found in Cooper et al. [1977, p. 344].)

THE GETLIST STATEMENT

Once ampervariables are defined, they can be read into the program by means of the GETLIST statement. This causes the program execution to stop until the ampervariables as specified in GETLIST are read from the user's input device. A prompt appears on the screen until the data are input. The general form of this statement is

```
◇LABEL   ▯GETLIST  ▯A,B,C,...
```

where the label is optional, the *A* operand is FILE=filename, and the remaining operands are ampervariables, the values of which must be input by the user. The filename GUSER is commonly used for interactive screen input. (However, the FILE = GUSER specification can be omitted as GPSS/H assumes this is the file by default.) Multiple ampervariables may be read in by a single GETLIST statement, for example,

```
◇        ▯GETLIST  ▯FILE=GUSER,&I,&J,&K
```

If values of 2, 3, and 7 are to be assigned to &I, &J, and &K, respectively, the data must be input as

```
2 3 7
```

i.e., they *must be separated by blanks, not commas.* If a GETLIST statement asks for more than 1 value to be read in and fewer than this number is input, the prompt remains on the screen until all the data are input.

Example 23.2

Go back to example 23.1 and add the necessary code so that the number of years to simulate for is a variable. Also, have the supply amounts and the increments as variables.

Solution

The changes are as follows: Replace

```
◇        ⎕INTEGER    ⎕&I,&STOCK,&DMD
```

with

```
◇        ⎕INTEGER    ⎕&I,&J,&K,&L,&M,&STOCK,&DMD
```

Add the code

```
◇        ⎕PUTSTRING ⎕(' HOW MANY YEARS DO YOU WANT TO SIMULATE FOR?')
◇        ⎕GETLIST   ⎕&J
◇        ⎕PUTSTRING ⎕('⎕⎕')
◇        ⎕PUTSTRING ⎕(' NEXT WE NEED THE SUPPLY AMOUNTS TO CONSIDER IN')
◇        ⎕PUTSTRING ⎕(' THE SIMULATION. INPUT THE BEGINNING AMOUNT,')
◇        ⎕PUTSTRING ⎕(' THE END AMOUNT, AND THE INCREMENT SEPARATED BY')
◇        ⎕PUTSTRING ⎕(' BLANKS - NOT COMMAS!!')
◇        ⎕PUTSTRING ⎕('⎕⎕')
◇        ⎕GETLIST   ⎕&K,&L,&M
```

Replace

```
◇        ⎕DO         ⎕&I=90,150,5
```

with

```
◇        ⎕DO         ⎕K,&L,&M
```

and

```
◇        ⎕GENERATE   ⎕200
```

with

```
◇        ⎕GENERATE   ⎕&J
```

and change

```
◇        ⎕LET        ⎕&RESULT=&PROFIT/200.
```

to

```
◇        ⎕LET        ⎕&RESULT=&PROFIT/&J
```

THE IF, GOTO, AND HERE STATEMENTS IN GPSS/H

It is possible to have IF statements in GPSS/H. These can be used in the control statements to check on the input data or after the program has executed to prompt the user to rerun the program, often with different data. The form of the IF statement is

```
◇LABEL  □IF       □A
◇       □-----
◇       □-----
◇       □ENDIF
```

where operand *A* is a condition. Conditions are logical statements such as the following:

```
&I+&J<=&K
&KLM-&MMM=0
&YES'E'1
```

If the condition in operand *A* is true, the group of statements after the IF statement are executed.

It is also possible to have ELSEIF and ELSE statements contained between the IF and ENDIF statements, such as is typical in other programming languages. Their form is

```
◇LABEL  □IF       □A
◇       □-----
◇       □-----
◇       □ELSEIF   □A
◇       □-----
◇       □-----
◇       □-----
◇       □ELSE     □A
◇       □-----
◇       □-----
◇       □ENDIF
```

In this code, the three *A* operands would be different conditions of interest, for example,

```
◇       □IF       □&I'E'2
◇       □LET      □&J=0
◇       □ELSEIF   □&I'E'3
◇       □LET      □&J=1
◇       □ELSE
◇       □LET      □&J=2
◇       □ENDIF
```

Often GOTO is used with the IF statement. The form is simply

```
◇       □GOTO     □A
```

where the operand *A* is the label of the block to which the transaction is to be transferred. For example, you might have

```
◇       □PUTSTRING  □(' THE DATA YOU TYPED IN IS AS FOLLOWS')
◇       □PUTPIC     □LINES=3,&I,&J,&SPEED
            THE VALUE OF I  =  ***
            THE VALUE OF J  =  ***
            THE SPEED IS       ***.**
◇       □PUTSTRING  □('□□')
◇       □PUTSTRING  □(' ARE YOU HAPPY WITH THESE?')
```

```
◇        □PUTSTRING □(' RESPOND: 1 = YES; 0 = NO')
◇        □GETLIST   □&L
◇        □IF        □&L'E'0
◇        □GOTO      □TRYAGAIN
◇        □ENDIF
```

The effect of the code in the example is to prompt the user to examine the data already input. If something in the data is not satisfactory, control returns to where the data items were originally input.

It is possible to have control return to a target of the GOTO that is a dummy statement known as the HERE statement. The form of this is

```
◇LABEL    □HERE
```

Thus, in the previous example, there might have been the following statement:

```
◇TRYAGAIN□HERE
```

The HERE statement is analogous to the Fortran CONTINUE statement.

Example 23.3

Consider the following code:

```
◇        □PUTSTRING □('□')
◇        □PUTSTRING □(' DO YOU WANT TO DO ANOTHER SIMULATION? (Y/N)')
◇        □PUTSTRING □('□')
◇        □GETLIST   □&ANS
◇        □IF        □(&ANS'E''Y')OR(&ANS'E''y'')
◇        □CLEAR
◇        □GOTO      □AGAIN
◇        □ENDDO
◇        □PUTSTRING □(' SIMULATION OVER')
```

What will this code do?

Solution

The ampervariable &ANS needs to be defined as a character ampervariable of length 1. The effect of the above code is to prompt the user to redo the example. If the response is Y (or y), the program clears the previous results, and control transfers to the statement with the label AGAIN.

Example 23.4

Consider the following code:

```
◇        □INTEGER   □&I
◇        □DO        □&I=1,25
◇        □PUTSTRING □('□')
◇        □ENDDO
```

What will this code do?

Solution

The computer screen holds 25 lines of text per screen. The effect of the above is to put 25 blank lines on the screen. This code, therefore, gives a clear screen.

EXERCISES, CHAPTER 23

1. A program is to prompt the user to input the number of trucks to use in the simulation. This number cannot be 0 or less and also cannot be more than 10. Give the code to check that the input number of trucks is correct.
2. A program is to be run 4 times for storages of TUGS from 3 to 6. Show how this situation can be simulated with an appropriate DO loop.
3. The user is to be prompted after a program is run as to whether it should be run again. If the answer is "Yes," the user is to be prompted as to whether all new data are to be input or the only change is to be the number of trucks.
4. A simple barbershop example has customers arriving every $A \pm B$ minutes; haircuts take $C \pm D$ minutes. Write a program to simulate such a system with all data as input variables, including the length (i.e., number of time units) of the simulation.
5. Add the code to Exercise 23.4 to prompt the user to run the program again.

SOLUTIONS, CHAPTER 23

1.

```
PUTSTRING    ('  HOW MANY TRUCKS TO HAVE IN THE SIMULATION?')
PUTSTRING    ('  ')
GETLIST      &TRUCK
IF           (&TRUCK'LE'0)OR(&TRUCK'G'10)
PUTSTRING    ('  THE NUMBER OF TRUCKS MUST BE BETWEEN')
PUTSTRING    ('  1 AND 10.  GO BACK AND INPUT THE')
PUTSTRING    ('  CORRECT NUMBER.')
GOTO         BACK
ENDIF
```

2.

```
DO           &I=3,6
CLEAR
INITIAL      S(TUGS),&I
START        1
PUTPIC       ..................
.................................
ENDDO
```

3.

```
PUTSTRING    ('  ')
PUTSTRING    ('  RUN AGAIN? (Y/N)')
PUTSTRING    ('  ')
GETLIST      &ANS
IF           (&ANS'E''Y)OR(&ANS'E''y')
PUTSTRING    (' WITH ONLY THE TRUCKS CHANGE? (Y/N)')
PUTSTRING    ('  ')
GETLIST      &ANS
```

```
IF          (&ANS'E''Y')OR(&ANS'E''y')
GOTO        TRUCKS
ENDIF
GOTO        ALLOVER
ENDIF
```

Note: For better readable of the code, it is a good idea to indent as is done here. However, care must be taken not to exceed column 25. If this happens, the OPERCOL statement will be needed.

4. The program is as follows:

```
            SIMULATE
            INTEGER     &I,&MINUTES
            REAL        &MEAN1,&SPREAD1,&MEAN2,&SPREAD2,&SHIFTS
            CHAR*1      &ANS
            DO          &I=1,25
            PUTSTRING   ('  ')
            ENDDO
AGAIN       PUTSTRING   (' FOR THE ARRIVAL OF PEOPLE, INPUT THE')
            PUTSTRING   (' MEAN AND THE SPREAD.  SEPARATE THESE')
            PUTSTRING   (' WITH A SPACE, NOT A COMMA')
            PUTSTRING   ('  ')
            GETLIST     &MEAN1,&SPREAD1
            PUTSTRING   ('  ')
            PUTSTRING   (' FOR THE SERVICE RETE, INPUT THE')
            PUTSTRING   (' MEAN AND THE SPREAD.  SEPARATE THESE')
            PUTSTRING   (' WITH A SPACE, NOT A COMMA')
            PUTSTRING   ('  ')
            GETLIST     &MEAN2,&SPREAD2
            PUTSTRING   ('  ')
            PUTSTRING   ('  INPUT THE NUMBER OF MINUTES IN A SHIFT')
            PUTSTRING   ('  ')
            GETLIST     &MINUTES
            PUTSTRING   ('  ')
            PUTSTRING   ('  HOW MANY SHIFTS TO SIMULATE FOR?')
            PUTSTRING   ('  ')
            GETLIST     &SHIFTS
          . PUTSTRING   ('  ')
            PUTPIC      LINES=4,&MEAN1,&SPREAD1,&MEAN2,&SPREAD2,_
                        &MINUTES,&SHIFTS
THE INPUT DATA IS AS FOLLOWS:
FOR ARRIVALS:   MEAN   ****.**    SPREAD                  ****.**
FOR SERVICE :   MEAN   ****.**    SPREAD                  ****.**
MINUTES OF WORK/SHIFT   ***       SHIFTS TO SIMULATE FOR ****
            PUTSTRING   ('  ')
            PUTSTRING   ('  IS THE DATA ALL RIGHT? (Y/N)')
            PUTSTRING   ('  ')
            GETLIST     &ANS
            IF          (&ANS'E''N')OR(&ANS'E''n')
            GOTO        AGAIN
            ENDIF
            PUTSTRING   ('  SIMULATION IN PROGRESS...')
            GENERATE    &MEAN1,&SPREAD1
            QUEUE       WAIT
            SEIZE       BARBER
```

```
          DEPART        WAIT
          ADVANCE       &MEAN2,&SPREAD2
          RELEASE       BARBER
          TERMINATE
          GENERATE      &MINUTES*&SHIFTS
          TERMINATE     1
          START         1
          DO            &I=1,25
          PUTSTRING     ('  ')
          ENDDO
          PUTPIC        LINES=3,FC(BARBER)/&SHIFTS,FR(BARBER)/10.
RESULTS OF SIMULATION FOR BARBER SHOP
HAIRCUTS PER DAY                ****.**
UTIL. OF BARBER                 ****.**
          END
```

5. The code to add is as follows:

```
          PUTSTRING      ('  ')
          PUTSTRING      (' RUN THE PROGRAM AGAIN? (Y/N)')
          PUTSTRING      ('  ')
          GETLIST        &ANSWER
          IF             (&ANSWER'E''Y')OR(&ANSWER'E''y')
          GOTO           AGAIN
          ENDIF
          PUTSTRING       ('  ')
          PUTSTRING       ('  SIMULATION OVER....')
```

The ampervariable &ANSWER needs to be specified by means of the statement CHAR*1.

REFERENCE

Cooper, L., U.N. Bhat, and L.J. LeBlanc. 1977. *Introduction to Operations Research Models.* Philadelphia: W.B. Saunders Company.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 24

The *SELECT* and *COUNT* Blocks

THE SELECT BLOCK

Each of the features of a GPSS/H program–including the blocks, the queues, the facilities, the savevalues, etc.–is known as an entity class. Each entity class has SNAs associated with it, and these are also known as entities. GPSS/H has several powerful blocks that can be used to search several of these entities (or SNAs) and test their current values for a specific condition when a transaction enters the block. When this condition is met, a record is placed in one of the transaction's parameters. Such scans are common in real-life situations. For example, a person entering a bank that has individual queues at each teller will scan these queues, determine which is the shortest, and then join the queue that is the shortest. The block to both scan and test at the same time is the SELECT block. When a transaction enters this block, a scan of selected entity members is made. When one of these scanned members is found that satisfies some stated condition, the scan is terminated. Other examples of where such a block might be used to model real-life situations are as follows:

1. Ore is delivered to 4 silos. The one that has the least ore in it is to be filled first. Alternatively, ore is to be taken out of the 4 silos starting with the one with the most ore in it.
2. A part comes along an assembly line to a point at which 3 machines can work on it. The part will be sent to the first machine that is not in use. If all 3 machines are in use, the part is sent to another section of the plant.

In order to do this scanning and testing, the SELECT block needs to know what entities to scan, what the test is, where to put the result of the scan, and what to do if the scan is not successful. As a result, the SELECT block can have up to 6 operands. As such, it is the most complicated block encountered since the GENERATE block. One general form of the SELECT block is

```
◇LABEL    □SELECT□R   □A,B,C,D,E,F
```

where *R* is a *relational operator,* which, as we saw in Chapter 16 on the TEST block, is one of the following:

G	greater than
GE	greater than or equal to
E	equal to
NE	not equal to
LE	less than or equal to
L	less than

Operand *A* is the parameter number (and parameter type unless you are only using halfword parameters) into which the first entity number that satisfies the test is to be placed. Thus, if queues numbered from 5 to 8 are scanned and queue number 7 satisfies the condition, the number 7 is placed in the transaction's parameter given by *A*. Operand *B* is the smallest number of the entities to be scanned and may be any SNA. Operand *C* is the largest number of the entities to be scanned and may be any SNA. Operand *D* is the specific condition sought in the test. Operand *E* is the family name of the SNA to be used in the scan. This might be F, PH, S, Q, etc. Operand *F* (optional) is the block label of the block to which the transaction is to be transferred if the scan is not successful.

Note that *R* follows the operation SELECT plus exactly 1 blank space. The SELECT block is best understood by considering examples of its use, as follows:

```
◇(a)    □SELECT□E  □3PF,1,5,2,Q
```

When a transaction enters this block, a scan will be made of queues (Q) 1 to 5. These will be tested to see if any has a queue length of 2. If so, the number of that queue will be placed in the transaction's third fullword parameter. If none of the queues 1 to 5 has a queue length equal to 2, the transaction moves to the next sequential block. If the queue lengths at all the queues are 0 except at queues 4 and 5 where they are both 2, the number 4 is placed in the transaction's second fullword parameter.

```
◇(b)    □SELECT□G  □5PH,3,7,250,FR
```

A scan is made of the utilization of facilities (FR) 3 to 7, starting with facility 3 and going up to facility 7. Once a facility is found that has a fractional utilization greater than 250 permil (recall that the fractional utilization FR is always expressed in parts per thousand), the scan is stopped, and the facility number is placed in the transaction's fifth halfword parameter. If no facility from 3 to 7 has a fractional utilization greater than 250 permil, the transaction moves to the next sequential block.

```
◇(c)    □SELECT□LE □10PH,1,3,1,R,AWAY
```

Storages 1 to 3 are scanned. If one has a remaining storage (R) of less than or equal to 1, that storage's number is copied into the transaction's tenth halfword parameter. If no storage in the scan satisfies the criterion, the transaction is routed to the block with the label AWAY.

```
◇(d)      □SELECT□NE □4PH,7,12,0,PH
```

The scan is of the transaction's halfword parameters (PH) 7 to 12. Once one of these is found to be not equal to 0, the number of that halfword parameter is copied into the transaction's fourth halfword parameter.

```
◇(e)      □SELECT□E  □3PH,PH3,PH4,3,Q,DOWN
```

The result of the scan will be placed in the transaction's third halfword parameter. The queues (Q) to be scanned will depend on the values of the transaction's third and fourth halfword parameters. Suppose these values are 5 and 9, respectively. Then queues 5 through 9 are scanned, and if any has a length equal to 3, the number of that queue is placed in the transaction's third halfword parameter. If none of the queues has a length equal to 3, the transaction is routed to the block with the label DOWN.

Example 24.1

Parts come along an assembly line to a point where they are to be worked on by 1 of 3 identical machines. The parts arrive at a rate of 1 every 8 ± 4.5 minutes. The machines finish a part in 20 ± 6 minutes. If a machine is free, the part is worked on by that machine, but if all 3 machines are busy, the part is sent away to another part of the factory. Determine the utilization of the 3 machines and how many parts are sent away for 20 shifts of 480 minutes each.

Solution

```
◇         □SIMULATE
◇         □INTEGER   □&I
◇         □DO        □&I=1,25
◇         □PUTSTRING □('□')
◇         □ENDDO
◇TIMES    □GENERATE  □8,4.5
◇         □SELECT□E  □2,1,3,0,F,AWAY □(NOTE: NO PARAMETER TYPE: SEE
*FOLLOWING EXPLANATION)
◇         □SEIZE     □PH2
◇         □ADVANCE   □20,5
◇         □RELEASE   □PH2
◇         □TERMINATE
◇AWAY     □TERMINATE
◇         □GENERATE  □480*20
◇         □TERMINATE □1
◇         □START     □1
◇         □DO        □&I=1,25
◇         □PUTSTRING □('□')
◇         □ENDDO
◇         □PUTPIC    □LINES=5,N(TIMES),N(AWAY),FR1/10.,FR2/10.,FR3/10.
  THE NUMBER OF PARTS ENTERING THE SYSTEM  =  ****
  THE NUMBER SENT TO OTHER MACHINES        =  ****
  UTILIZATION OF MACHINE 1                 =   ***.**%
  UTILIZATION OF MACHINE 2                 =   ***.**%
  UTILIZATION OF MACHINE 3                 =   ***.**%
◇         □END
```

The output from the program is as follows:

```
THE NUMBER OF PARTS ENTERING THE SYSTEM   = 1198
THE NUMBER SENT TO OTHER MACHINES         =  151
UTILIZATION OF MACHINE 1                  =   81.56%
UTILIZATION OF MACHINE 2                  =   75.12%
UTILIZATION OF MACHINE 3                  =   62.48%
```

For the 20 shifts, 1198 parts came to the 3 machines. Of these total parts, 151 were turned away because all 3 machines were busy. The utilization of the 3 machines was 81.56% for machine 1, 75.12% for machine 2, and 62.48% for machine 3. The reason for the decrease in utilization is because, when a new part arrives at the machines, the scan checks to see if any of the 3 machines is free. The scan stops when one machine is found to be free, and the scanning always starts at machine 1.

This problem also could have been solved by using the TRANSFER ALL block.

Notice that, in Example 24.1, the parameter type was not specified. As long as you have only halfword parameters, this shortcut is acceptable. But if the transaction had other types of parameters, it is necessary to specify the type in the SELECT block. For example,

```
◇        GENERATE   20,6.5,,,,2PH,10PF
◇        -----      -----
◇        -----      -----
◇        SELECT E   2,1,5,1,Q
```

would give an error. The correct form of the SELECT block would have to be

```
◇        SELECT E   2PH,1,5,1,Q   (or 2PF)
```

THE COUNT BLOCK

The COUNT block resembles the SELECT block in that, when a transaction enters, it triggers a scan of specified entities. The result of the scan is placed in a specified parameter. In the case of the SELECT block, once the scan finds an entity to satisfy the given test, the scan is over. The COUNT block counts the number of the entities that satisfy the criteria and places the number that satisfy the test criteria into the transaction's specified parameter. One general form of the COUNT block is

```
◇        COUNT R    A,B,C,D,E
```

R is one of the relational operators used in the SELECT block. Thus, it must be one of the following: G, GE, E, NE, L, or LE. *A, B, C, D,* and *E* have the same meanings as the operands in the SELECT block. Since the number of the entities in a COUNT block will always be a number greater than or equal to 0, there cannot be any *F* operand.

Some examples of the COUNT block are as follows:

```
◇(a)     COUNT E    1,1,4,0,Q
◇(b)     COUNT G    3,3,6,250,FR
```

In (a), a count is made of all the queues from 1 to 4 that have lengths equal to 0. The result of this count is placed in the transaction's first halfword parameter. In (b), a count is made of the facilities 3 to 6 that have a utilization greater than 250 permil. The total number of such facilities is placed in the transaction's third (halfword) parameter.

OTHER FORMS OF THE SELECT AND COUNT BLOCKS

The SELECT Block in MIN or MAX Mode

The SELECT block can be used to scan a group of entities and determine which has a maximum (or minimum) value. The general form of this block is either

```
◇          □SELECT□MAX□A,B,C,,E
```

or

```
◇          □SELECT□MIN□A,B,C,,E
```

The "word" MIN (or MAX) is called an auxiliary operator and appears in the same position as a relational operator, i.e., separated from the operation SELECT by exactly 1 space (as shown by the open box in the code line). The *A, B, C,* and *E* operands are the same as for the regular SELECT block. There is no *D* operand, but the two commas are necessary.

When a transaction enters this form of SELECT block, a scan is made of the entities specified by the *B* and *C* operands. The processor selects the minimum (or maximum) from the desired entity class and places the number of that entity into the transaction's parameter number and type as specified by the *A* operand.

```
◇(a)       □SELECT□MIN□3PH,1,4,,FR
◇(b)       □SELECT□MAX□1PH,3,7,,Q
◇(c)       □SELECT□MIN□5PH,1,3,,R
```

In (a), facilities 1 to 4 are scanned, and the processor places the number of the facility in this group that has the least fractional utilization into the transaction's halfword parameter 3. Thus, if facility 1 had FR = 350, facility 2 had FR = 599, facility 3 had FR = 500, and facility 4 had FR = 222, the processor would place the number 4 in the transaction's third halfword parameter. In (b), queues 3 to 7 are scanned. The queue that has the greatest length is determined, and its number is placed in the transaction's halfword parameter 1. In case of a tie, i.e., if both queue 4 and queue 7 had equal lengths of 5 and this value was the maximum, the number 4 is placed in halfword parameter 1. This approach of "the first wins in the event of a tie" is also the case if SELECT MIN is used. In (c), storages from 1 to 3 are scanned, and the one with the least remaining storage is placed in the transaction's halfword parameter 5.

Use with Logic Switches

A SELECT or COUNT block can be used with logic switches. The general form is

```
◇          □SELECT□LS □A,B,C,,,F
```

LR can be used in place of LS. Operands *D* and *E* are omitted and *F* is optional. If *F* is used, the commas are required; otherwise, they can be omitted.

The scan is made of logic switches from *B* (minimum value) to *C* (maximum value). As soon as the processor encounters one that satisfies the test (either LS for set or LR for reset), the scan is finished, and the number of the logic switch is placed in the transaction's parameter given by operand *A*. The *F* operand is a block label where the transaction is routed if no logic switch satisfies the scan criteria. If the scan fails and there is no *F* operand, the transaction is routed to the next sequential block and nothing else is done.

The COUNT block in this mode is

```
◇          COUNT LS  A,B,C
```

LR could be used in place of LS. The only difference in form from that of the SELECT block is that operand *F* is never used. In the COUNT block, a scan is made of the logic switches from *B* to *C*, and the number of switches in set (or reset) condition is placed in the parameter given by operand *A*. For example, suppose a transaction entered the following two sequential blocks in a program:

```
◇          SELECT LS 3PH,2,6
◇          COUNT LR  4PH,2,6
```

And suppose that the logic switches 2 through 6 were as follows:

LS2 is reset
LS3 is reset
LS4 is set
LS5 is reset
LS6 is set

The transaction would have a 4 in halfword parameter 3 and a 3 in halfword parameter 4.

Use with Facilities and Storages

The form of the SELECT block is

```
◇          SELECT(x)A,B,C,,,F
```

where *x* is an auxiliary operator. It can be one of the following:

Auxiliary Operator	Meaning
FU	facility in use
FNU	facility not in use
SE	storage empty
SNE	storage not empty
SF	storage full
SNF	storage not full

There are no *D* or *E* operands, and the *F* operand is optional. The meaning of the operands is the same as before.

The general form of the COUNT block in this mode is

```
◇          COUNT (x)  A,B,C
```

A scan is made of the entities and the total number of them satisfying the criteria is placed in the transaction's parameter as specified by the *A* operand. Several examples of these blocks are

```
◇(a)       SELECT FNU 1PH,3,6
◇(b)       COUNT FNU  2PH,3,6
◇(c)       SELECT SE  3PH,1,7,,,AWAY
◇(d)       COUNT SF   4PH,1,7
```

In (a), the scan is of facilities 3 to 6 in ascending order. Once one is found to be not in use, the scan is finished and the facility number is placed in the transaction's halfword parameter 1. In (b), a count is made of facilities 3 to 6 that are not in use. The total is placed in halfword parameter 2. In (c), a scan of storages 1 to 7 is made to determine if any are empty. The number of the first one to satisfy the criterion is placed in halfword parameter 3, and the scan is stopped. If no storage is found to satisfy the criterion, the transaction is sent to the block with the label AWAY. In (d), a count is made of the storages 1 to 7 that are found to be full. The total is placed in halfword parameter 4.

Example 24.2

Did you ever wonder why banks, post offices, airline ticket agent counters, and other places where multiple servers are used now have customers wait in an individual queue rather than forming separate queues at each teller or agent? The single-queue system is known as a "quickline" system. This example will illustrate why a quickline system is better than individual queues.

Suppose customers arrive in a store that has 6 clerks behind desks to serve the customers. Customers come in and first see if any clerk is free. If so, the customer will go to that desk. If all the clerks are busy, the person will go to the back of the shortest queue. Once at a desk, no queue jumping is allowed.

Customers arrive in a Poisson stream, with an interarrival time of 10 seconds. A given customer will transact any one of 4 types of business. The amount of time each type of business requires and the relative percentage of customers transacting each type of business are as follows:

	Time Taken (seconds)		
Business	**Mean**	**Standard Deviation**	**Relative Percent**
Type 1	25	4	28
Type 2	32	5	17
Type 3	60	10	30
Type 4	80	10	25

Model this system for 10 days with both individual queues and the quickline approach. The store works 10 hours straight.

Solution

The program to model the quickline is quite easy to write by using the clerks as storages. The code is as follows:

```
◇        □SIMULATE
◇        □INTEGER   □&I
◇        □REAL      □&AVGQ,&AVGUTIL,&AVGIN,&AVGTIM
◇MYOUT   □FILEDEF   □'BANK1.OUT'
◇MEAN    □FUNCTION  □RN1,D4
.28,25/.4,32/.7,60/1,80
◇STDDEV  □FUNCTION  □PH1,D4
25,4/32,6/60,10/80,10
◇        □STORAGE   □S(TELLER),6
◇        □GENERATE  □RVEXPO(1,10) □CUSTOMERS ARRIVE
◇        □QUEUE     □WAIT
◇        □ASSIGN    □1,FN(MEAN),PH
◇        □ASSIGN    □2,FN(STDDEV),PH
◇        □QUEUE     □QLINE
◇        □ENTER     □TELLER
◇        □DEPART    □QLINE
◇        □ADVANCE   □RVNORM(1,PH1,PH2)
◇        □LEAVE     □TELLER
◇        □DEPART    □WAIT
◇        □TERMINATE
◇        □GENERATE  □3600*10      □SIMULATE FOR 10 DAYS
◇        □TERMINATE □1
◇        □DO        □&I=1,10
◇        □CLEAR
◇        □START     □1
◇        □PUTPIC    □LINES=7,FILE=MYOUT,&I,QM(QLINE)QX(QLINE)_
               QA(QLINE)SR(TELLER)/10,QA(WAIT),QT(WAIT)
 SIMULATION RESULTS FOR DAY    ****
 MAXIMUM QUEUE WAS             ****
 AVERAGE TIME IN QUEUE          ***.**
 AVERAGE NUMBER IN QUEUE        ***.**
 UTILIZATION OF TELLERS         ***.**%
 AVERAGE PEOPLE IN SHOP         ***.**
 AVERAGE TIME IN THE BANK       ***.**
◇         □LET       □&AVGQ=&AVGQ+QA(QLINE)
◇         □LET       □&AVGUTIL=&AVGUTIL+SR(TELLER)
◇         □LET       □&AVGIN=&AVGIN+QA(WAIT)
◇         □LET       □&AVGTIM=&AVGTIM+QT(WAIT)
◇         □ENDDO
◇         □PUTPIC    □LINES=4,FILE=MYOUT,&AVGQ/10.,(&AVGUTIL/10.)/10.,_
               &AVGIN/10.,&AVGTIM/10.
 AVERAGE NUMBER IN QUEUE FOR THE 10 DAYS WAS          ***.**
 AVERAGE UTILIZATION OF THE TELLERS FOR THE 10 DAYS WAS ***.**%
 AVERAGE NUMBER OF PEOPLE IN THE BANK WAS             ***.**
 AVERAGE TIME IN THE BANK FOR EACH CUSTOMER WAS       ***.**
◇         □END
```

The output is sent to the file BANK1.OUT. The part of the output that is of concern is the average length of time each person spends in the bank.

```
AVERAGE NUMBER IN QUEUE FOR THE 10 DAYS WAS            2.86
AVERAGE UTILIZATION OF THE TELLERS FOR THE 10 DAYS WAS 88.19%
AVERAGE NUMBER OF PEOPLE IN THE BANK WAS               8.05
AVERAGE TIME IN THE BANK FOR EACH CUSTOMER WAS        81.31
```

The important statistic is that each person could expect to be in the bank for 81.31 seconds. Next, the program is given for the case of individual queues for each teller. As each person enters the bank, he or she scans the tellers to see if any is free. If so, the person goes to that teller. If no teller is free, the person scans the queues and goes to the teller who has the minimum queue length. The program code is as follows:

```
◇         SIMULATE
◇         INTEGER    &I
◇         REAL       &INBANK,&TIMBANK
◇MYOUT    FILEDEF    'BANK2.OUT'
◇WAIT     EQU        10,Q
◇MEAN     FUNCTION   RN1,D4
.28,25/.4,32/.7,60/1,80
◇STDDEV   FUNCTION   PH1,D4
25,4/32,6/60,10/80,10
◇         GENERATE   RVEXPO(1,10)
◇         ASSIGN     1,FN(MEAN),PH
◇         ASSIGN     2,FN(STDDEV),PH
◇         QUEUE      WAIT
◇         SELECT E   3PH,1,6,0,F,AWAY
◇BACK     QUEUE      PH3
◇         SEIZE      PH3
◇         DEPART     PH3
◇         ADVANCE    RVNORM(1,PH1,PH2)
◇         RELEASE    PH3
◇         DEPART     WAIT
◇         TERMINATE
◇AWAY     SELECT MIN 3PH,1,6,,Q
◇         TRANSFER   ,BACK
◇         GENERATE   3600*10
◇         TERMINATE  1
◇         DO         &I=1,10
◇         CLEAR
◇         START      1
◇         PUTPIC     LINES=6,FILE=MYOUT,&I,QM(WAIT),QX(WAIT),_
                     QA(WAIT),FR1/10,FR2/10,FR3/10,FR4/10,FR5/10,_FR6/10.
 SIMULATION RESULTS FOR DAY    ****
 MAXIMUM QUEUE WAS             ****
 AVERAGE TIME IN QUEUE          ***.**
 AVERAGE NUMBER IN QUEUE        ***.**
 UTIL. TELLER 1 ***.**%  UTIL. TELLER 2 ***.**%  UTIL. TELLER 3 ***.**%
 UTIL. TELLER 4 ***.**%  UTIL. TELLER 5 ***.**%  UTIL. TELLER 6 ***.**%
◇         LET        &INBANK=&INBANK+QA(WAIT)
◇         LET        &TIMBANK=&TIMBANK+QT(WAIT)
◇         ENDDO
◇         PUTPIC     FILE=MYOUT,LINES=2,&INBANK/10.,&TIMBANK/10.
 AVERAGE NUMBER IN BANK FOR THE 10 DAYS WAS ***.**
 AVERAGE TIME IN BANK FOR THE 10 DAYS WAS   ***.**
◇         END
```

Again, the important statistic is the length of time each person can expect to be in the bank. The output from the program is as follows:

```
AVERAGE NUMBER IN BANK FOR THE 10 DAYS WAS      9.07
AVERAGE TIME IN BANK FOR THE 10 DAYS WAS       90.55
```

As can be seen, the quickline is faster than individual queues. For the bank using the quickline, people were in the bank an average of 81.31 seconds, and there were an average of 8.05 people in the bank at any one time. For the bank using individual queues, people were in the bank for 90.55 seconds, and there were an average of 9.07 people in the bank.

Notice that the EQU compiler directive was necessary in the program for individual queues since the various queues could not be identified during compiling. If the EQU compiler directive had not been used, the queue WAIT would have been queue 1. This would be the same queue as the people in line before teller 1 and, thus, would give incorrect statistics.

EXERCISES, CHAPTER 24

1. Explain what happens when a transaction enters each of the following SELECT blocks:

```
◇(a)     □SELECT□E  □3PH,1,5,3,Q
◇(b)     □SELECT□G  □1PH,2,7,900,FR
◇(c)     □SELECT□NE □5PH,1,10,1,F
◇(d)     □SELECT□E  □3PH,4,12,0,PH,AWAY
◇(e)     □SELECT□L  □1PH,PH2,PH3,100,DDDD
◇(f)     □COUNT□E   □1PH,1,5,0,XH
◇(g)     □COUNT□E   □10PH,5,20,1,Q
```

2. Explain what happens when a transaction enters each of the following SELECT blocks.

```
◇(a)     □SELECT□MAX□2PF,1,10,,Q
◇(b)     □SELECT□MIN□3PH,1,5,,XF
◇(c)     □SELECT□MAX□4PH,PH1,PH2,,FR
```

3. Parts come along for repairs every 10 ± 4.5 minutes. There are 3 identical machines to do these repairs. Each takes 26 ± 5 minutes to repair an item. Write a program to simulate the situation in which a part is assigned to the machine that has worked the least amount. Contrast the results with those for a situation in which the part always goes to the first machine if it is free and only goes to the second if the first is busy or goes to the third only if the first and second machines are busy.

SOLUTIONS, CHAPTER 24

1. a. A scan is made of the queues from 1 to 5 to determine if any has a length of 3. If so, the number of this is placed in the transaction's 3rd halfword parameter.

   b. A scan is made of the facilities from 2 to 7. If any has a fractional utilization greater than 900, the number of this facility is placed in the transaction's 1st halfword parameter.

c. A scan is made of the facilities from 1 to 10. If any is not currently being seized, a the number of this facility is placed in the transaction's 5th halfword parameter.

d. A scan is made of the parameters from 4 to 12 to see if any has the value 0. If so, the number of this parameter is placed in the transaction's 3rd halfword parameter. If no parameter in the scan has the value zero, control transfers to the block with the label AWAY.

e. A scan is made of the parameters which are given by the values in parameter 2 and 3. The test is for any parameter in this range with the value less than 100. If so, the number of this parameter is placed in the transactions's first halfword parameter.

f. Halfword savevalues from 1 to 5 are scanned. The number of these having the value of 0 is placed in the transaction's 1st halfword parameter.

g. The queues from 5 to 20 are scanned. The number having a length of 1 is placed in the transaction's 10th halfword parameter.

2. a. The queues from 1 to 10 are scanned. The number of the largest one is placed in the transaction's 2nd fullword parameter.

b. The full word savevalues from 1 to 5 are scanned. The number of the one having the lease value is placed in the transaction's 3rd halfword parameter.

c. Facilities are scanned to determine which has the greatest utilization. The number of these facilities to be scanned depends on what values are stored in halfword parameters PH1 and PH2.

3. The program to simulate this is:

```
          SIMULATE
          GENERATE     10,4.5
          QUEUE        WAIT
          SELECT MIN   1PH,1,3,,FR
          SEIZE        PH1
          DEPART       WAIT
          ADVANCE      26,5
          RELEASE      PH1
          TERMINATE
          GENERATE    480*10
          TERMINATE   1
          START       1
          PUTPIC      LINES=3,FR1/10.,FR2/10,FR3/10.
UTIL. OF FIRST MACHINE    ***.**%
UTIL. OF SECOND MACHINE   ***.**%
UTIL. OF THIRD MACHINE    ***.**%
          END
```

The results of the simulation are:

```
UTIL. OF FIRST MACHINE    84.34%
UTIL. OF SECOND MACHINE   84.65%
UTIL. OF THIRD MACHINE    84.49%
```

The program for the second part is:

```
          SIMULATE
          GENERATE    10,4.5
          QUEUE       WAIT
          TRANSFER    ALL,FIRST,LAST,5
 FIRST    SEIZE       MACH1
          DEPART      WAIT
          ADVANCE     26,5
          RELEASE     MACH1
          TERMINATE
          SEIZE       MACH2
          DEPART      WAIT
          ADVANCE     26,5
          RELEASE     MACH2
          TERMINATE
 LAST     SEIZE       MACH3
          DEPART      WAIT
          ADVANCE     26,5
          RELEASE     MACH3
          TERMINATE

          GENERATE   480*10
          TERMINATE  1
          START      1
          PUTPIC     LINES=3,FR(MACH1)/10.,FR(MACH2)/10,FR(MACH3)/10.
UTIL. OF FIRST MACHINE   ***.**%
UTIL. OF SECOND MACHINE  ***.**%
UTIL. OF THIRD MACHINE   ***.**%
          END
```

Now, the results of the simulation are:

```
UTIL. OF FIRST MACHINE    89.06%
UTIL. OF SECOND MACHINE   83.06%
UTIL. OF THIRD MACHINE    84.03%
```

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 25

Matrices

GPSS/H allows for the use of matrices in a manner similar to that found in other computer languages. The matrix has to be first defined. This means specifying the number of rows, the number of columns, the type of elements that will be in the matrix, and the name (or number) of the matrix. A savevalue can be considered as a linear array. A matrix can be considered as a savevalue with two or more dimensions. The general form of the statement that specifies the matrix is

```
◇LABEL  ⎕MATRIX   ⎕A,B,C
```

The label is a name (or number) that follows the usual rules for naming savevalues. Operand *A* designates the type of matrix, operand *B* gives the number of rows, and operand *C* gives the number of columns. There are 4 types of matrices in GPSS/H:

Matrix Type Specification	Kind of Matrix Savevalue
MX (M can be omitted)	fullword
MH (M can be omitted)	halfword
MB	byte-word
ML	floating-point

The size and integer nature of the elements used in the various types of matrices are the same as for regular savevalues. Some examples of matrix definitions are

```
◇FIRST  ⎕MATRIX   ⎕MH,1,3
◇4      ⎕MATRIX   ⎕ML,2,10
◇LAST   ⎕MATRIX   ⎕MB,3,3
◇OTHER  ⎕MATRIX   ⎕MX,5,4
```

Matrix FIRST is a halfword matrix having 1 row and 3 columns, i.e., it might be

(2 4 −3)

For this example, the values of the halfword matrix FIRST are as follows: (1,1) is 2; (1,2) is 4; and (1,3) is -3. Matrix 4 is a floating-point matrix with 2 rows and 10 columns. Matrix LAST is a byte-word matrix of size 3 by 3. Matrix OTHER is declared to be a fullword matrix with 5 rows and 4 columns. Note that it would have been all right (but not recommended, according to the best programming practice) to omit the M in MH and the M in MX:

```
◇FIRST   □MATRIX    □H,1,3
◇OTHER   □MATRIX    □X,5,4
```

GIVING INITIAL VALUES TO MATRICES

Once a matrix is defined via the matrix declaration statement, all its elements are set to zero. It is possible to have initial values assigned to the various elements by using the INITIAL statement, just as was done for ordinary savevalues. When initializing ordinary savevalues, two forms were possible. One was the older form that used the dollar sign, "$," and the other used left and right parentheses. Only the $ sign is allowed for initializing matrices. Thus, whereas for savevalues one can have

```
◇        □INITIAL   □XF$JILL,4
◇        □INITIAL   □XF$(TOMMY),-5/XH(SAM),10/XL$SALLY),12.345
```

When initializing matrices that are referenced by labels that are not numbers, the only form one can use is the $ sign. Thus, one might have

```
◇FIRST   □MATRIX    □MH,1,3
◇        □INITIAL   □MH$FIRST(1,1),2/MH$FIRST(1,2),-4
```

The elements of the 1 by 3 dimensioned halfword matrix would be

(2 -4 0)

If a matrix is referenced by a label that is a number, one need not use the $ approach, for example,

```
◇1       □MATRIX    □MH,2,2
◇        □INITIAL   □MH1(1,1),3/MH2(2,2).5
```

would be all right. Note that in the INITIAL statement, MH1 refers to the halfword matrix with the name 1 and MH2 refers to the halfword matrix with the name 2.

Example 25.1

Large and small trucks come to an inspection and refueling station. A large truck arrives every 8 ± 1 minutes, and a small truck arrives every 6.5 ± 2.2 minutes. The sequence for each truck as it passes through the station is as follows:

1. It takes 2.5 ± 1 minutes to position each truck. Only one truck can be positioned at a time.
2. A single worker then fuels a truck in 2.7 ± 1 minutes.
3. After fueling, a single worker checks gauges inside the cab. The gauge check takes 2.9 ± 1 minutes.

4. Next, two workers wash each truck in 6.5 ± 1.2 minutes.
5. A single worker checks and fills tires if needed. This process takes 3 ± 1 minutes.
6. There are two final checks, and these are done by two people. Their times vary depending on the type of truck and are as follows:

	2 Axles	3 Axles
Inspector 1	6.1 ± 2	6.4 ± 2
Inspector 2	6.9 ± 1	7.5 ± 1

Simulate for 1 week's operation to determine if there are any bottlenecks in the system.

Solution

The solution will utilize a matrix whose size is 2 by 2 and whose elements are the mean times.

```
◇         □SIMULATE
◇         □INTEGER    □&I
◇SERV     □MATRIX     □ML,2,2
◇         □INITIAL    □ML$SERV(1,1),6.1/ML$SERV(1,2),6.4
◇         □INITIAL    □ML$SERV(2,1),6.9/ML$SERV(2,2),7.5
◇         □STORAGE    □S(WORKERS),2
◇         □GENERATE   □8,1           □LARGE TRUCK COMES ALONG
◇         □ASSIGN     □1,1,PH        □CALL IT 1
◇         □TRANSFER   □,DDDD
◇         □GENERATE   □6.5,2.2       □SMALL TRUCK COMES ALONG
◇         □ASSIGN     □1,2,PH        □CALL IT 2
◇DDDD     □ADVANCE    □2.5,1         □POSITION IT
◇         □SEIZE      □WORK1
◇         □ADVANCE    □2.7,1
◇         □RELEASE    □WORK1
◇         □SEIZE      □WORK2
◇         □ADVANCE    □2.9,1
◇         □RELEASE    □WORK2
◇         □ENTER      □WORKERS
◇         □ADVANCE    □6.5,1.2
◇         □LEAVE      □WORKERS
◇         □SEIZE      □WORK3
◇         □ADVANCE    □3,1
◇         □RELEASE    □WORK3
◇         □TRANSFER   □BOTH,,OTHER
◇         □SEIZE      □INSP1
◇         □ADVANCE    □ML$SERV(1,PH1),2
◇         □RELEASE    □INSP1
◇         □TRANSFER   □,BACKIN
◇OTHER    □SEIZE      □INSP2
◇         □ADVANCE    □ML$SERV(2,PH1),1
◇         □RELEASE    □INSP2
◇BACKIN   □TEST□E     □PH1,1,TYPE2
◇TYPE1    □TERMINATE
◇TYPE2    □TERMINATE
```

```
◇        GENERATE   480*100
◇        TERMINATE  1
◇        START      1
◇        DO         &I=1,25
◇        PUTSTRING  (' ')
◇        ENDDO
◇        PUTPIC     LINES=8,FR(WORK1)/10.,FR(WORK2)/10.,_
                     SR(WORKERS)/10.,FR(WORK3)/10.,_
                     FR(INSP1)/10.,FR(INSP2)/10.,_
                     N(TYPE1)/10.,N(TYPE2)/10.
 UTIL. OF WORKER 1       ***.**%
 UTIL. OF WORKER 2       ***.**%
 UTIL. OF WORKERS        ***.**%
 UTIL. OF WORKER 3       ***.**%
 UTIL. OF INSP. 1        ***.**%
 UTIL. OF INSP. 2        ***.**%
 NUMBER OF LARGE TRUCKS/DAY   ***.**
 NUMBER OF SMALL TRUCKS/DAY   ***.**
◇        END
```

The output from the program is as follows:

```
UTIL. OF WORKER 1      75.39%
UTIL. OF WORKER 2      80.92%
UTIL. OF WORKERS       90.81%
UTIL. OF WORKER 3      83.63%
UTIL. OF INSP. 1       92.45%
UTIL. OF INSP. 2       94.67%
NUMBER OF LARGE TRUCKS/DAY  599.50
NUMBER OF SMALL TRUCKS/DAY  737.50
```

As can be seen, the system is fairly well balanced. If anything, the two final inspectors are a bit overworked.

THE MATRIX SAVEVALUE BLOCK

Matrices can have their elements modified by having a transaction move into a block known as the MSAVEVALUE block. The general form of it is

```
◇        MSAVEVALUE A,B,C,D,E
```

where *A* is the name or number of the MSAVEVALUE, *B* is the row, *C* is the column, *D* is the new value, and *E* refers to the matrix type. Some examples of this block are

```
◇(a)     MSAVEVALUE JOE,1,2,5,MH
◇(b)     MSAVEVALUE 4,4,4,4,MX
◇(c)     MSAVEVALUE TOMMY,1,10,5.678,ML
◇(d)     MSAVEVALUE BETTY,FN(JOE)PH2,XH(BILL),MH
```

In (a), the element (1,2) of the matrix JOE is given the value 5; JOE is a half-word matrix. In (b), the element (4,4) of matrix 4 is given the value 4; the matrix 4 is a fullword matrix. In (c), the element (1,10) of the matrix TOMMY is given the value of 5.678; TOMMY is a floating-point matrix. In (d), the matrix BETTY is given a value as specified by savevalue BILL. This will go in

the element referenced by the function JOE and the transaction's second halfword parameter; BETTY is a halfword matrix.

Matrix savevalues can also be used in increment and decrement mode just as ordinary savevalues. Thus,

```
◇(a)      □MSAVEVALUE□FIRST+,2,5,8,MH
◇(b)      □MSAVEVALUE□3-,1,5,FN(AMTONE),MH
```

In (a), the matrix FIRST will have the element at (2,5) incremented by 8. In (b), the matrix 3 will have the element at (1,5) decremented by the value obtained by referencing the function AMTONE.

Example 25.2

This example is a modification of one presented by Schriber (1974, p. 295-301; see Preface for availability).

A production shop is composed of 5 different groups of machines. Each group consists of a number of machines of a given kind, as indicated by Table 25.1.

TABLE 25.1 Composition of machine groups

	Machines in Group		
Group	Kind	Number of That Kind Available	Storage Designation
A	Machine 1	9	S1
B	Machine 2	5	S2
C	Machine 3	3	S3
D	Machine 4	7	S4
E	Machine 5	16	S5

Note: In the simulation, these groups will be used as storages.

The shop produces four variations of the same product, designated as model 1, 2, 3, or 4. Production of a given model requires that operations be performed at specified kinds of machines in a specified sequence. The total number and kind of machines that each model must visit during its production—and the corresponding visitation sequences—are shown in Table 25.2. For example, model 1 must visit a total of 4 machines. The machines themselves, listed in the sequence in which they must be visited to make a finished model 1 product, are machine 1 (there are 9 machines of this kind available, per Table 25.1), machine 2 (there are 5 machines of this kind available), machine 4 (there are 7 machines of this kind available), and machine 5 (there are 16 machines of this kind available). (Machine 3 is not used to produce model 1.) Table 25.2 also shows the mean time required by each machine for each operation that must be performed to produce a given model. For example, the machine 1 operation to produce model 1 requires 250 seconds, on average. These times are all exponentially distributed. Jobs arrive at the shop in a Poisson stream at a mean rate of 1 job every 15 seconds; 15% of the jobs are model 1, 25% are model 2, 40% are model 3, and the rest are model 4.

TABLE 25.2 Visitation sequence and operation times for the machines to produce each model

Model Number	Total Number of Machines to be Visited	Machine Visitation Sequence	Mean Operation Time (seconds)
1	4	Machine 1	250
		Machine 2	105
		Machine 4	140
		Machine 5	130
2	3	Machine 2	45
		Machine 3	95
		Machine 5	50
3	5	Machine 4	75
		Machine 5	270
		Machine 3	25
		Machine 2	85
		Machine 1	150
4	3	Machine 3	30
		Machine 4	65
		Machine 5	210

Write the GPSS/H code to simulate the operation of the shop to determine the utilization of each of the 5 different machines and the production of each of the 4 models per day. Run the simulation for 10 days with each day consisting of 8 hours of work.

Solution

The following is a rather remarkable example of the use of matrices for greatly reducing the number of programming lines. It would be a straightforward and simple exercise to program this situation by using different code segments with the jobs arriving and then being routed to the appropriate segment, depending on which model was to be produced. A much more elegant solution using matrices is described next.

Table 25.3 gives in matrix form the "visitation sequence" for each model for the machine groups *yet to be visited.* The visitation sequence is "backwards" from what may be expected. The reason for this sequencing will be explained below.

TABLE 25.3 The "visitation sequence" matrix

Row Labels Represent the Model Numbers	Column Headings Represent the Machines Yet to be Visited, in Reverse Sequence				
	1	2	3	4	5
1	5	4	2	1	
2	5	3	2		
3	1	2	3	5	4
4	5	4	3		

Note: Values in the matrix are the specific machine numbers for each step of producing each model. Compare this matrix with Table 25.2.

There are four rows in the matrix, one for each model. The matrix entries are the machines yet to be visited, in reverse sequence, for the specific model. Thus, for model 1, the machine sequence for doing the required work is 1, 2, 4, and then 5 (see Table 25.2). Another matrix is shown in Table 25.4 that gives the corresponding times in seconds for each machine's operation. Again, for each model, this matrix gives the times for each machine's operation *yet to be done.*

Table 25.4 Mean operation time matrix

Row Labels Represent the Model Numbers	Column Headings Represent the Machines Yet to be Visited, in Reverse Sequence				
	1	2	3	4	5
1	130	140	105	250	
2	50	95	45		
3	150	85	25	270	75
4	210	65	30		

Note: Values in the matrix are the required times of operation (in seconds) for machines yet to be visited, in reverse sequence, for each step of producing each model. Compare this matrix with Table 25.2.

At this point, it is probably not at all obvious why these matrices were set up and how they will be used to simulate the system. Here is how the program will work (see the code listing that follows).

Suppose a job (i.e., a transaction) comes into the system and requires some kind of work; the exact kind of work, i.e., what model the job is, depends on the probability of the job's being a particular model (model 1, 2, 3, or 4), which is determined by the function labeled MODEL. For example, a job arrives to be worked on that is determined by this function to be a model 3; farther on in the code, an ASSIGN block places a 3, the value of FN(MODEL), in the transaction's PH1. Next, the function labeled CLASS matches the value of the model–3–to the number of machines required to produce the model–which is 5, as seen in Table 25.2; a subsequent ASSIGN block places a 5, the value of FN(CLASS), in the transaction's PH2.

To produce a model 3, Table 25.2 also shows that the sequence of the job's visitation to the various kinds of machines is 4, 5, 3, 2, 1. The same table shows that the times for these machines to finish their operations to produce a model 3 are 75, 270, 25, 85, and 150 seconds, respectively. In the code, therefore, matrices are set up that have the same values as those in Tables 25.3 and 25.4. Continuing the example, the first machine to work on model 3 is machine 4. The next one is machine 5, the next is machine 3, then machine 2, and finally machine 1. Notice that the value in Table 25.3 in position (3,5) is 4. The entry for (3,4) is 5, that for (3,3) is 3, that for (3,2) is 2, and that for (3,1) is 1. The code has the transaction enter the machine sequence in the same manner as defined in the third INITIAL statement, representing the third row of halfword matrix 1 (Table 25.3). A similar sequence is made for the four rows in Table 25.4 for the times to do each operation (halfword matrix 2). Finally, the various numbers of machines of a given kind are set up as storages, i.e., group A has 9 machines of kind 1 so S1 = 9 (see Table 25.1).

```
◇       □SIMULATE
*       The next line makes certain that enough computer storage is
*       available. This statement is required for the student
*       version of GPSS/H.
◇       □REALLOCATE□COM,30000
◇       □INTEGER    □&I,&MODELA,&MODELB,&MODELC,&MODELD
◇       □DO         □&I=1,25
◇       □PUTSTRING □('□□')
◇       □ENDDO
*
•       The next pair of lines creates halfword matrices numbered 1 and 2,
*       each of size 4 by 5.
◇1      □MATRIX     □MH,4,5
◇2      □MATRIX     □MH,4,5
*
*       The next pair of lines statistically determines which model the next
*       incoming job will be.
◇MODEL  □FUNCTION   □RN1,D4
.15,1/.4,2/.8/3/1,4
*
*       The next pair of lines matches the model of the incoming job
*       to number of machines required to produce the model (Table 25.2).
◇CLASS  □FUNCTION   □PH1,L4
1,4/2,3/3,5/4,3
*
*       The next 4 lines represent the visitation sequence for models 1 to 4,
*       respectively (halfword matrix 1; Table 25.3).
◇       □INITIAL    □MH1(1,1),5/MH1(1,2),4/MH1(1,3),2/MH1(1,4),1
◇       □INITIAL    □MH1(2,1),5/MH1(2,2),3/MH1(2,3),2
◇       □INITIAL    □MH1(3,1),1/MH1(3,2),2/MH1(3,3),3/MH1(3,4),5/MH1(3,5),4
◇       □INITIAL    □MH1(4,1),5/MH1(4,2),4/MH1(4,3),3
•
*       The next 4 lines represent the operation times for models 1 to 4,
•       respectively (halfword matrix 2; Table 25.4).
◇       □INITIAL    □MH2(1,1),130/MH2(1,2),140/MH2(1,3),105/MH2(1,4),250
◇       □INITIAL    □MH2(2,1),50/MH2(2,2),95/MH2(2,3),45
◇       □INITIAL    □MH2(3,1),150/MH2(3,2),85/MH2(2,3),25/MH2(3,4),270/MH2(3,5),75
◇       □INITIAL    □MH2(4,1),210/MH2(4,2),65/MH2(4,3),30
*
*       The next line represents the number of machines in each group (Table 25.1).
◇       □STORAGE    □S1,9/S2,5/S3,3/S4,7/S5,16
◇       □GENERATE   □RVEXPO(1,15)
◇       □ASSIGN     □1,FN(MODEL),PH
◇       □ASSIGN     □2,FN(CLASS),PH
◇BACKUP □ENTER      □MH1(PH1,PH2)
◇       □ADVANCE    □RVEXPO(1,MH2(PH1,PH2))
◇       □LEAVE      □MH1(PH1,PH2)
◇       □LOOP       □2PH,BACKUP
◇       □TEST□E     □PH1,1,NEXT1
◇       □BLET       □&MODELA=&MODELA+1
◇       □TERMINATE
◇NEXT1  □TEST□E     □PH1,2,NEXT2
◇       □BLET       □&MODELB=&MODELB+1
◇       □TERMINATE
◇NEXT2  □TEST□E     □PH1,3,NEXT3
```

```
◇         BLET       &MODELC=&MODELC+1
◇         TERMINATE
◇NEXT3    BLET       &MODELD=&MODELD+1
◇         TERMINATE
◇         GENERATE   3600*8*10
◇         TERMINATE  1
◇         START      1
◇         DO         &I=1,25
◇         PUTSTRING  ('')
◇         ENDDO
◇         PUTPIC     LINES=11,SR1/10.,SR2/10.,SR3/10.,SR4/10.,SR5/10.,_
                      &MODELA/10.,&MODELB/10.,&MODELC/10.,&MODELD/10.
 UTIL. OF MACHINES IN GROUP A  ***.**%
 UTIL. OF MACHINES IN GROUP B  ***.**%
 UTIL. OF MACHINES IN GROUP C  ***.**%
 UTIL. OF MACHINES IN GROUP D  ***.**%
 UTIL. OF MACHINES IN GROUP E  ***.**%
       AMOUNT MADE PER DAY
    MODEL          AMOUNT
      1            ***.**
      2            ***.**
      3            ***.**
      4            ***.**
◇         END
```

The output from the program follows:

```
UTIL. OF MACHINES IN GROUP A   71.76%
UTIL. OF MACHINES IN GROUP B   79.00%
UTIL. OF MACHINES IN GROUP C   88.92%
UTIL. OF MACHINES IN GROUP D   60.10%
UTIL. OF MACHINES IN GROUP E   74.71%
      AMOUNT MADE PER DAY
   MODEL          AMOUNT
     1            281.00
     2            477.40
     3            755.70
     4            382.70
```

As can be seen, the system seems to be working all right, i.e., all the groups of machines have about the same utilization. The machines in group D are used the least (60.10%), and those in group C are used the most (88.92%). Perhaps another machine needs to be added to group C. This is left as an exercise.

Example 25.3

The following matrix is called a "transition matrix" to find brand loyalty for 3 brands of soft drinks.

	Brand X	Brand Y	Brand Z	Total
Brand X	0.65	0.25	0.10	1.00
Brand Y	0.30	0.40	0.30	1.00
Brand Z	0.15	0.10	0.75	1.00

The matrix gives the probability that a person who last purchased a particular brand will purchase it again or another brand. Thus, a person who purchased brand Y the last time has a 30% chance of purchasing brand X the next time, a 40% chance of purchasing brand Y again, and a 30% chance of switching to brand Z. Notice that the probabilities add up to 100%. This matrix assumes that there are only the 3 choices available. In reality, there will be quite a few other possibilities, and the transition matrix will be considerably larger.

The problem is to determine what the long-term market shares for each of the 3 products will be, assuming that the transition matrix does not change owing to different advertising campaigns, consumer purchasing habits, etc. For example, if there are 10,000 purchases, which of these will be brand X, which brand Y, and which brand Z? Such a process is called a Markov process.

The method of solving this by simulation is as follows. A matrix such as the following is set up with the following initial given values:

	X	Y	Z
X	1	0	0
Y	0	0	0
Z	0	0	0

This says that at time 0 one person is drinking brand X. This person is now going to make a purchase. Suppose this is brand X again. The matrix is now

	X	Y	Z
X	2	0	0
Y	0	0	0
Z	0	0	0

The next purchase is brand Y. This gives the matrix

	X	Y	Z
X	2	1	0
Y	0	0	0
Z	0	0	0

This says that a person who was drinking brand X made brand Y his next purchase. If the next purchase is brand Z, the matrix becomes

	X	Y	Z
X	2	1	0
Y	0	0	1
Z	0	0	0

This is continued for a very long number of purchases. The matrix will eventually reach a steady state. The percentage of the number of switches to each brand will give the market shares. For example, if there were 1000 purchases and brand X was selected 600 times, its market share would be 60%.

Solution

The program to do the simulation is remarkably short:

```
◇          □SIMULATE
◇          □INTEGER   □&NEW,&OLD,&I
◇          □REAL      □&SUM
◇TLOYAL    □TABLE     □&NEW,1,1,10
◇LOYAL     □MATRIX    □MH,3,3
◇DRINK     □FUNCTION  □&NEW,M3
1,FN(BRANDX)/2,FN(BRANDY)/3,FN(BRANDZ)
◇BRANDX    □FUNCTION  □RN1,D3
0.65,1/.9,2/1,3
◇BRANDY    □FUNCTION  □RN1,D3
0.3,1/.7,2/1,3
◇BRANDZ    □FUNCTION  □RN1,D3
.15,1/.25,2/1,3
◇          □LET       □&NEW=1
◇          □GENERATE  □1
◇          □TABULATE  □TLOYAL
◇          □BLET      □&OLD=&NEW
◇          □BLET      □&NEW=FN(DRINK)
◇          □MSAVEVALUE□LOYAL+,&OLD,&NEW,1,MH
◇          □TERMINATE □1
◇          □START     □100000
◇          □DO        □&I=1,25
◇          □PUTSTRING □('□')
◇          □ENDDO
◇          □LET       □&SUM=MH$LOYAL(1,1)+MH$LOYAL(1,2)+MH$LOYAL(1,3)
◇          □LET       □ML$NEWMAT(1,1)=MH$LOYAL(1,1)/&SUM
◇          □LET       □ML$NEWMAT(1,2)=MH$LOYAL(1,2)/&SUM
◇          □LET       □ML$NEWMAT(1,3)=MH$LOYAL(1,3)/&SUM
◇          □LET       □&SUM=MH$LOYAL(2,1)+MH$LOYAL(2,2)+MH$LOYAL(2,3)
◇          □LET       □ML$NEWMAT(2,1)=MH$LOYAL(2,1)/&SUM
◇          □LET       □ML$NEWMAT(2,2)=MH$LOYAL(2,2)/&SUM
◇          □LET       □ML$NEWMAT(2,3)=MH$LOYAL(2,3)/&SUM
◇          □LET       □&SUM=MH$LOYAL(3,1)+MH$LOYAL(3,2)+MH$LOYAL(3,3)
◇          □LET       □ML$NEWMAT(3,1)=MH$LOYAL(3,1)/&SUM
◇          □LET       □ML$NEWMAT(3,2)=MH$LOYAL(3,2)/&SUM
◇          □LET       □ML$NEWMAT(3,3)=MH$LOYAL(3,3)/&SUM
◇          □PUTPIC    □LINES=7,MH$LOYAL(1,1),MH$LOYAL(1,2),MH$LOYAL(1,3),_
                             MH$LOYAL(2,1),MH$LOYAL(2,2),MH$LOYAL(2,3),_
                             MH$LOYAL(3,1),MH$LOYAL(3,2),MH$LOYAL(3,3)
                                 LOYALTY MATRIX
0                                TIME SWITCHED TO
                      BRAND   X        Y        Z
       WHEN DRINKING   X    ****     ****     ****
                       Y    ****     ****     ****
                       Z    ****     ****     ****
0      NUMBERS CORRESPOND TO NEW PURCHASES WHEN DRINKING OLD BRAND
◇          □PUTPIC    □LINES=5,ML$NEWMAT(1,1),ML$NEWMAT(1,2),_
                        ML$NEWMAT(1,3),_
                        ML$NEWMAT(2,1),ML$NEWMAT(2,2),ML$NEWMAT(2,3),_
                        ML$NEWMAT(3,1),ML$NEWMAT(3,2),ML$NEWMAT(3,3)
```

```
0                NEW TRANSITION MATRIX
0             BRAND       X        Y        Z
                X        *.**     *.**     *.**
                Y        *.**     *.**     *.**
                Z        *.**     *.**     *.**
◇          END
```

The output looks as follows:

```
                                   LOYALTY MATRIX
                                   TIME SWITCHED TO
                              BRAND   X        Y        Z
           WHEN DRINKING        X   23392     9130     3677
                                Y    6526     8614     6726
                                Z    6280     9122    31533
  NUMBERS CORRESPOND TO NEW PURCHASES WHEN DRINKING OLD BRAND
  NEW TRANSITION MATRIX
  BRAND       X       Y       Z
    X       0.65    0.25    0.10
    Y       0.30    0.39    0.31
    Z       0.15    0.10    0.75
```

EXERCISES, CHAPTER 25

1. Give the code to specify a halfword matrix with dimensions of 3 by 3. The initial values along the diagonals are 1, 2, and 3. The other initial values are 0.
2. Give the code to specify a floating-point matrix with dimensions of 1 by 4. The initial values are to be 0, 1, 2, and 3.
3. In the following exercise, values for the various entities are R(TUGS) = 1, Q(FIRST) = 3, FN(JJJJ) = 5, and all PH values are 3. Determine what happens when a transaction passes through each of the following matrix savevalue blocks:

```
◇(a)     MSAVEVALUE TOMMY,1,3,5,MH
◇(b)     MSAVEVALUE BILLY+,3,5,7,MH
◇(c)     MSAVEVALUE FIRST,1,1,PH3,MH
◇(d)     MSAVEVALUE PH3,2,2,Q(FIRST),MH
◇(e)     MSAVEVALUE FN(JJJJ),PH1,PH2,PH3,MH
```

4. Suppose a transition matrix for 5 brands is as follows:

	Brand A	Brand B	Brand C	Brand D	Brand E
Brand A	0.45	0.20	0.10	0.20	0.05
Brand B	0.20	0.60	0.10	0.05	0.05
Brand C	0.05	0.20	0.50	0.15	0.10
Brand D	0.05	0.05	0.05	0.70	0.15
Brand E	0.10	0.10	0.25	0.15	0.40

Modify the program used in Example 25.3 to determine the long-term market shares.

5. In Exercise 25.4, the creators of Brand E have decided to spend considerable funds on a new advertising campaign. They feel that the transition matrix will be changed to the following:

	Brand A	Brand B	Brand C	Brand D	Brand E
Brand A	0.40	0.10	0.10	0.15	0.25
Brand B	0.10	0.45	0.05	0.05	0.35
Brand C	0.05	0.10	0.50	0.10	0.25
Brand D	0.05	0.05	0.05	0.50	0.35
Brand E	0.10	0.10	0.05	0.05	0.70

Modify the program developed in Exercise 25.4 to determine the new long-term market shares assuming that none of the other brands reacts to the Brand E advertising campaign.

SOLUTIONS, CHAPTER 25

1.
```
PROB1   MATRIX     MH3,3
     INITIAL MH$PROB1(1,1),1/MH$PROB1(2,2),2/MH$PROB1(3,3),3
```

2.
```
PROB2   MATRIX     ML,1,4
     INITIAL ML$PROB2(1,2),1/ML$PROB2(1,3),2/ML$PROB2(1,4),3
```

3. a. The halfword matrix savevalue TOMMY will have the value 5 in position (1,3).
   b. The halfword matrix savevalue BILLY will have the value 7 added to whatever is currently in position (3,5).
   c. The halfword matrix savevalue FIRST will have the value 3 in position (1,1).
   d. The 3rd halfword matrix savevalue will have the value of 3 in position (2,2).
   e. The 5th halfword matrix savevalue will have the value of 3 in position (3,3).
4. The program to do the simulation is as follows:

```
         SIMULATE
         INTEGER      &NEW,&OLD,&I
         REAL         &SUM
 MYOUT   FILEDEF      'MARKET2.OUT'
 TLOYAL  TABLE        &NEW,1,1,10
 LOYAL   MATRIX       MH,5,5
 NEWMAT  MATRIX       ML,5,5
 DRINK   FUNCTION     &NEW,M5
1,FN(BRANDA)/2,FN(BRANDB)/3,FN(BRANDC)/4,FN(BRANDD)/5,FN(BRANDE)
 BRANDA  FUNCTION     RN1,D5
0.45,1/.65,2/.75,3/.95,4/1,5
 BRANDB  FUNCTION     RN1,D5
0.2,1/.8,2/.9,3./.95,4/1,5
 BRANDC  FUNCTION     RN1,D5
.05,1/.25,2/.75,3/.9,4/1,5
 BRANDD  FUNCTION    RN1,D5
```

```
.05,1/.1,2/.15,3/.85,4/1,5
 BRANDE    FUNCTION  RN1,D5
.1,1/.2,2/.45,3/.6,4/1,5
         LET         &NEW=1
         GENERATE    1
         TABULATE    TLOYAL
         BLET        &OLD=&NEW
         BLET        &NEW=FN(DRINK)
         MSAVEVALUE  LOYAL+,&OLD,&NEW,1,MH
         TERMINATE   1
         START       100000
         DO          &I=1,25
         PUTSTRING   ('  ')
         ENDDO
         LET          &SUM=MH$LOYAL(1,1)+MH$LOYAL(1,2)+MH$LOYAL(1,3)+_
                      MH$LOYAL(1,4)+MH$LOYAL(1,5)
         LET          ML$NEWMAT(1,1)=MH$LOYAL(1,1)/&SUM
         LET          ML$NEWMAT(1,2)=MH$LOYAL(1,2)/&SUM
         LET          ML$NEWMAT(1,3)=MH$LOYAL(1,3)/&SUM
         LET          ML$NEWMAT(1,4)=MH$LOYAL(1,4)/&SUM
         LET          ML$NEWMAT(1,5)=MH$LOYAL(1,5)/&SUM
         LET          &SUM=MH$LOYAL(2,1)+MH$LOYAL(2,2)+MH$LOYAL(2,3)+_
                      MH$LOYAL(2,4)+MH$LOYAL(2,5)
         LET          ML$NEWMAT(2,1)=MH$LOYAL(2,1)/&SUM
         LET          ML$NEWMAT(2,2)=MH$LOYAL(2,2)/&SUM
         LET          ML$NEWMAT(2,3)=MH$LOYAL(2,3)/&SUM
         LET          ML$NEWMAT(2,4)=MH$LOYAL(2,4)/&SUM
         LET          ML$NEWMAT(2,5)=MH$LOYAL(2,5)/&SUM
         PUTPIC       &SUM,MH$LOYAL(2,5),ML$NEWMAT(2,5)
   SUM ***    LOYAY(2,5)   ***    NEWMAT *.**
         LET          &SUM=MH$LOYAL(3,1)+MH$LOYAL(3,2)+MH$LOYAL(3,3)+_
                      MH$LOYAL(3,4)+MH$LOYAL(3,5)
         LET          ML$NEWMAT(3,1)=MH$LOYAL(3,1)/&SUM
         LET          ML$NEWMAT(3,2)=MH$LOYAL(3,2)/&SUM
         LET          ML$NEWMAT(3,3)=MH$LOYAL(3,3)/&SUM
         LET          ML$NEWMAT(3,4)=MH$LOYAL(3,4)/&SUM
         LET          ML$NEWMAT(3,5)=MH$LOYAL(3,5)/&SUM
         LET          &SUM=MH$LOYAL(4,1)+MH$LOYAL(4,2)+MH$LOYAL(4,3)+_
                      MH$LOYAL(4,4)+MH$LOYAL(4,5)
         LET          ML$NEWMAT(4,1)=MH$LOYAL(4,1)/&SUM
         LET          ML$NEWMAT(4,2)=MH$LOYAL(4,2)/&SUM
         LET          ML$NEWMAT(4,3)=MH$LOYAL(4,3)/&SUM
         LET          ML$NEWMAT(4,4)=MH$LOYAL(4,4)/&SUM
         LET          ML$NEWMAT(4,5)=MH$LOYAL(4,5)/&SUM
         LET          &SUM=MH$LOYAL(5,1)+MH$LOYAL(5,2)+MH$LOYAL(5,3)+_
                      MH$LOYAL(5,4)+MH$LOYAL(5,5)
         LET          ML$NEWMAT(5,1)=MH$LOYAL(5,1)/&SUM
         LET          ML$NEWMAT(5,2)=MH$LOYAL(5,2)/&SUM
         LET          ML$NEWMAT(5,3)=MH$LOYAL(5,3)/&SUM
         LET          ML$NEWMAT(5,4)=MH$LOYAL(5,4)/&SUM
         LET          ML$NEWMAT(5,5)=MH$LOYAL(5,5)/&SUM
         PUTPIC       LINES=9,FILE=MYOUT,MH$LOYAL(1,1),MH$LOYAL(1,2),_
```

```
                    MH$LOYAL(1,3),_
                    MH$LOYAL(1,4),MH$LOYAL(1,5),_
                     MH$LOYAL(2,1),MH$LOYAL(2,2),MH$LOYAL(2,3),_
                     MH$LOYAL(2,4),MH$LOYAL(2,5),_
                     MH$LOYAL(3,1),MH$LOYAL(3,2),MH$LOYAL(3,3),_
                     MH$LOYAL(3,4),MH$LOYAL(3,5),_
                     MH$LOYAL(4,1),MH$LOYAL(4,2),MH$LOYAL(4,3),_
                     MH$LOYAL(4,4),MH$LOYAL(4,5),_
                     MH$LOYAL(5,1),MH$LOYAL(5,2),MH$LOYAL(5,3),_
                     MH$LOYAL(5,4),MH$LOYAL(5,5)
                  LOYALTY MATRIX IS AS FOLLOWS:
0                           TIME SWITCHED TO:
                   BRAND   A        B        C        D        E
        WHEN USING   A   *****    *****    *****    *****    *****
        THIS BRAND   B   *****    *****    *****    *****    *****
                     C   *****    *****    *****    *****    *****
                     D   *****    *****    *****    *****    *****
                     E   *****    *****    *****    *****    *****
0             NUMBERS CORRESPOND TO NEW PURCHASES WHEN USING OLD BRAND
         PUTPIC       LINES=7,FILE=MYOUT,ML$NEWMAT(1,1),_
                      ML$NEWMAT(1,2),ML$NEWMAT(1,3),_
                      ML$NEWMAT(1,4),ML$NEWMAT(1,5),_
                      ML$NEWMAT(2,1),ML$NEWMAT(2,2),ML$NEWMAT(2,3),_
                      ML$NEWMAT(2,4),ML$NEWMAT(2,5),_
                      ML$NEWMAT(3,1),ML$NEWMAT(3,2),ML$NEWMAT(3,3),_
                      ML$NEWMAT(3,4),ML$NEWMAT(3,5),_
                      ML$NEWMAT(4,1),ML$NEWMAT(4,2),ML$NEWMAT(4,3),_
                      ML$NEWMAT(4,4),ML$NEWMAT(4,5),_
                      ML$NEWMAT(5,1),ML$NEWMAT(5,2),ML$NEWMAT(5,3),_
                      ML$NEWMAT(5,4),ML$NEWMAT(5,5)
0              NEW TRANSITION MATRIX IS:
0              BRAND       A       B       C       D       E
                 A        *.**    *.**    *.**    *.**    *.**
                 B        *.**    *.**    *.**    *.**    *.**
                 C        *.**    *.**    *.**    *.**    *.**
                 D        *.**    *.**    *.**    *.**    *.**
                 E        *.**    *.**    *.**    *.**    *.**
         END
```

The results of the simulation are:

```
                    LOYALTY MATRIX IS AS FOLLOWS:

                              TIME SWITCHED TO:
                  BRAND   A        B        C        D        E
      WHEN USING    A    6771     3034     1578     2997      767
      THIS BRAND    B    4662    13937     2248     1259     1241
                    C     917     3453     8899     2688     1737
                    D    1435     1505     1477    20956     4595
                    E    1361     1418     3493     2068     5504

      NUMBERS CORRESPOND TO NEW PURCHASES WHEN USING OLD BRAND

           NEW TRANSITION MATRIX IS:

           BRAND       A       B       C       D       E
```

```
A      0.45  0.20  0.10  0.20  0.05
B      0.20  0.60  0.10  0.05  0.05
C      0.05  0.20  0.50  0.15  0.10
D      0.05  0.05  0.05  0.70  0.15
E      0.10  0.10  0.25  0.15  0.40
```

From the .LIS file the market shares are:

BRAND	Share
A	15.15
B	23.35
C	17.69
D	29.97
E	13.84

5. The program given for Exercise 25.4 is easily modified by changing the 5 functions BRANDA, BRANDB, BRANDC, BRANDD and BRANDE.

   The results of the simulation are as follows:

BRAND	Share
A	12.61
B	14.11
C	10.21
D	12.30
E	50.25

As can be seen, the advertising campaign was tremendously successful with Brand E now enjoying over 50% of the market.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 26

Variables, Expressions, and the PRINT Block

ARITHMETIC IN GPSS/H

Arithmetic in GPSS/H was mentioned in Chapter 5 and also in other chapters. It has been used regularly throughout the book with little explanation because arithmetic is allowed in the operands of the various blocks and is done in a logical manner. This feature did not exist in earlier versions of GPSS: all arithmetic was done by reference to an expression. Thus, in this book, when we have had a block such as ADVANCE 2*100, it has not been necessary to point out that the time on the FEC for the transaction was 200. This chapter formally presents the steps in performing arithmetic; however, there are certain cautions that must be observed. These cautions have to do with the original integer nature of calculations in GPSS. In addition, there will be a time when one will need to do arithmetic calculations by referencing an expression, which is introduced in this chapter.

Recall that arithmetic is accomplished in GPSS/H by using SNAs together with various arithmetic operations:

+	addition
–	subtraction (both unary and binary subtraction)
/	division
*	multiplication
@	modular division

We have been using all of these operations when needed except, possibly, modular division. Modular division can be quite handy. It is defined as the remainder when two numbers are divided. For example, 7 @ 3 is 1; 9 @ 9 is 0 (no remainder); 8 @ 12 is 8 since the result of division is 0 with a remainder of 8. An example of its use is

```
            ASSIGN    2,RN1@3+1,PH
```

would assign to halfword parameter 2 a number from 1 to 3 with equal probability.

```
◇TIMEF   □FUNCTION  □AC1@480,E8
60,FN(TIME1)/120,FN(TIME2)/180,FN(TIME3)/240,FN(TIME4)
300,FN(TIME5)/360,FN(TIME6)/420,FN(TIME7)/480,FN(TIME8)
```

would reference the absolute clock and sample from 1 of 8 functions. If a working shift is 480 minutes in duration, this code might be used to have a different time for doing a process depending on the hour.

An arithmetic expression is most often placed in an operand, but it also can be referenced in a manner similar to referencing functions. This approach is useful when a particular expression is referenced many times and can save considerable time in writing the code. Expressions are referenced by defining the expression through the use of either a VARIABLE (for integer arithmetic) or an FVARIABLE (for floating-point arithmetic) statement. The forms are

```
◇LABEL   □VARIABLE  □A
```

and

```
◇LABEL   □FVARIABLE □A
```

where the *A* operand is the expression to be defined. Numbers can be used for the labels. So,

```
◇TOMMY   □VARIABLE  □3PH+FR(MACH)/200-Q(WAIT)
◇1       □VARIABLE  □XF(FIRST)@XF(SECOND)+QM(NEXT)
◇CALC    □FVARIABLE □3+Q(LAST)*F(MACH)-R(TUGS)
```

are possible ways to define variables TOMMY, 1, and CALC. *Notice that no spaces are allowed in expressions* whereas, in other programming languages, such as Fortran or Pascal, blank spaces are recommended for clarity. A blank space in an expression in GPSS/H will terminate the expression and most probably cause an error.

Referencing variables is done by

```
V(LABEL)
```

or

```
V$LABEL
```

i.e., by V followed by the label of the variable definition either in parentheses or preceded by a dollar sign, "$." When an expression is referenced, it is evaluated and the result is returned. Evaluation of expressions follows the usual rules found in other programming languages: calculations proceed from left to right, and multiplication, division, and modular division have precedence over addition and subtraction. Parentheses are used for grouping and clarity. Whatever expression is innermost in nested parentheses is done first. Inner parentheses have preference over outer parentheses.

If the variable expression is defined by a VARIABLE statement, only integer calculations are done. All division is integer division, i.e., the result is truncated. However, *the final result is a decimal, whole-number value.* This decimal value must end in 6 zeroes owing to the nature of having done integer calculations. Thus, the variable JERRY defined as

```
◇JERRY    □VARIABLE  □3/2+1
```

would return a value of 2.000000 when referenced by V(JERRY). With FVARIABLES, the expression is evaluated by doing floating-point calculations. If necessary, nondecimal values are converted to floating-point values. So, if the variable JERRY was instead defined as follows, the value returned will be 2.5:

```
◇JERRY    □FVARIABLE □3/2+1
```

If an expression is used in an operand, then the type of calculations performed depend on whether integer or floating-point values are used. Many times a floating-point result is desired, but the variables used in the expression are integers. For example, suppose that the simulation went for 100 days and what was desired was the time a facility was captured per day. The expression for this might look like

```
◇         □LET        □&AVGPROD=FC(CRUSHER)/&DAYS
```

The result will be an integer even though the ampervariable &AVGPROD has been specified as REAL. As described in Chapter 20, GPSS/H has two built-in functions to handle arithmetic calculations when one wants to specify either fixed-point or floating-point calculations. These are FIX for fixed-point conversion and FLT for floating-point conversion. Once you specify FLT in an expression for a single SNA, the whole expression is evaluated as though it was using floating-point calculations. Thus, to achieve a floating-point result, the preceding code could have been written as

```
◇         □LET        □&AVGPROD=FLT(FC(CRUSHER))/&DAYS
```

Consider the following examples:

```
◇(a)      □ADVANCE    □3/2+1
◇(b)      □ADVANCE    □FLT(3)/2+1
```

In (a), the delay time is 2.0, but in (b), the delay time is 2.5. However, to achieve the desired result in any given line of code, one must use appropriate GPSS/H components, for example:

```
◇(c)      □SAVEVALUE □FIRST,FLT(3)/2+1,XH
◇(d)      □SAVEVALUE □SECOND,FLT(3)/2+1,XL
◇(e)      □SAVEVALUE □THIRD,3/2+1,XL
```

In (c), although a floating-point result is apparently desired from the use of FLT, FIRST has an integer value of 2 because of the use of XH (there would be a compiler warning: "floating-point constant truncated to an integer value"). In (d), the value of SECOND is 2.5 since the savevalue is specified as being floating point. In (e), the value of THIRD is 2.0.

THE PRINT BLOCK

It is possible to have statistical information sent to the report while the program is being run. This statistical information consists of the SNAs associated with a particular entity at the time the information is sent to the report. Collecting this information is done using a PRINT block. When a transaction enters this block, all the statistics associated with the specified entity (or entities) are sent to the report file. The form of the PRINT block is

```
◇       PRINT     A,B,C
```

where operand *A* is the lower limit of the entity range (often omitted), operand *B* is the upper limit of the entity range (often omitted), and operand *C* is the family name of the entity to be printed out. Formerly, there was an operand *D* that was used as a printer directive, but is now ignored. It will not be used in this book. If *A* and *B* are omitted, all the statistics for the entity class are printed out. Furthermore, if you use labels for an entity, as is the common case, it is not possible to have only selected ones printed out. Some examples of the PRINT block are as follows:

```
◇(a)    PRINT     1,3,XL
◇(b)    PRINT     ,,Q
◇(c)    PRINT     3,7,F
◇(d)    PRINT     ,,MH
◇(e)    PRINT     2,2,XF
```

In (a), the floating-point savevalues 1 to 3 are printed out. In (b), all the queue statistics are printed out. In (c), statistics for facilities 3 to 7 are printed out. In (d), all the halfword matrices are printed out. In (e), the fullword savevalue XF2 is printed out.

Since output is added to the normal GPSS/H report every time a transaction passes through a PRINT block, caution must be taken in using it. It is mostly used for debugging purposes and, then, only when the program is run for a limited time.

Some other possible entities that can be used in the PRINT block are

AMP	all ampervariables
B	current and total block execution counts
C	absolute and relative clock values
F	facilities
LG	all logic switches that are in a set position
MB, MH, ML, MX	various matrix values
N	current and total block execution counts
Q	queue statistics
RN	random number stream
S	storage statistics
T	table statistics
W	current and total block execution counts

Notice that, regardless of whether B, N, or W is used, the statistics sent to the report are identical.

EXERCISES, CHAPTER 26

1. Determine the value of the ampervariable or savevalue after the expression or operand is evaluated. Assume the following values and definitions: &FIRST is real; &I and &J are integers; and PH1 = 3, PH2 = 4, PH3 = 3, PH4 = 2, and &J = 560.

```
◇(a)     ▯BLET       ▯&FIRST=2+3/(PH1+1)
◇(b)     ▯BLET       ▯&I=&J@400
◇(c)     ▯SAVEVALUE  ▯TOM,5/PH3-PH4,XL
◇(d)     ▯SAVEVALUE  ▯PH3,PH3/PH2-&J@558,XH
◇(e)     ▯BLET       ▯&FIRST=PH3*PH1-PH4*4
```

2. Variables are defined as follows:

```
◇BETTY   ▯VARIABLE   ▯2+PH3-&J@555
◇JANE    ▯FVARIABLE  ▯PH3/2+1-PH3/5
```

Using the same data as for exercise 26.1, determine the values that will be obtained when the operands are evaluated for the following:

```
◇(a)     ▯ADVANCE    ▯V(BETTY)
◇(b)     ▯SAVEVALUE  ▯TOM,V(BETTY)-FV(JANE)XH
◇(c)     ▯ADVANCE    ▯PH3*PH2-PH4/4
◇(d)     ▯SAVEVALUE  ▯1,PH3-V(BETTY),XL
◇(e)     ▯SAVEVALUE  ▯MIKE,&J@550/PH3,XL
```

SOLUTIONS, CHAPTER 26

1. a. &FIRST has the value 2.00
   b. &I = 160
   c. TOM is -3.00
   d. The third floating point savevalue has the value -2.00
   e. &FIRST has the value 0.00
2. a. The transaction will be placed on the FEC for a time of 2.00
   b. TOM has the value 1
   c. The transaction will be placed on the FEC for a time of 12
   d. The first savevalue has the value 1.00
   e. MIKE has the value 4.00

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 27

Boolean Variables

The TEST block and the GATE block have been used to allow the programmer to control the flow of transactions through the program blocks. When the transaction enters a TEST block or a GATE block, depending on the type it is, the transaction may be delayed until some condition is true, it may pass through to the next sequential block, or it may be routed to another part of the program. However, both the TEST block and the GATE block only allow for *one* condition to be tested. In many situations, there will be multiple conditions to be tested for when a transaction arrives at a particular part of the program. These types of delay situations can be modeled by the use of Boolean variables. Boolean variables allow the programmer to specify user-supplied logic conditions to control the flow of transactions through a system. As such, they can be used to model very complex situations that require many different conditions to be satisfied. For example, a plane attempting a landing at a distant airport might need to meet the conditions: Is the airport open? Is the runway clear? Is there room in the hangar for the particular type of plane, etc.? The captain of a ship entering a harbor has to ask if the following conditions are true: Is there room for the ship to dock? Is a tugboat free to berth it? Is the harbor even open?

OPERATORS

Chapter 26 covered how fixed-point and floating-point expressions could be defined by using the VARIABLE and FVARIABLE statements. Another type of variable used in GPSS/H is known as a Boolean variable and must be defined in a BVARIABLE statement. This is a variable that is defined by the programmer. It will have only one of two values—either 0 or 1. Just as with other variables, the Boolean variable will have an associated expression, called a *logical expression*, which is evaluated. The value of the expression will be either 1 (true) or 0 (false). Boolean expressions are made up of SNAs or entities connected by one or more of the following:

1. relational operators
2. Boolean operators
3. logical operators

Relational (Comparison) Operators

These were introduced with the TEST and SELECT blocks. For convenience, they will be repeated here. When they are used in the TEST block or the SELECT block, they are used alone, but in Boolean expressions, they must have single (left) quotes on either side or the equivalent symbol may be used:

Meaning	Relational Operator	Equivalent Symbol
Greater than	'G'	>
Greater than or equal	'GE'	> =
Equal	'E'	=
Not equal	'NE'	! =
Less than or equal	'LE'	< =
Less than	'L'	<

Both the relational operator and the equivalent symbols are used in this book, but preference is given to the relational operator form as so many GPSS/H programs already written use this form. Some Boolean expressions illustrating the above are given next. For

```
Q(TOM)'E'Q(BILL)
```

or the equivalent

```
Q(TOM)=Q(BILL)
```

if the queue length of the queue TOM is equal to the queue length of the queue BILL, the expression is true and is set equal to 1; otherwise, the value is 0. For

```
XF(TEST1)'L'XF(TEST2)
```

the fullword savevalue TEST1 must be less than the fullword savevalue TEST2 in order for the expression to be true and, thus, set equal to 1. For

```
FR(MACHA)'G'250
```

the utilization of the facility MACHA must be greater than 250 for the expression to be true and thus set equal to 1. (Recall that the utilization of a facility is expressed in parts per thousand.)

Boolean Operators

The real power of Boolean variables comes from using Boolean operators to connect relational operators. There are three Boolean operators in GPSS/H. These are AND, OR, and NOT. These actual words are inserted between expressions that are enclosed in parentheses. The effect of each is identical to similar operators in languages such as Fortran, Pascal, BASIC, etc. Thus, the

Boolean operator AND returns a true result–and, thus, sets the Boolean variable's value to 1–only when the value of the expressions on both sides of it are true. Thus,

(true)AND(true)	is true or 1
(false)AND(true)	is false or 0
(true)AND(false)	is false or 0
(false)AND(false)	is false or 0

The operator OR returns a true result if *either or both* of the values of the expressions is true. Thus,

(true)OR(true)	is true or 1
(false)OR(true)	is true or 1
(true)OR(false)	is true or 1
(false)OR(false)	is false or 0

NOT inverts the value of an expression. Thus,

NOT(true)	is false or 0
NOT(false)	is true or 1

In the following examples assume that Q(TOM) = 3, Q(BILL) = 2, X(FIRST) = 5, X(SECOND) = 6, and PH1 = –4. The Boolean expressions are

```
(Q(TOM)'LE'4)AND(PH1'G'-5)
(Q(BILL)'E'2)OR(XF(FIRST)'E'6)
(XF(SECOND)'LE'XF(FIRST))AND(PH1'G'0)
(Q(TOM)'G'2)AND(NOT(XF(SECOND)'E'6))
```

The first two are true (value = 1), but the second two are false (value = 0).

Alternate symbols that can be used are + for OR and * for AND. Since these symbols also are used for arithmetic operations, this means that *one cannot do addition or multiplication in Boolean expressions!* Thus, if any addition or multiplication is required, one has to do it in another variable that is then referenced in the Boolean expression. For example,

```
◇ONE      ▯VARIABLE  ▯XF(FIRST)+XF(SECOND)-PH2
◇TWO      ▯BVARIABLE ▯(Q(TOM)>=2)OR(V(ONE=0))
```

The Boolean variable TWO will be true if the queue TOM is equal to or greater than 2 or if the variable ONE is equal to 0. It would not have been correct to write the Boolean variable as follows:

```
◇TWO      ▯BVARIABLE ▯(Q(TOM)>=2)OR(XF(FIRST)+XF(SECOND)-PH2)
```

The plus sign would not mean addition but would be taken as the Boolean operator OR.

The choice of + and * for OR and for AND is also confusing because it is so easy to look at the plus sign and mentally associate it with AND. However unfortunate this convention is, it is a part of the GPSS/H language. This is a

carryover from the early versions of GPSS. There is no symbol that can be used for NOT as it was not a feature of the early versions of GPSS.

The previous examples could have been written as

```
(Q(TOM)'LE'4)*(PH1'G'-5)
(Q(BILL)'E'2)+(XF(FIRST)=6)
(XF(SECOND)'LE'XF(FIRST))*(PH1'G'0)
(Q(TOM)'G'2)*(NOT(XF(SECOND)'E'6))
```

Since it is much more logical to write out AND, OR, and/or NOT, however, that is the approach used in this book. Also, many of the parentheses used above are not needed because of the rules for evaluation of Boolean expressions, described after the next section.

Logical Operators

It is possible to use logical operators to reference various entities in GPSS/H. A logical operator will check on the status of an entity condition. If the result of this check is true, the value is 1; otherwise, it is 0. The logical operators are SNAs. Some of them are listed here:

Logical Operator	Condition Referenced
FU(name) [or F(name)]	Is the facility in use?
FNU(name)	Is the facility not in use?
FS(name)	Is the facility seizeable?
FNS(name)	Is the facility not seizeable?
SE(name)	Is the storage empty?
SNE(name)	Is the storage not empty?
SF(name)	Is the storage full?
LS(name)	Is the logic switch set?
LR(name)	Is the logic switch reset?

If the entities referenced by the logical operator are identified by a number instead of by a name, the use of parentheses is optional. GPSS/H also supports the older form of referencing these logical operators using a single $ sign. Thus, LS$FIRST and LS(FIRST) are the same. Some examples of these operators are as follows: In SNF(TUGS), if the storage of TUGS is not full, the value is 1. In LR(STOP12), if the logic switch STOP12 is reset, the value is 1. In F(MACH1), if the facility MACH1 is in use, the value is 1. In LS6, the logic switch 6 is referenced; if it is set, the value is 1.

By combining relational operators, Boolean operators, and logical operators, many complex situations can be easily modeled. For example, consider the following:

```
(LR(GOIN))AND(Q(WAIT)'LE'3)AND(FNU(MACH2))
```

In order for this expression to be true, the following conditions must all be true: (1) the logic switch GOIN must be reset, (2) the queue at WAIT must be less than or equal to 3, and (3) MACH2 must be free.

REFERENCING BOOLEAN VARIABLES

Boolean variables are defined and referenced in much the same way as other variables. The general form is

```
◇LABEL   ▯BVARIABLE ▯A
```

where *A* is the desired expression. It is possible to have a Boolean variable with a number for a label. One references the Boolean variable by its label in the form BV(LABEL) or by the number *n* in the form BV*n*. Thus, one might have something like

```
◇          ▯TEST▯E     ▯BV(STOPIT),1
```

When a transaction arrives at this TEST block, the value of the Boolean variable STOPIT is determined. Unless it has the value 1, the transaction cannot move to the next sequential block. Instead, it is kept on the CEC and scanned again when a rescan is made.

It is also possible to reference Boolean variables by use of the single dollar sign, "$." Thus, the example above could have been written

```
◇          ▯TEST▯E     ▯BV$STOPIT,1
```

RULES FOR EVALUATION OF BOOLEAN EXPRESSIONS

The rules for evaluation of Boolean expressions are as follows:

1. Logical operators and relational operators have preference in a left to right order.
2. The operators AND, OR, and NOT are evaluated in a left to right order.

Parentheses can (and should) be used for clarity. When used, whatever is in the innermost pair is evaluated first.

Example 27.1

Consider a repair shop for trucks. Trucks arrive at the repair shop every 18 ± 4.5 hours. There are 3 service bays to work on the trucks. However, the repair shop is down quite often for one reason or the other. It is down on an average of every 20 hours (exponentially distributed) and remains down for 3.5 hours, also exponentially distributed. When it is down, repaired trucks cannot leave the repaired area. After a truck is repaired, a single worker has to inspect and refuel each truck. These tasks take 3 ± 1 hours. Model this repair shop to determine the expected number of trucks to be repaired each day, the utilization of the shop, the utilization of the single worker, and the average number of trucks in the queue. The repair shop operates 365 days a year, each day being defined as 24 hours of operation. Simulate for 100 years of operation.

Solution

```
◇          ▯SIMULATE
◇          ▯INTEGER    ▯&I
◇          ▯STORAGE    ▯S(BAYS),3
```

```
◇INBAY    □BVARIABLE □(LR(ALLOK))AND(SNF(BAYS))
◇OUTOK    □BVARIABLE □(FS(HELPER))AND(LR(ALLOK))
◇         □GENERATE  □18,4.5
◇         □QUEUE     □ATREPAIR          □JOIN QUEUE AT REPAIR SHOP
◇         □TEST□E    □BV(INBAY),1       □OK TO GO IN?
◇         □ENTER     □BAYS              □TAKE ONE OF THREE BAYS
◇         □DEPART    □ATREPAIR          □LEAVE THE QUEUE
◇         □ADVANCE   □.25               □POSITION
◇         □ADVANCE   □RVEXPO(1,30)      □REPAIRS DONE
◇         □TEST□E    □BV(OUTOK),1       □OK TO LEAVE?
◇         □SEIZE     □HELPER            □USE THE ASSISTANT
◇         □ADVANCE   □3,1               □MISC. WORK DONE
◇         □RELEASE   □HELPER            □FREE THE HELPER
◇         □LEAVE     □BAYS              □LEAVE THE BAY
◇REGULAR  □TERMINATE                    □NO, LEAVE
◇         □GENERATE  □,,,1              □DUMMY TRANSACTION
◇BACK     □ADVANCE   □RVEXPO(1,20)      □ALL RIGHT FOR SO MANY HOURS
◇         □LOGIC□S   □ALLOK             □SET SWITCH
◇         □ADVANCE   □RVEXPO(1,3.5)     □DOWN FOR A WHILE
◇         □LOGIC□R   □ALLOK             □RESET SWITCH
◇         □TRANSFER  □,BACK             □AGAIN
◇         □GENERATE  □24*365*100        □TIMER TRANSACTION ARRIVES
◇         □TERMINATE □1
◇         □START     □1
◇         □DO        □&I=1,25
◇         □PUTSTRING □('□')
◇         □ENDDO
◇         □PUTPIC    □LINES=4,N(REGULAR)/(365.*100),_
                      FR(HELPER)/10.,_
                      SR(BAYS)/10.,QA(ATREPAIR)
 TRUCKS REPAIRED PER DAY        ***.**
 UTIL. OF HELPER                ***.**%
 UTIL. OF REPAIR BAYS           ***.**%
 AVERAGE TRUCKS WAITING         ***.**
◇          □END
```

Notice how the Boolean variables INBAY and OUTOK control the movement of the trucks both in and out of the repair bays. It was necessary to simulate for such a length of time because the basic unit of time was taken as 1 hour and exponential distributions are being used in the model. The result of the simulation is

```
TRUCKS REPAIRED PER DAY       1.33
UTIL. OF HELPER              16.07%
UTIL. OF REPAIR BAYS         63.05%
AVERAGE TRUCKS WAITING        0.17
```

As can be seen, the repair facility is certainly not being overused. The person who finishes the trucks is not very busy at all.

Example 27.2

Refer to Example 27.1. Suppose that the person who works on the trucks after they are repaired is given other duties. Other items are now to be sent to the repair shop every 20 ± 8 hours. These have a higher priority than the trucks, so the person needs to work on any of them that may be waiting before he can finish a truck. Work on these other items takes 10 ± 4 hours. Determine how this change affects the repair facility.

Solution

The necessary changes are made by adding the following code:

```
◇        GENERATE   20,8,,,1          OTHER WORK FOR HELPER
◇        QUEUE      WAIT              JOIN QUEUE
◇        SEIZE      HELPER            USE THE HELPER
◇        DEPART     WAIT              LEAVE THE QUEUE
◇        ADVANCE    10,4              JOB BEING DONE
◇        RELEASE    HELPER            FREE THE HELPER
◇        TERMINATE                    LEAVE THE SYSTEM
```

The results of the simulation yield the following:

```
TRUCKS REPAIRED PER DAY     1.33
UTIL. OF HELPER           66.747%
UTIL. OF REPAIR BAYS       70.99%
AVERAGE TRUCKS WAITING      0.27
```

As can be seen, the worker is now busy 66% of the time. The additional work assigned him did not affect the repairs of the trucks.

Example 27.3

The repair facility is still not utilized fully. The owner decides to have the facility also repair trucks from another company. The owner allows 3 of these trucks to come for repairs. They take the same amount of time as the other trucks. However, they cycle back in 150 hours, exponentially distributed. Otherwise, the system remains the same. Determine the utilization of the repair area now.

Solution

The complete program is as follows:

```
◇        SIMULATE
◇        INTEGER    &I
◇        STORAGE    S(BAYS),3
◇INBAY   BVARIABLE  (LR(ALLOK))AND(SNF(BAYS))
◇OUTOK   BVARIABLE  (FS(HELPER))AND(LR(ALLOK))
```

```
◇         GENERATE    18,4.5
◇INMINE   QUEUE       ATREPAIR          JOIN QUEUE AT REPAIR SHOP
◇         TEST E      BV(INBAY),1       OK TO GO IN?
◇         ENTER       BAYS              TAKE ONE OF THREE BAYS
◇         DEPART      ATREPAIR          LEAVE THE QUEUE
◇         ADVANCE     .25               POSITION
◇         ADVANCE     RVEXPO(1,30)      REPAIRS DONE
◇         TEST E      BV(OUTOK),1       OK TO LEAVE?
◇         SEIZE       HELPER            USE THE ASSISTANT
◇         ADVANCE     3,1               MISC. WORK DONE
◇         RELEASE     HELPER            FREE THE HELPER
◇         LEAVE       BAYS              LEAVE THE BAY
◇         TEST NE     PH1,1,CYCLE       IS IT OTHER TRUCK?
◇REGULAR  TERMINATE                     NO, LEAVE
◇CYCLE    ADVANCE     RVEXPO(1,150)     YES, OTHER TRUCK, CYCLE
◇         TRANSFER    ,INMINE           BACK TO THE MINE
◇         GENERATE    ,,,1              DUMMY TRANSACTION
◇BACK     ADVANCE     RVEXPO(1,20)      ALL RIGHT FOR SO MANY HOURS
◇         LOGIC S     ALLOK             SET SWITCH
◇         ADVANCE     RVEXPO(1,3.5)     DOWN FOR A WHILE
◇         LOGIC R     ALLOK             RESET SWITCH
◇         TRANSFER    ,BACK             AGAIN
◇         GENERATE    20,8,,,1          OTHER WORK FOR HELPER
◇         QUEUE       WAIT              JOIN QUEUE
◇         SEIZE       HELPER            USE THE HELPER
◇         DEPART      WAIT              LEAVE THE QUEUE
◇         ADVANCE     10,4              JOB BEING DONE
◇         RELEASE     HELPER            FREE THE HELPER
◇         TERMINATE                     LEAVE THE SYSTEM
◇         GENERATE    ,,,3              THREE OTHER TRUCKS
◇         ASSIGN      1,1,PH            CALL THESE 1
◇         TRANSFER    ,INMINE           WORK IN THE MINE
◇         GENERATE    24*365*100        TIMER TRANSACTION ARRIVES
◇         TERMINATE   1
◇         START       1
◇         DO          &I=1,25
◇         PUTSTRING   ('  ')
◇         ENDDO
◇         PUTPIC      LINES=5,N(REGULAR)/(365.*100),N(CYCLE)/(365.*100),_
                       FR(HELPER)/10.,_
                       SR(BAYS)/10.,QA(ATREPAIR)
 REGULAR TRUCKS REPAIRED PER DAY   ***.**
 OTHER TRUCKS REPAIRED PER DAY     ***.**
 UTIL. OF HELPER                   ***.**%
 UTIL. OF REPAIR BAYS              ***.**%
 AVERAGE TRUCKS WAITING            ***.**
◇         END
```

The results of the simulation yield the following:

```
REGULAR TRUCKS REPAIRED PER DAY      1.33
OTHER TRUCKS REPAIRED PER DAY        0.34
UTIL. OF HELPER                     70.07%
UTIL. OF REPAIR BAYS                89.99%
AVERAGE TRUCKS WAITING               2.02
```

Now the repair of the owner's trucks has not changed, but the repair area is busy 89.99% of the time.

EXERCISES, CHAPTER 27

1. State what conditions need to be satisfied so that each of the following Boolean variables will be true.

```
◇COMEIN  □BVARIABLE □(R(TUGS)'LE'2)OR((Q(FIRST)'E'3)AND(FR1'GE'900)
□(a)
◇AWAY1   □BVARIABLE □R(TUGS)'LE'(V(THREE))
□(b)
◇DOWN3   □BVARIABLE □FR(MACH1)'L'500)OR(FR(MACH2)'G'500)_
                     OR(FR(MACH3)'G'200)
□(c)
◇MYTEST  □BVARIABLE □(LR(SWITCH1)AND(SNF(TUGS)))OR(&X'E'0)
□(d)
◇MYNEXT  □BVARIABLE □(LS(STOP1))AND(BV(UPTOP))
□(e)
◇MYOWN   □BVARIABLE □((S(TUGS)'E'0)*(Q(FIRST)'LE'2))+&Y'E'0
□(f)
```

2. Write an appropriate Boolean variable that will be true if the following conditions are true:
   a. Storage of BOAT1 is not empty, and the storage of BOAT2 is not full.
   b. The sum of the contents of the queue WAIT1 and the queue WAIT2 is less than the contents of the queue WAIT3.
   c. Logic switch SWIT1 is set and logic switch SWIT2 is reset or logic switch SWIT3 is set.
   d. The ampervariable &X is less than 0 and the ampervariable &Y is greater than 0 or the logic switch STOPIT is set.
   e. The halfword savevalue MIKE is 0 or both logic switches TOM3 and TOM2 are reset.
   f. The fractional utilization of facility MACHA is less than 200 permil or logic switch 1 is set.
   g. Logic switch 3 is reset or 2 times the queue length of HOLDIT is less than 6.

3. Trucks appear on the edge of a mine every 6 ± 4 minutes. Each holds 170 tons of ore. Each shift, the trucks will go to the first shovel until 10,000 tons are hauled, in which case the trucks will go to a second shovel. If the first shovel is down, the trucks will also go to the second shovel. The time to go to the first shovel is 4 ± 1 minutes, and the time to go to the second shovel is 5 ± 2 minutes. The first shovel is down every 90 ± 20 minutes and remains down for 12 ± 4 minutes. This pattern is repeated each shift of 480 minutes. Determine the daily production and the time that production shifts to the second shovel. Run the program for 10 shifts of 480 minutes each and output the average times for when the switch to the second shovel takes place and the average production from each shovel per shift. Have the results from each shift sent to a file MYOUT.OUT and have only the averages appear on the screen.
4. In exercise 3, change the program so that, when the program executes, it prompts the user to input the amount of ore the first shovel is to produce each shift before production shifts to the second shovel. After the program executes for one shift, it is to prompt the user to run again. Rather than have the program run for 10 shifts, have the results sent to the screen after each run.

SOLUTIONS, CHAPTER 27

1. a. First, the queue at FIRST is tested to see if it is equal to 3 and if the fractional utilization of the 1st facility is greater than or equal to 900. If either is true or if the remaining storage of the storage names TUGS is less than or equal to 2, the boolean variable will be true.

   b. The variable given by V(THREE) is evaluated first. If the remaining storage in the storage TUGS is less than or equal to this, the boolean variable is true.

   c. The boolean variable is true if any of the following is true:

   The fractional utilization of facility MACH1 is less than 500.

   The fractional utilization of facility MACH2 is greater than 500.

   The fractional utilization of facility MACH3 is greater than 200.

   d. First the logic switch SWITCH1 needs to be in a reset position and the storage of TUGS must not be full. The boolean variable will be true if either this is true or if the ampervariable &X is equal to 0.

   e. The boolean variable UPTOP is evaluated first. The boolean variable MYNEXT is true if this is true and the logic switch STOP! is in a set position.

   f. First, the storage of TUGS is tested to see if the current content is equal to 0 and the queue at FIRST is less than or equal to 2. If this is true or ampervariable &Y is equal to 0, the boolean variable is true. Recall that + means "OR" and * means "AND."

2. a.
```
BOATS    BVARIABLE  (SNE(BOAT1))AND(SNF(BOATS))
```
   b.
```
SUMS     VARIABLE   Q(WAIT1)+Q(WAIT2)
SUMSBV   BVARIABLE  V(SUMS)'L'Q(WAIT3)
```
   c.
```
LTEST    BVARIABLE  (LS(SWIT1))AND(LR(SWIT2)OR(LS(SWIT3)))
```
   d.
```
NUMTEST BVARIABLE  (&X'L'0)AND((&Y'G'0)OR(LS(STOPIT))
```

e. MIKES BVARIABLE (XH(MIKE)'E'0)OR(LS(TOM3)AND(LS(TOM2))

f. UTILBV BVARIABLE (FR(MACHA)'L'200)OR(LS1)

g. ARITH VARIABLE 2*Q(HOLDIT)
   SWITHBV BVARIABLE (LR3)OR(V(ARITH)'L'6)

3. The program to simulate the mine is given below:

```
         SIMULATE
         REAL        &AMT1,&AMT2,&TIME,&SUM,&PROD1,&PROD2
         INTEGER     &I,&J
MYOUT    FILEDEF     'MYOUT.OUT'
         DO          &I=1,25
         PUTSTRING   ('  ')
         ENDDO
         PUTSTRING   ('  SIMULATION IN PROGRESS....')
FIRST    BVARIABLE   (&AMT1'LE'10000)AND(LR(STOP1))
         GENERATE    6,4                   TRUCKS COME
         TEST E      BV(FIRST),1,AWAY3   WHICH SHOVEL?
         BLET        &AMT1=&AMT1+170
         TEST G      &AMT1,10000,AWAY3
         BLET        &TIME=AC1@480
         ADVANCE     4,1                   TRAVEL TO SHOVEL 1
         QUEUE       SHOVEL1               JOIN QUEUE
         SEIZE       SHOVEL1               USE THE SHOVEL
         DEPART      SHOVEL1               LEAVE THE QUEUE
         ADVANCE     2,1                   LOAD
         RELEASE     SHOVEL1               FREE THE SHOVEL
         TERMINATE                         LEAVE MINE
AWAY3    ADVANCE     5,2                   TRAVEL TO SHOVEL 2
         QUEUE       SHOVEL2               JOIN QUEUE
         SEIZE       SHOVEL2               USE THE SHOVEL
         DEPART      SHOVEL2               LEAVE THE QUEUE
         ADVANCE     2,1                   LOAD
         RELEASE     SHOVEL2               FREE THE SHOVEL
         BLET        &AMT2=&AMT2+170       ADD TO AMOUNT
         TERMINATE                         LEAVE MINE
         GENERATE    ,,,1                  DUMMY TRANSACTION
BACK     ADVANCE     90,20                 OK FOR NOW
         LOGIC S     STOP1                 SET SWITCH
         ADVANCE     12,4                  DOWN NOW
         LOGIC R     STOP1                 RESET SWITCH
         TRANSFER    ,BACK                 TO WORK
         GENERATE    480                   TIMER TRANSACTION
         TERMINATE   1                     ALL DONE
         DO          &J=1,10
         CLEAR
         START       1
         PUTPIC      LINES=3,FILE=MYOUT,&TIME,&AMT1,&AMT2
 TIME PRODUCTION WAS GREATER THAN 10000 TONS  ****.**
 AMT OF ORE FROM FIRST SHOVEL                 *****
 AMT OF ORE FROM SECOND SHOVEL                *****
         LET         &SUM=&SUM+&TIME
         LET         &PROD1=&PROD1+&AMT1
         LET         &PROD2=&PROD2+&AMT2
         LET         &AMT1=0
```

```
        LET         &AMT2=0
        LET         &TIME=0
        CLEAR
        ENDDO
        PUTSTRING   ('  ')
        PUTPIC      LINES=3,&SUM/10.,&PROD1/10.,&PROD2/10.
AVG. TIME PRODUCTION WAS GREATER THAN 10000 TONS    ***.**
AVG. AMT OF ORE FROM FIRST SHOVEL                  *****
AVG. AMT OF ORE FROM SECOND SHOVEL                 *****
        PUTSTRING   ('  ')
        PUTSTRING   (' SIMULATION OVER....')
        END
```

Notice that as soon as a truck enters the mine for shovel 1, the amount of ore mined by that shovel is increased by 170 tons. The test for the tonnage being exceeded is done as soon as truck is at the mine site.

The results of the simulation are:

```
AVG. TIME PRODUCTION WAS GREATER THAN 10000 TONS    394.11
AVG. AMT OF ORE FROM FIRST SHOVEL                  10030
AVG. AMT OF ORE FROM SECOND SHOVEL                 13124
```

4. The changes are easy to make. Add an ampervariable &CHANGE to the program and replace the 10000 with it. At the end of the program the code will be:

```
        PUTPIC      LINES=3,&CHANGE,&TIME,&AMT1,&AMT2
TIME PRODUCTION WAS GREATER THAN ****** TONS   ***.**
AMT OF ORE FROM FIRST SHOVEL                  *****
AMT OF ORE FROM SECOND SHOVEL                 *****
        PUTSTRING   ('  ')
        PUTSTRING   (' DO AGAIN? (Y/N)')
        PUTSTRING   ('  ')
        GETLIST     &ANS
        IF          (&ANS'E''Y')OR(&ANS'E''y')
        LET         &AMT1=0
        LET         &AMT2=0
        LET         &TIME=0
        CLEAR
        GOTO        AGAIN
        ENDIF
        PUTSTRING   ('  ')
        PUTSTRING   (' SIMULATION OVER....')
        END
```

The first time the program is run for a change over at 8000 tons the output is:

```
TIME PRODUCTION WAS GREATER THAN 8000 TONS    310.52
AMT OF ORE FROM FIRST SHOVEL                 8160
AMT OF ORE FROM SECOND SHOVEL               13770
```

However, the program needs to be run multiple times and the averages values used as shown by the results of Exercise 26.3

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 28

The *BUFFER* Block

THE BUFFER BLOCK

To fully appreciate the way a GPSS/H program works, it is important to always understand the way transactions are moved on the various chains. Recall that, once a transaction is moved by the processor, it will be moved forward as far as possible until one of three things happens to it: (1) it is terminated, (2) it is put on the FEC, or (3) it is blocked. When one of these three things happens, the processor will start a rescan. So far, this procedure is what we have wanted. However, there are times when it is necessary to start a rescan of the CEC before the active transaction has come to a rest. Such a necessity can, perhaps, best be understood by considering a short example.

Suppose a delivery company has 5 trucks in its fleet. Each has a daily availability of 70% because some are used in other parts of the company or some are in the shop for maintenance and/or repairs. At the start of each day, it is desired to determine how many are available. If 5 are available, they will be allocated a certain way; if 4 are available, they will be allocated a different way, etc. The program lines for this might be (it will be explained why the PRIORITY 0 block is added):

```
HOWMANY  FUNCTION   RN1,D2
.7,OKTRK/1,BYEBYE
WHERE    FUNCTION   NTRUCK,L5
1,BLOCKA/2,BLOCKB/3,BLOCKC/4,BLOCKD/5,BLOCKE
         GENERATE   ,,,5,1
         PRIORITY   0
         TRANSFER   ,FN(HOWMANY)
OKTRK    BLET       &NTRUCK=&NTRUCK+1
         -----      blank for now
         TRANSFER   ,FN(WHERE)
```

Five truck transactions are generated at time $t = 0$. Each has priority = 1. Once a transaction leaves the GENERATE block, its priority is reduced to 0. It then enters the block TRANSFER ,FN(HOWMANY). The function HOWMANY will determine whether the truck transaction will be available for the day or not. If it

is to be available, the next block, OKTRK BLET &NTRUCK=&NTRUCK+1, keeps a count of the number of trucks that are available. At the point when a transaction enters the TRANSFER ,FN(WHERE) block, you want the transaction to be transferred to different parts of the program depending on how many trucks are available. These are indicated by the block labels, BLOCKA, BLOCKB, etc. Suppose the first transaction leaves the GENERATE block and is routed to the block labeled OKTRK. It will increment the count of NTRUCK to 1 and then go to the next sequential block. If this is the TRANSFER ,FN(WHERE) block, it will be routed to BLOCKA, which is where you want the transactions to go if there is only one truck available. The next thing to do is to have the second truck transaction leave the GENERATE block. However, the first transaction is still active and will be moved to the next sequential block from BLOCKA. Since there are more trucks to be tested, this approach is incorrect. The way to handle this is to force a rescan of the CEC at this point and move the other truck transactions that are residing in the GENERATE block. This procedure is exactly what the BUFFER block does. Its general form is

```
◇          BUFFER
```

There are no operands. When a transaction enters this block, it causes a rescan of the CEC. It has no other purpose.

Let us see how the presence of the block BUFFER would work for the example above. Imagine that there is a BUFFER block where the comment says "blank for now." The processor starts a rescan. The transaction in the BUFFER block is part of this scan. Its priority is 0. The second truck transaction in the GENERATE ,,,5,1 block has a priority of 1 so it is moved first. Thus, the first transaction will be held at the BUFFER block until the second transaction has moved forward. How far is it moved? Just as far as the BUFFER block, which triggers another rescan. Now, the third transaction is moved forward from the GENERATE ,,,5,1 block as far as the BUFFER block. This process is repeated until all 5 of the transactions have left the GENERATE block. Now, when each transaction enters the TRANSFER ,FN(WHERE) block, the correct count of the available trucks in the system for the day is used to allocate them. A BUFFER block should be used whenever an action in one part of a program triggers an action in another segment. For example, a ship may be in a harbor waiting to be loaded, and it takes 12 truckloads. Whenever a truck dumps a load, the counter &LOADS is incremented by 1. After every load is dumped, it is necessary to check to see if the ship is loaded and can leave the harbor. A BUFFER block will accomplish this task. The ship transaction needs to have a higher priority than the truck transaction, of course.

Example 28.1

Customers come to a small repair shop in a plant with up to 4 items for service. The probability of a given customer bringing 1 item is 50%; 2 items, 30%, 3 items, 15%; and 4 items, 5%. There are 4 identical workers to assist the customers. Each will service 1 customer at a time. The people will always wait while the items are being repaired. The repairs require a time obtained by sampling from the normal distribution with a mean of 8 and a standard deviation of 1.4 for each item. Before each repair begins, there is some paperwork to be filled out, which takes 1 ± 0.5 minute, and it takes the same time

to check the items after they are repaired. It costs the company $25.25 per hour for each worker who must wait for the repairs since these people are getting paid and not producing anything. The workers, however, only receive the minimum wage of $6.25 per hour. Simulate for 4, 5, and then 6 workers to determine if another worker should be added. Use a DO loop. In addition to determining the cost of the delays, the program should also determine the following: (1) How long each customer needs to wait before a worker is free. (2) How long each customer is in the shop. (3) How busy the workers are. Simulate for 100 shifts of 480 minutes each and for 3 shifts per day. Have the results sent to a data file.

Solution

```
         SIMULATE
         RMULT     65432
         INTEGER   &SERVER,&TOOLS,&I,&REPAIRS
MYOUT    FILEDEF   'MYOUT.OUT'
         DO        &I=1,25
         PUTSTRING (' ')
         ENDDO
WAITUP   EQU       20,L               NECESSARY!!
HOWMANY  FUNCTION  RN1,D4
.5,1/.8,2/.95,3/1,4
         PUTSTRING (' SIMULATION IN PROGRESS....')
COMEIN   GENERATE  RVEXPO(1,5),,,,10
         ASSIGN    2,FN(HOWMANY),PH
         QUEUE     WAIT
         QUEUE     TOTAL
         GATE LS   WAITUP             WAIT FOR FREE WORKER
         DEPART    WAIT
         LOGIC R   WAITUP
         ASSIGN    1,&SERVER,PH       SERVER NUMBER IS ASSIGNED
         BLET      &TOOLS=PH2
         GATE LS   PH1
         DEPART    TOTAL
         LOGIC R   PH1
         TERMINATE
WORKER   GENERATE  ,,,&REPAIRS        NO. OF REPAIRMEN
         ASSIGN    1,N(WORKER),PH
UPTOP    TEST G    Q(WAIT),0          ANYONE WAITING?
         ENTER     WORKERS            START WORKING
         BLET      &SERVER=PH1
         LOGIC S   WAITUP             LET PERSON THROUGH
         BUFFER                       GO TO OTHER SEGMENT
         ASSIGN    2,&TOOLS,PH
         ADVANCE   1,.5               PAPER WORK
BACKUP   ADVANCE   RVNORM(1,8,1.4)    MAKE REPAIRS
         LOOP      2PH,BACKUP
         ADVANCE   1,.5               MORE PAPER WORK
         LOGIC S   PH1
         BUFFER
         LEAVE     WORKERS
         TRANSFER  ,UPTOP
         GENERATE  480*100*3
```

```
◇       □TERMINATE □1
◇       □DO        □&REPAIRS=4,6
◇       □CLEAR
◇       □STORAGE   □S(WORKERS),&REPAIRS
◇       □START     □1
◇       □PUTPIC    □LINES=8,FILE=MYOUT,&REPAIRS,N(COMEIN)/(500.*3),_
                    QT(WAIT),QA(WAIT),_
                    QT(TOTAL),QA(TOTAL),SR(WORKERS)/10.,_
                    25.25*QA(TOTAL)*8+6.25*&REPAIRS*8
 NUMBER OF WORKERS                    *
 PEOPLE WHO COME IN PER SHIFT      ****.**
 AVERAGE TIME WAITING FOR SERVER    ***.**
 AVERAGE NO. WAITING FOR SERVER     ***.**
 AVERAGE TIME IN SHOP               ***.**
 AVERAGE NO. IN SHOP                ***.**
 UTIL. OF WORKERS                   ***.**%
 COST PER SHIFT OF THIS             ***.**
◇       □PUTSTRING □('□')
◇       □PUTSTRING □('□')
◇       □ENDDO
◇       □END
```

There are several important features of the program that will be discussed here. The workers come into the system via the GENERATE RVEXPO(1,5),,,,10 block. They are assigned the number of parts to be repaired, and this value is placed in their second halfword parameter. They are held in the GATE block WAITUP until a worker is free. There are 4 workers. Each is numbered, and they are held in the block TEST G Q(WAIT),0 until there is someone in the block QUEUE WAIT. Once this is the case, the worker enters a busy status. The ampervariable &SERVER is set equal to the number of workers. The logic switch WAITUP is placed in a set position, which corresponds to the first waiting person being served. Next the BUFFER block is entered and a rescan of the CEC is performed. The transaction being held in the GATE WAITUP block has a priority of 10, so it is moved next while the transaction that caused the rescan is held in the BUFFER block. This transaction moves, leaves the queue, places the logic switch WAITUP in a reset position so that only 1 customer can be served at a time, and sets its first halfword parameter equal to the worker number. It then sets the variable &TOOLS equal to its second parameter. This parameter has the number of parts to be repaired. It then is held in the block GATE LS PH1. Since the logic switch corresponding to the first parameter is in a reset position, the transaction will remain at this block, and control passes to the worker segment. Here the worker repairs the parts and then places the logic switch that corresponds to his or her number (from 1 to 4) in a set position. The transaction enters the BUFFER block, which causes a rescan of the CEC. The customer who is waiting for the worker to finish then leaves the shop, which places the logic switch PH1 in a reset position to delay the next customer who brings the next repair item(s) to the same worker. Notice that it was necessary to have the statement WAITUP EQU 20,L. The output from running the program is as follows:

```
NUMBER OF WORKERS                 4
PEOPLE WHO COME IN PER SHIFT     19.05
AVERAGE TIME WAITING FOR SERVER   7.47
AVERAGE NO. WAITING FOR SERVER    1.48
AVERAGE TIME IN SHOP             23.51
AVERAGE NO. IN SHOP               4.67
UTIL. OF WORKERS                 79.61%
COST PER SHIFT OF THIS         1142.69

NUMBER OF WORKERS                 5
PEOPLE WHO COME IN PER SHIFT     19.12
AVERAGE TIME WAITING FOR SERVER   1.48
AVERAGE NO. WAITING FOR SERVER    0.29
AVERAGE TIME IN SHOP             17.46
AVERAGE NO. IN SHOP               3.48
UTIL. OF WORKERS                 63.67%
COST PER SHIFT OF THIS          952.57

NUMBER OF WORKERS                 6
PEOPLE WHO COME IN PER SHIFT     19.34
AVERAGE TIME WAITING FOR SERVER   0.46
AVERAGE NO. WAITING FOR SERVER    0.09
AVERAGE TIME IN SHOP             16.43
AVERAGE NO. IN SHOP               3.31
UTIL. OF WORKERS                 53.59%
COST PER SHIFT OF THIS          968.47
```

As can be seen, it would be profitable to hire a fifth repair person.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 29

The SPLIT Block

CLONES OF TRANSACTIONS

Transactions have been placed in our simulations by the GENERATE block. In fact, this is the only way to create original transactions. However, once a transaction has been created, it is possible to make clones of it. Such clones will normally be identical to the original transactions, although they can be made to differ. The clones will always be identical to the original transactions in their priority level and their time of entry, which is called their "Mark Time." This latter point is worth noting. If the original transaction entered the simulation at time $t = 2050$ and a new transaction was cloned at time $t = 3500$, the clone has a Mark Time of 2050, *not* 3500. The clones will normally have the same number and type of parameters as the original, but in these particulars, the clones can be made to differ.

The block that creates these clones is the SPLIT block. The form to create identical transactions is

```
◇        SPLIT     A,B
```

where operand *A* is the number of clones to create and operand *B* is the label of the block the transactions are routed to.

When a transaction enters a SPLIT block, *A* identical transactions are created and leave the block one at a time (incrementing the block count as they leave). These are all routed to the destination block whose label is specified in the *B* operand of the SPLIT block. The original transaction is *not* routed to this destination block but goes to the next sequential block. In fact, this original transaction is moved before the clones are. Some examples of the SPLIT block are

```
◇(a)     SPLIT     1,DOWN1
◇(b)     SPLIT     10,UPTOP
```

In (a), one clone is created and sent to the block with the label DOWN1. In (b), 10 clones are created and sent to the block with the label UPTOP. In both cases, the original transactions are routed to the next sequential block.

Often it is desired to have the original transaction and the clones routed to the same block. This can be done by making the next sequential block the one where the clones are sent to, e.g.,

```
         SPLIT     3,NEXT1
NEXT1    -----     -----
```

Here the original and the 3 clones are all sent to the same block.

SPLIT blocks can be very handy in programming problems in which a single unit comes along and several things have to be done on different parts of the unit simultaneously. For example, suppose a partially completed car comes along an assembly line. At a given point, one person makes an adjustment to the front, another connects the bumper to the rear, and a third bolts on the steering wheel. It is convenient to split the car transaction by making 2 clones and then to work on the clones separately. Only when all 3 transactions (the original and the 2 clones) are finished can the car be moved along. How to do this will be covered in Chapter 30, when the MATCH block is presented.

Example 29.1

In a manufacturing process, parts come along an assembly line every 4 ± 1.2 minutes. An overhead crane is used to lift them from the line to another section where they will be worked on further. It takes 2 ± 0.8 minutes to load and transport the parts. The crane then must return to its original position. This takes 1.6 ± 0.3 minutes. Give the GPSS/H code to model this segment of the system.

Solution

```
         GENERATE   4,1.2       PARTS COME ALONG
         QUEUE      WAIT        JOIN QUEUE
         SEIZE      CRANE       USE THE CRANE
         DEPART     WAIT        LEAVE THE QUEUE
         ADVANCE    2,.8        MOVE PART TO NEW SECTION
         SPLIT      1,DOWN1     CREATE CLONE
         ADVANCE    1.6,.3      MOVE CRANE BACK
         RELEASE    CRANE       FREE CRANE
         TERMINATE              REMOVE TRANSACTION
DOWN1    -----
         -----
```

Notice that, after the crane moved the part, the SPLIT block formed a clone of the part, and the clone was sent to the block DOWN1. The original part transaction entered the block ADVANCE 1.6,.3 to represent the travel time for the crane to return to the original position. When the "original part" transaction leaves the ADVANCE block, it releases the crane. Only then can another part be moved by the crane.

Example 29.2

A part in a large machine will periodically fail and have to be replaced with a spare part while the failed part is fixed. When this happens, the machine is shut down, and the failed part is removed. The machine is then down until one of the following two things happen:

1. A good part is available to replace the failed part, and, once this is installed, the machine starts again.
2. No spare part is available, and the machine must wait until the failed part (or parts) are repaired.

The life of a part is 108 hours, exponentially distributed. The time to remove a failed part is normally distributed with a mean of 7 hours and a standard deviation of 1 hour. The time to put a repaired part in the machine is also normally distributed with a mean of 12 hours and a standard deviation of 1.1 hours. A single repair person fixes failed parts, and a single machinist runs the machine. The time to repair a failed part is obtained from sampling the uniform distribution 10 ± 4 hours. Each hour the machine is down costs the company $15.25. Each spare part costs the company $125 per week to keep in stock. The company works 7 days a week, 24 hours a day. The repair shop works for 20 hours and then is closed for 4 hours. Run the simulation for 500 years and for the following numbers of spares: 0, 1, 2, and 3. Determine the optimum number of spares to have. Have the results sent to an output file.

Solution

```
        SIMULATE
        INTEGER    &I,&SPARES
        REAL       &COST
MYOUT   FILEDEF    'EXAM29B.OUT'
        GENERATE   ,,,1               DUMMY TRANSACTION
BACKIN  SEIZE      MACH1
        ADVANCE    RVEXPO(1,108)      WORKING MACHINE
        RELEASE    MACH1
        ADVANCE    RVNORM(1,7,1)      TAKE PART OUT
        SPLIT      1,REPAIR           MAKE A CLONE
        SEIZE      FIXER
        ADVANCE    12,4                FIX THE PART
        RELEASE    FIXER
        BLET       &SPARES=&SPARES+1  ADD ONE TO SPARES
        TERMINATE
REPAIR  TEST G     &SPARES,0          ANY SPARES AVAILABLE?
        GATE LR    NOTWORK
        BLET       &SPARES=&SPARES-1
        ADVANCE    RVNORM(1,12,1.1)   PUT SPARE IN MACHINE
        TRANSFER   ,BACKIN
        GENERATE   ,,,1
```

```
BACKUP  ADVANCE    20
        LOGIC S    NOTWORK
        ADVANCE    4
        LOGIC R    NOTWORK
        TRANSFER   ,BACKUP
        GENERATE   24*7*52*500          SIMULATE FOR 500 YEARS
        BLET       &COST=15.25*(1-FR(MACH1)/1000.)*24*7+(&I-1)*125.
        TERMINATE  1
        DO         &I=1,4
        LET        &SPARES=&I-1
        START      1
        PUTPIC     FILE=MYOUT,LINES=8,&I-1,FR(MACH1)/10.,FR(FIXER)/10.,_
                     FC(MACH1)/500.,_
                     FC(FIXER)/500.,&COST

 NUMBER OF SPARE PARTS                                  *
 UTIL. OF MACHINE                                    ***.**%
 UTIL. OF REPAIR PERSON                              ***.**%
 NUMBER OF BREAKDOWNS/YR.                            ***.**
 NUMBER OF TIMES REPAIR PERSON USED/YR.              ***.**
 EXPECTED COST PER DAY                              ****.**

        LET        &COST=0
        CLEAR
        ENDDO
        END
```

The program represents the machinist as a single transaction. This transaction simply turns the machine on and off. When a part fails, the machine is turned off and the part removed. A SPLIT block is used to create a clone. The original transaction is sent to the repair shop. Once it is repaired, the number of spare parts is incremented by 1 and the original transaction is terminated. The clone tests to see if a spare part is available. If so, the repair person immediately starts to put one in the machine. If no part is available, the repair person has to wait for one to be repaired.

The segment with the repair shop being down for 4 hours after working for 20 hours is easily handled with the combination of a LOGIC switch and a GATE block. The output of the simulation is as follows:

```
NUMBER OF SPARE PARTS                      0
UTIL. OF MACHINE                          77.55%
UTIL. OF REPAIR PERSON                     8.60%
NUMBER OF BREAKDOWNS/YR.                  62.56
NUMBER OF TIMES REPAIR PERSON USED/YR.    62.56
EXPECTED COST PER DAY                     75.11

NUMBER OF SPARE PARTS                      1
UTIL. OF MACHINE                          84.75%
UTIL. OF REPAIR PERSON                     9.47%
NUMBER OF BREAKDOWNS/YR.                  68.94
NUMBER OF TIMES REPAIR PERSON USED/YR.    68.94
EXPECTED COST PER DAY                    515.71
```

```
NUMBER OF SPARE PARTS                          2
UTIL. OF MACHINE                           84.70%
UTIL. OF REPAIR PERSON                      9.48%
NUMBER OF BREAKDOWNS/YR.                   69.09
NUMBER OF TIMES REPAIR PERSON USED/YR.     69.09
EXPECTED COST PER DAY                     641.97

NUMBER OF SPARE PARTS                          3
UTIL. OF MACHINE                           84.86%
UTIL. OF REPAIR PERSON                      9.40%
NUMBER OF BREAKDOWNS/YR.                   68.42
NUMBER OF TIMES REPAIR PERSON USED/YR.     68.42
EXPECTED COST PER DAY                      62.86
```

As can be seen, the optimum number of spare parts to have is 1. Having more spares than this number does not change the availability of the machine as it remains around 85%.

Example 29.3

In the previous example, the repair person was busy only about 9.5% of the time. Suppose that this person also had duties to repair other items. Other failed parts come along every 20 ± 6 minutes, and these take 18 ± 2.1 minutes to repair. Since the repair shop will be down for 4 hours each day, it is necessary to give these failed parts a higher priority than the parts from the machine. This procedure guarantees that, when the repair shop is down, these "other" parts will be repaired first when the shop reopens. If this is not the case, when it starts up again, the queues in front of it may be too large, and the program will blow up.

Solution

The only segment that must be added to the program is as follows (this code forms a complete segment and so can be inserted anywhere in the program after a TERMINATE block or before a GENERATE block):

```
        GENERATE   20,6,,,10     OTHER PARTS
        QUEUE      WAIT
        SEIZE      FIXER
        DEPART     WAIT
        ADVANCE    18,2.1
        RELEASE    FIXER
        TERMINATE
```

Selected parts of the output from running the program are:

Spare Parts	Util. of Machine	Util. of Repair Shop	Cost
0	53.50%	96.50%	2334
1	64.27%	98.03%	1918
2	69.53%	98.50%	1779
3	72.48%	98.81%	1757
4	73.90%	99.10%	1801
5	74.03%	99.87%	1889

As can be seen, the number of spare parts to have is now 3. The utilization of the machine is quite low, and so, in practice, something should be done to improve the repair-shop times.

SIMULATING 24-HOUR (OR "MILITARY") TIME

The absolute clock measures the simulation time from the start of the simulation. In many simulations, however, it is also necessary to keep track of the simulation time, the day of the simulation, the week, etc. For example, it may be necessary to have the workers stop for 30 minutes at noon for lunch, to shut down the mine for the third shift (but keep the mill working), or to shut a crusher down for 1 hour of maintenance every day at 4:00 p.m.

If a time value corresponding to what is known as "military clock time" or "24-hour time" is used, many of these stoppages can easily be modeled, often much more easily than by using the absolute clock. Some examples comparing 12-hour time, military time, and GPSS/H simulation time are

	Midnight	9:30 A.M.	Noon	2:30 P.M.	Midnight – 1 minute
12-hour time	12:00 P.M.	9:30 A.M.	12:00 A.M.	2:30 P.M.	11:59 P.M.
Military time	0000	0930	1200	1430	2359
Simulation time	0 hours	930 hours	1200 hours	1430 hours	2359 hours

This conversion of the clock time is often much easier to program compared to using the absolute clock. For example, if the miners take their lunch break every day at noon, this is 1200 hours. If the mine does not work on the weekend, then the days to shut down the mining part of the simulation will be days 6 and 7.

Example 29.4

Write a program to give the military time for 2 days. Have output every hour and place this output in a data file. Use a basic time unit of 1 minute. The data file should give the absolute clock time, the military time, and the day of the simulation. The program should also count the day of the week (from 1 to 7) as well as the number of the week. (This example was kindly supplied by Robert Crain of Wolverine Software.)

Solution

```
         SIMULATE
         INTEGER    &MILITIME         Current time in 24-hour format
         INTEGER    &DAYNO            Consecutive days are numbered
         INTEGER    &WKDAYNO          Week days 1-7
         INTEGER    &WEEKNO           Weeks are numbered
MYOUT    FILEDEF    'MCLOCK.OUT'
*
***********************************************************************
```

```
◇        GENERATE  ,,,1,150                  1 master clock Xact
*
◇        BLET      &WKDAYNO=1                Start on Monday
*
◇NEXTMIN ADVANCE   1                         Clock "ticks" each minute
◇        BLET      &MILITIME=&MILITIME+1     Add 1 minute to 4-digit time
*
*                  To close mine for lunch and weekends,
*                  code is added here; see explanation that
*                  follows the output.
*
◇        TEST E    &MILITIME@100,60,NEXTMIN  Last 2 digits = 60?
◇        BLET      &MILITIME=&MILITIME+40    YES => Add 40
◇        BPUTPIC   FILE=MYOUT,&WKDAYNO,AC1,&MILITIME
DAY NUMBER ***    SIMULATION TIME ****   MILITARY TIME ****
◇        TEST E    &MILITIME,2400,NEXTMIN    Have we reached 2400?
◇        BLET      &MILITIME=0000            YES => reset to 0000
◇        BLET      &DAYNO=&DAYNO+1           Increment total days
◇        BLET      &WKDAYNO=&WKDAYNO+1       Increment day of week (1-7)
◇        TEST E    &WKDAYNO,8,NEXTMIN        Is this new Monday? NO
◇        BLET      &WKDAYNO=1                YES => reset day of week to 1
◇        BLET      &WEEKNO=&WEEKNO+1         Increment total weeks
◇        TRANSFER  ,NEXTMIN                  Tick again
*************************************************************************
◇        GENERATE  480*3*2                   Simulate for 2 days.
◇        TERMINATE 1
◇        START     1
◇        END
```

The program starts by setting the day counter, &WKDAYNO, equal to 1, and the clock is assumed to start at midnight. The clock repeatedly advances by 1 minute until time $t = 60$ is reached, which is 1 o'clock in the morning. At this time, the military clock is to be at time $t = 0100$. "Setting" the military clock is done by using modular arithmetic. The military clock is advanced every 60 basic time units (= minutes) until 1 day (24 hours) has passed. At this time, the military clock reads 2400 hours, so the day counter is incremented and the military clock is set back to 0000. The following is sample output from the program:

```
DAY NUMBER  1   SIMULATION TIME  60   MILITARY TIME  100
DAY NUMBER  1   SIMULATION TIME 120   MILITARY TIME  200
DAY NUMBER  1   SIMULATION TIME 180   MILITARY TIME  300
DAY NUMBER  1   SIMULATION TIME 240   MILITARY TIME  400
DAY NUMBER  1   SIMULATION TIME 300   MILITARY TIME  500
DAY NUMBER  1   SIMULATION TIME 360   MILITARY TIME  600
DAY NUMBER  1   SIMULATION TIME 420   MILITARY TIME  700
DAY NUMBER  1   SIMULATION TIME 480   MILITARY TIME  800
DAY NUMBER  1   SIMULATION TIME 540   MILITARY TIME  900
DAY NUMBER  1   SIMULATION TIME 600   MILITARY TIME 1000
DAY NUMBER  1   SIMULATION TIME 660   MILITARY TIME 1100
DAY NUMBER  1   SIMULATION TIME 720   MILITARY TIME 1200
```

```
DAY NUMBER   1   SIMULATION TIME  780   MILITARY TIME 1300
DAY NUMBER   1   SIMULATION TIME  840   MILITARY TIME 1400
DAY NUMBER   1   SIMULATION TIME  900   MILITARY TIME 1500
DAY NUMBER   1   SIMULATION TIME  960   MILITARY TIME 1600
DAY NUMBER   1   SIMULATION TIME 1020   MILITARY TIME 1700
DAY NUMBER   1   SIMULATION TIME 1080   MILITARY TIME 1800
DAY NUMBER   1   SIMULATION TIME 1140   MILITARY TIME 1900
DAY NUMBER   1   SIMULATION TIME 1200   MILITARY TIME 2000
DAY NUMBER   1   SIMULATION TIME 1260   MILITARY TIME 2100
DAY NUMBER   1   SIMULATION TIME 1320   MILITARY TIME 2200
DAY NUMBER   1   SIMULATION TIME 1380   MILITARY TIME 2300
DAY NUMBER   1   SIMULATION TIME 1440   MILITARY TIME 2400
DAY NUMBER   2   SIMULATION TIME 1500   MILITARY TIME  100
DAY NUMBER   2   SIMULATION TIME 1560   MILITARY TIME  200
DAY NUMBER   2   SIMULATION TIME 1620   MILITARY TIME  300
DAY NUMBER   2   SIMULATION TIME 1680   MILITARY TIME  400
DAY NUMBER   2   SIMULATION TIME 1740   MILITARY TIME  500
DAY NUMBER   2   SIMULATION TIME 1800   MILITARY TIME  600
DAY NUMBER   2   SIMULATION TIME 1860   MILITARY TIME  700
DAY NUMBER   2   SIMULATION TIME 1920   MILITARY TIME  800
DAY NUMBER   2   SIMULATION TIME 1980   MILITARY TIME  900
DAY NUMBER   2   SIMULATION TIME 2040   MILITARY TIME 1000
DAY NUMBER   2   SIMULATION TIME 2100   MILITARY TIME 1100
DAY NUMBER   2   SIMULATION TIME 2160   MILITARY TIME 1200
DAY NUMBER   2   SIMULATION TIME 2220   MILITARY TIME 1300
DAY NUMBER   2   SIMULATION TIME 2280   MILITARY TIME 1400
DAY NUMBER   2   SIMULATION TIME 2340   MILITARY TIME 1500
DAY NUMBER   2   SIMULATION TIME 2400   MILITARY TIME 1600
DAY NUMBER   2   SIMULATION TIME 2460   MILITARY TIME 1700
DAY NUMBER   2   SIMULATION TIME 2520   MILITARY TIME 1800
DAY NUMBER   2   SIMULATION TIME 2580   MILITARY TIME 1900
DAY NUMBER   2   SIMULATION TIME 2640   MILITARY TIME 2000
DAY NUMBER   2   SIMULATION TIME 2700   MILITARY TIME 2100
DAY NUMBER   2   SIMULATION TIME 2760   MILITARY TIME 2200
DAY NUMBER   2   SIMULATION TIME 2820   MILITARY TIME 2300
DAY NUMBER   2   SIMULATION TIME 2880   MILITARY TIME 2400
```

Notice that the week day and the week number were not output. However, for a simulation that ran for a longer time, they would be used. This simulation took place for only 2 days in order to illustrate the nature of the military code segment. If this segment is to be used in an actual program, the following change would be needed: (1) The BPUTPIC can be removed to reduce the program output, and (2) the timer transaction segment

```
◇        GENERATE  480*3*2                Simulate for 2 days.
◇        TERMINATE 1
```

should be removed as the main program will already have a timer transaction to control the length of the simulation.

Example 29.4, although very useful for simulation in general, did not contain any SPLIT blocks. However, suppose that one wanted to shut down a part of the program for ½ hour (30 time units) while the miners had a lunch break

every weekday at noon and the miners did not work on Saturday or Sunday. The main program needs blocks GATE LR LUNCH and GATE LR WEEKEND to control these. For example, the main program where the mining is taking place might have these GATE blocks just before the blocks that model the crusher. When the truck transactions arrive at the crusher, the first must check to see if these GATE blocks are open or shut. A skeleton of the program might be

```
◇         SIMULATE
◇         -----
◇         -----
◇         GATE LR    LUNCH     IS IT LUNCH TIME?
◇         GATE LR    WEEKEND   IS IT A WEEKEND?
◇         QUEUE      CRUSHER   JOIN QUEUE AT CRUSHER
◇         -----
◇         -----
```

These two GATE blocks will stop the truck transactions from moving through the system at lunch and during the weekends when the mining is to be stopped.

The changes to the program are shown next. Rather than repeat the whole program, only the new code is given, as well as the old code above and below:

```
◇         -----
◇         -----
◇         BLET       &MILITIME=&MILITIME+1      Add 1 minute to 4-digit time
◇         TEST E     &MILITIME,1200,NEXT1
◇         SPLIT      1,WORKSHUT
◇NEXT1    TEST E     &MILITIME@100,60,NEXTMIN   Last 2 digits = 60?
◇         BLET       &MILITIME=&MILITIME+40     YES => Add 40
◇         BPUTPIC    FILE=MYOUT,&WKDAYNO,AC1,&MILITIME
DAY NUMBER ***    SIMULATION TIME ****    MILITARY TIME ****
◇         TEST E     &MILITIME,2400,NEXTMIN     Have we reached 2400?
◇         BLET       &MILITIME=0000             YES => reset to 0000
◇         BLET       &DAYNO=&DAYNO+1            Increment total days
◇         BLET       &WKDAYNO=&WKDAYNO+1        Increment day of week (1-7)
◇         TEST E     &WKDAYNO,6,NEXT2
◇         SPLIT      1,WEEKDONE
◇NEXT2    TEST E     &WKDAYNO,8,NEXTMIN         Is this new Monday? NO
◇         BLET       &WKDAYNO=1                 YES => reset day of week to 1
◇         -----
◇         -----
◇WORKSHUT LOGIC S    LUNCH
◇         ADVANCE    30                         SHUT FOR 1/2 HOUR LUNCH
◇         LOGIC R    LUNCH
◇         TERMINATE
◇WEEKDONE LOGIC S    WEEKEND
◇         ADVANCE    480*3*2                    SHUT FOR TWO DAYS
◇         LOGIC R    WEEKEND
◇         TERMINATE
◇         TEST E     &MILITIME@100,60,NEXTMIN   Last 2 digits = 60?
◇         -----
◇         -----
```

When the military clock is at 1200 (noon), the original transaction enters the SPLIT block, and 1 clone is created; the clone is routed to the block labeled WORKSHUT. The associated logic switch LUNCH is placed in a set position. The clone is held there for 30 time units, and then the logic switch LUNCH is reset. A similar thing happens when the day is number 6 (= Saturday). The original transaction enters a SPLIT block; the clone produced is then routed to the block with the label WEEKDONE. The entry of the clone places the associated logic switch WEEKEND in a set position and holds it there for 480*3*2 time units (= 2 days).

GENERAL FORM OF THE SPLIT BLOCK

The most general form of the SPLIT block is

```
◇        SPLIT     A,B,C,D,E,F,G
```

As before, operand *A* is the number of clones to create, and operand *B* is the label of the block the transactions are routed to. Operand *C* is an integer parameter (PH, PB, or PF) and is called the *serialization option*; it allows one to place sequential numbers in the parameter identified in operand *C* in *both* the original transaction and the clones. The original transaction arrives at the SPLIT block with a previously determined value in the parameter specified by the *C* operand. The original transaction leaves the SPLIT block with this value incremented by 1. The first clone leaves with this value incremented by 2, etc. GPSS/H increments the parameter of each succeeding clone by 1.

Operands *D*, *E*, *F*, and *G* are the type and number of parameters each of the clones will have. They will normally have the same values as the original transaction, unless the serialization option is used. If more parameters are specified than the original transaction had, each will have the value zero. Most of the time operands *D*–*G* are not used.

An example of the *C* operand is:

```
◇        SPLIT     1,NEXT,3PH
```

Suppose that the value in the third halfword parameter was 4. Then the original transaction would have the value 5 (increment by 1) in its third halfword parameter, and the clone would have the value 6 in its third halfword parameter. In another example,

```
◇        SPLIT     5,AWAY,2PH
```

suppose that the value in the transaction's second halfword parameter was 0. Then the values placed in each of the various transactions would be

Transaction	Value in Second Halfword Parameter
original	1
1st clone	2
2nd clone	3
3rd clone	4
4th clone	5
5th clone	6

If one tried to use code such as

```
◇         □SPLIT     □2,BBBB,2PL
```

a compiling error would result because only PH, PF, and PB parameters are allowed.

Two examples in which the number and type of parameters are changed are

```
◇         □SPLIT     □3,UPTOP,,1PH,2PL,3PB,4PF
◇         □SPLIT     □2,AWAY1,3PH,12PH,12PL
```

This option is rarely used, as it is good programming practice to make each transaction—even the timer transaction—have the same number and type of parameters.

EXERCISES, CHAPTER 29

1. State what each of the following SPLIT blocks do:

```
◇         □SPLIT     □3,DDDD                       □(a)
◇         □SPLIT     □4,AWAY,3PH                   □(b)
◇         □SPLIT     □1,BACK,5PB,1PH,10PB          □(c)
◇         □SPLIT     □6,MARY,,3PH,3PL,3PB,3PF      □(d)
◇BLOCK    □SPLIT     □5,FN(WHERE)                  □(e)
```

The function WHERE is defined as

```
◇WHERE    □FUNCTION  □N( BLOCK)@5+1,L5
1,BLOCKA/2,BLOCKB/3,BLOCKC/4,BLOCKD/5,BLOCKE
```

2. Consider the following code:

```
◇         □GENERATE  □,,,1
◇         □ASSIGN    □1,3,PH
◇         □ASSIGN    □2,5,PH
◇         □SPLIT     □2,AWAY,2PH
◇AWAY     □ADVANCE   □0
```

What number and types of parameters will the transactions have when they enter the ADVANCE block? What are the values in each parameter?

3. Parts come along a conveyor belt every 10 ± 4.5 minutes. A single worker takes 8 ± 3.6 minutes to work on them. When she is done, she attaches each to an overhead crane that carries them to the next work station. As soon as they arrive at this station, there are two workers who take 18 ± 7 minutes to finish the parts. The crane takes 2 ± 0.5 minutes to move from the single worker to the finishing station and then takes 1.5 ± 0.25 minutes to return. Model this system for 1 day's production. Determine how busy the workers are and how many units are produced.

SOLUTIONS, CHAPTER 29

1. a. Three transactions are created which are clones of the transaction entering the block. All three are routed to the block with the label DDDD. The parent transaction moves to the next sequential block. All four transactions are identical.
   b. Four transactions are created and routed to the block with the label AWAY. The parent transaction will move to the next sequential block and will have the value in halfword parameter 3 incremented by 1. The first offspring will have its third halfword parameter incremented by 2, etc.
   c. One transaction will be created which will have 1 halfword parameter and 10 fullword parameters. The value placed in the fifth byte-word parameter for the parent will be the old value plus 1. The value placed in the fifth-word parameter of the clone will be the old value plus 2. The cloned transaction will be routed to the block with the label AWAY and the parent moves to the next sequential block.
   d. Six transactions are created which are routed to the block with the label MARY. The parent is routed to the next sequential block. All six new transactions will have 3 parameters of type halfword, floating-point, byte- and fullword. The parameters and number of the parent are unchained.
   e. Five transactions are created. The first offspring is routed to the block with the label BLOCKA, the second to the block with the label BLOCKB, etc. This is done by using modular arithmetic. The value of N(BLOCK)@5+1 will always be 1, 2, 3, 4, or 5. The function WHERE directs the transactions to the various blocks in sequence.
2. Two transactions are created at the SPLIT block. They, together with the parent transaction are all sent to the next sequential block, the ADVANCE block. All transactions will have 12 halfword parameters (from the GENERATE block) with a value of 3 in the first parameter and zeros in all other parameters except for parameter 2. The parent will have a value of 6, the first offspring a value of 7 and the second offspring a value of 8.

3. The program to do the simulation is:

```
          SIMULATE
          INTEGER    &I
          STORAGE    S(FINAL),2
          DO         &I=1,25
          PUTSTRING  ('  ')
          ENDDO
COMEIN    GENERATE   10,4.5
          QUEUE      WAIT1
          SEIZE      WORKER
          DEPART     WAIT1
          ADVANCE    8,3.6
          RELEASE    WORKER
          SEIZE      CRANE    USE THE CRANE
          ADVANCE    2,.5
          SPLIT      1,DDDD    MAKE CLONE
          ADVANCE    1.5,.25   CRANE RETURNS
          RELEASE    CRANE
          TERMINATE
DDDD      QUEUE      WAIT2    NEXT STEP
          ENTER      FINAL
          DEPART     WAIT2
          ADVANCE    18,7
          LEAVE      FINAL
          TERMINATE
          GENERATE   480
          TERMINATE  1
          START      1
          DO         &I=1,25
          PUTSTRING  ('  ')
          ENDDO
          PUTPIC     LINES=4,N(COMEIN),FR(WORKER)/10.,_
                     SR(FINAL)/10.,FR(CRANE)/10.
   PARTS TO COME INTO THE SHOP    ***
   UTIL. OF WORKER                ***.**%
   UTIL. OF FINAL WORKERS         ***.**%
   UTIL. OF CRANE                 ***.**%
          PUTSTRING  ('  ')
          PUTSTRING  ('  ')
          PUTSTRING  ('  SIMULATION OVER....')
          END
```

The output from running the program is:

```
PARTS TO COME INTO THE SHOP    48
UTIL. OF WORKER                80.29%
UTIL. OF FINAL WORKERS         83.52%
UTIL. OF CRANE                 34.65%
```

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 30

Assembly Sets and the ASSEMBLE, GATHER, and MATCH Blocks

All transactions belong to different groups known as *assembly sets*. When a transaction is created via the GENERATE block, it is assigned its own assembly set. Once a transaction is assigned its own assembly set, it remains there until it leaves the system. These sets are not numbered or named, so a person cannot refer to them. Only after a transaction leaves the system and reenters later can it be assigned a different assembly set. When only a single transaction is in an assembly set, there is not much of interest with the set. It is when there are multiple transactions belonging to a particular assembly set that it is of interest to the programmer.

When a transaction enters a SPLIT block, the clone transactions belong to the same assembly set as the original. Even if these clones themselves enter a SPLIT block, the resulting new clones belong to the same assembly set. Several blocks are used with the concept of assembly sets. The first is the ASSEMBLE block.

THE ASSEMBLE BLOCK

The ASSEMBLE block acts in a manner opposite to the SPLIT block. The SPLIT block clones new transactions into the system and the ASSEMBLE block removes them. The form of it is

```
◇          ASSEMBLE   A
```

Operand *A* gives the number of transactions to be removed from the system. When a transaction enters the ASSEMBLE block, it is delayed there until other members of its assembly set also arrive in the block, where each is removed. Only when the counter (as given by operand *A*) is reached is the *first transaction to enter the ASSEMBLE block* allowed to move to the next sequential block. It is immaterial to GPSS/H if the first transaction to arrive at the ASSEMBLE block is the original uncloned (parent) transaction or not. The first transaction in the block is the one that is allowed to move on.

For example, when a transaction enters the block

```
◇         ASSEMBLE   2
```

it will be delayed until another transaction from its assembly set also enters the block and one is subsequently destroyed. A program might have a set of blocks such as

```
◇         SPLIT      2,DOWNA
◇         -----      -----
◇         -----      -----
◇DOWNA    -----      -----
◇         -----      -----
◇         ASSEMBLE   3
```

The SPLIT block creates two clones when the original transaction enters it. Later, two of the transactions in the same assembly set are removed from the system. *The transaction that goes to the sequential block after the ASSEMBLE block is not necessarily the original transaction.*

It is possible to have the same ASSEMBLE block working on more than one assembly set, and for a given assembly set, it is possible to have assembling operations being done at more than one ASSEMBLE block. Thus,

```
◇         SPLIT      6,DOWN1
◇         -----      -----
◇DOWN1    -----      -----
◇         -----      -----
◇         TRANSFER   ,FN(AWAY)
◇BLOCKA   ASSEMBLE   2
◇         -----      -----
◇         -----      -----
◇BLOCKB   ASSEMBLE   3
◇         -----      -----
◇         -----      -----
```

The block TRANSFER ,FN(AWAY) might send some of the clones to BLOCKA whereas others might go to BLOCKB.

Example 30.1

A drill press is used to drill various numbers of holes in boxes. There are assumed to be an infinite supply of these boxes. A single worker takes 20 ± 8 seconds to position each undrilled box in front of the drill. Next, the worker drills a hole, and the time for this drilling is normally distributed with a mean of 45 seconds and a standard deviation of 7 seconds. After each hole is drilled, it is necessary to adjust the drill and check for tolerances. Then, if all the needed holes are drilled, the box is sent along to the next stage, and the worker takes a new box to be drilled.

Simulate for 10 shifts of work; each shift is 480 minutes long. Determine how many boxes are drilled by the worker. The program must have the number of holes to be drilled as a variable.

Solution

```
◇        SIMULATE
◇        INTEGER    &HOLES,&I,&BOXES
◇        DO         &I=1,25
◇        PUTSTRING  ('')
◇        ENDDO
◇        PUTSTRING  (' HOW MANY HOLES TO DRILL?')
◇        PUTSTRING  ('')
◇        GETLIST    &HOLES
◇        GENERATE   ,,,1
◇BACKUP  SEIZE      DRILLER
◇        SPLIT      1,BACKUP
◇        ADVANCE    20,8              POSITION BOX
◇        ADVANCE    RVNORM(1,45,7)    DRILL
◇        ADVANCE    6,3               ADJUST
◇        ADVANCE    RVEXPO(1,8)       CHECK
◇        RELEASE    DRILLER
◇        ASSEMBLE   &HOLES
◇        BLET       &BOXES=&BOXES+1
◇        TERMINATE
◇        GENERATE   60*60*8*10        SIMULATE FOR 10 SHIFTS
◇        TERMINATE  1
◇        START      1
◇        DO         &I=1,25
◇        PUTSTRING  ('')
◇        ENDDO
◇        PUTPIC     LINES=2,&HOLES,&BOXES/10.
 NUMBER OF HOLES TO DRILL/BOX ***
 AVERAGE BOXES DONE/SHIFT     ***.**
◇        PUTSTRING  ('')
◇        PUTSTRING  (' SIMULATION OVER....')
◇        END
```

Notice that the worker is represented by a single transaction. He or she seizes the drill and immediately makes a clone of the drill. This clone is sent to the SEIZE block but cannot enter this block because it is already seized. Next, the various processes for drilling a single hole are carried out. When they are all done, the drill is released, and the transaction enters the ASSEMBLE &HOLES block. If the number of transactions in this block is one less than the number of holes needed to be drilled, the single transaction is allowed to pass through to increment the number of boxes drilled and then is terminated. At this point, the transaction attempting to enter the SEIZE block can now do so. The output from the program for various numbers of holes to be drilled is

Number of Holes	Boxes Per Day
10	36.40
12	30.30
14	26.00
16	22.70

Example 30.2

It is common to represent a project to be completed as a series of subprojects, each of which must be completed before the whole project is finished. For example, in the construction of a house, the foundation needs to be finished before the walls can be started. However, once the foundation is finished, it is possible to begin work on several projects, not just the walls. The wiring, the plumbing, etc., may be included in these projects. One method of graphically showing the order of all of the subprojects that go into an overall project, such as the construction of a house, is known as a PERT (program evaluation and review technique) diagram. Once a PERT diagram is developed for a project, the expected time to complete the project can be determined. These PERT diagrams are very easy to simulate in GPSS/H by using a combination of the SPLIT and ASSEMBLE blocks.

As an example, the PERT diagram in Figure 30.1 shows the possible steps that are involved in a system involving trucks that come to a shop for routine maintenance. The trucks arrive every 20 ± 8 minutes. The different steps to service a truck and the times for each step are shown. All times are in minutes. Estimate the average time to service a truck. How many trucks are serviced in a typical 8-hour shift? How long is a truck in the service shop? How busy is the shop? Assume that there are always enough workers in the service shop so that a subproject can always be performed.

The program will handle the places where several jobs are to begin by the use of the SPLIT block. This approach allows simulation of more than one job being worked on at once.

Solution

```
◇         □SIMULATE
◇         □INTEGER    □&I
◇         □DO         □&I=1,25
◇         □PUTSTRING  □('□□')
◇         □ENDDO
◇TIMEIN   □TABLE      □M1,15,1,50
* TIME USED IN OUTPUT TO DETERMINE AVG TIME IN SYSTEM
◇         □GENERATE   □20,8,,,,1PL        □TRUCKS ARRIVE
◇         □QUEUE      □WAIT
◇         □GATE□LR    □WAITUP             □CAN IT GO IN?
◇         □LOGIC□S    □WAITUP             □SET GATE FOR NEXT TRUCK
◇         □BUFFER
◇         □DEPART     □WAIT
◇         □ADVANCE    □2,1                □HOIST TRUCK
◇         □SPLIT      □1,FIRST
◇         □ADVANCE    □1,.5               □READY FOR OIL
◇         □SPLIT      □1,SECOND
◇         □ADVANCE    □2,.45              □INSPECT
◇         □TRANSFER   □,NEXT
```

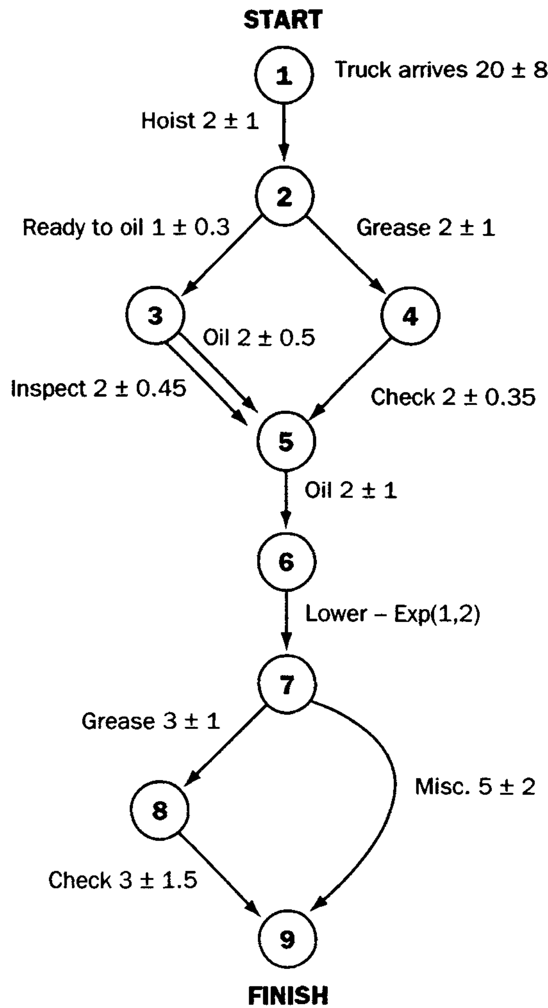


Figure 30.1 PERT diagram for truck in repair shop.

```
SECOND  ADVANCE   2,.5           OIL
        TRANSFER  ,NEXT
FIRST   ADVANCE   2,1            GREASE
        ADVANCE   2,.35          CHECK
NEXT    ASSEMBLE  3              WAIT FOR 3 JOBS TO FINISH
        ADVANCE   2,1            MORE OIL
        ADVANCE   RVEXPO(1,2)    LOWER TRUCK
        SPLIT     1,THIRD
        ADVANCE   3,1            GREASE
        ADVANCE   3,1.5          CHECK ENGINE
        TRANSFER  ,FOURTH
THIRD   ADVANCE   5,2
FOURTH  ASSEMBLE  2
        LOGIC R   WAITUP
        TABULATE  TIMEIN
```

```
◇ALLDONE □TERMINATE
◇        □GENERATE  □,,,1,1          □DUMMY TRANSACTION
◇BACKUP  □GATE□LS   □WAITUP          □HOLD UNTIL SERVICE BUSY
◇        □SEIZE     □DUMMY
◇        □GATE□LR   □WAITUP
◇        □RELEASE   □DUMMY
◇        □TRANSFER  □,BACKUP
◇        □GENERATE  □480*100
◇        □TERMINATE □1
◇        □START     □1
◇        □DO        □&I=1,25
◇        □PUTSTRING □('□')
◇        □ENDDO
◇        □PUTPIC    □LINES=4,N(ALLDONE)/100.,QA(WAIT),TB(TIMEIN),_
                     FR(DUMMY)/10.
  TRUCKS SERVICED EACH DAY    ***.**
  AVERAGE WAITING NUMBER       **.**
  AVERAGE TIME IN THE SYSTEM  ***.**
  UTIL. OF SERVICE            ***.**%
◇        □END
```

Notice how the utilization of the system must be written. A dummy transaction is created that seizes a dummy facility whenever a truck is being serviced. It is not possible to have regular SEIZE and RELEASE blocks in the main program because only the original (nonclone) transaction SEIZEs a facility. Since the ASSEMBLE block destroys transactions without regard to the order in which they have arrived, it may try to destroy a transaction that has a facility seized. This attempt would result in a run-time error. The following output from the program shows that the system appears to be performing satisfactorily:

```
TRUCKS SERVICED EACH DAY    24.07
AVERAGE WAITING NUMBER       0.09
AVERAGE TIME IN THE SYSTEM  18.06
UTIL. OF SERVICE            81.94%
```

Example 30.3 (from Schriber, 1974, p. 466; see Preface for availability)

In Example 30.2, it was assumed that there were enough workers to perform the service on the truck. In practice, each job will require a set amount of workers. In the following example, the number of workers for each is specified. It is assumed that the job cannot begin until these workers are available. Thus, if a job requires 5 workers and only 4 are available, the job cannot be started.

The network shown in Figure 30.2 represents a series of subprojects that must be carried out to complete an overall project. A pair of circles (nodes) connected by a directed line segment is used to depict each particular subproject. For example, node 1 is connected to node 2, depicting what is called subproject 1 to 2. Each directed line segment is labeled to show how many people and how many time units are required to perform the corresponding subproject. Subproject 1 to 2 requires 4 people and takes 14 ± 6 time units to complete.

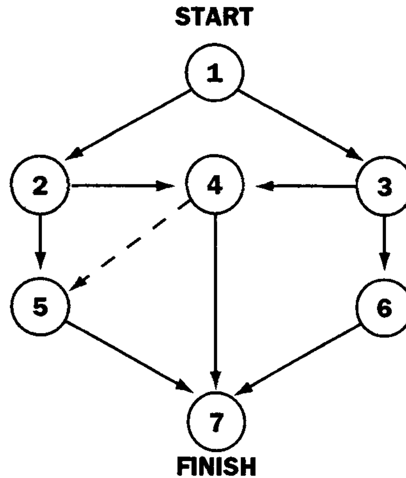


FIGURE 30.2 PERT diagram.

The times and people needed for each project are as follows:

Project	People Needed	Time
1 to 2	4	14 ± 6
1 to 3	3	20 ± 9
2 to 4	3	10 ± 3
2 to 5	5	18 ± 4
3 to 4	2	22 ± 5
3 to 6	1	25 ± 7
4 to 7	4	15 ± 5
4 to 5	0	no time
5 to 7	2	8 ± 3
6 to 7	4	10 ± 3

Figure 30.2 also displays the precedence constraints, indicating which subprojects must be completed before other subprojects can be started. For example, subprojects 2 to 4 and 3 to 4 must be completed before subproject 4 to 7 can be started. Similarly, subproject 5 to 7 cannot be initiated until subproject 2 to 5 is finished, and until the subprojects leading into node 4 have been completed.

In projects of this kind, it is of interest to know how much time will be required to complete the overall project. Use GPSS/H to simulate the undertaking of the overall project shown in Figure 30.2. Have the model determine the average number of workers on the job at any one time. The program should be interactive so that the number of jobs simulated is a variable as is the number of workers.

Solution

```
◇        □SIMULATE
◇        □INTEGER    □&I,&JOBS,&NJOBS,&WORK
◇        □CHAR*1     □&ANS
◇AGAIN1  □DO         □&I=1,25
◇        □PUTSTRING  □('□')
◇        □ENDDO
◇        □PUTSTRING  □(' HOW MANY JOBS TO SIMULATE FOR?')
◇        □PUTSTRING  □('□')
◇        □GETLIST    □&NJOBS
◇AGAIN2  □PUTSTRING  □('□')
◇        □PUTSTRING  □('HOW MANY WORKERS ARE THERE?')
◇        □PUTSTRING  □('□')
◇        □GETLIST    □&WORK
◇        □PUTSTRING  □('□')
◇        □STORAGE    □S(WORKER),&WORK
◇        □PUTSTRING  □('  SIMULATION IN PROGRESS....')
◇RTIME   □TABLE      □MP1PL,25,25,30
◇INUSE   □TABLE      □S(WORKER),7,1,20
◇        □GENERATE   □,,,1,,1PL
◇BACK    □MARK       □1PL
◇        □SPLIT      □1,SUB13          □WORK ON 1 - 3 AND 1 - 2
◇        □ENTER      □WORKER,4         □WORKERS FOR 1 - 2
◇        □ADVANCE    □14,6
◇        □LEAVE      □WORKER,4
◇        □SPLIT      □1,SUB24
◇        □ENTER      □WORKER,5         □WORKERS FOR 2 - 4
◇        □ADVANCE    □18,4
◇        □LEAVE      □WORKER,5
◇NODE5   □ASSEMBLE   □2
◇        □ENTER      □WORKER,2         □WORK ON 5 - 7
◇        □ADVANCE    □8,3
◇        □LEAVE      □WORKER,2
◇        □TRANSFER   □,NODE7
◇SUB24   □ENTER      □WORKER,3         □WORK ON 2 - 4
◇        □ADVANCE    □10,3
◇        □LEAVE      □WORKER,3
◇NODE4   □ASSEMBLE   □2
◇        □SPLIT      □1,NODE5
◇        □ENTER      □WORKER,4         □WORKERS FOR 4 - 7
◇        □ADVANCE    □15,5
◇        □LEAVE      □WORKER,4
◇NODE7   □ASSEMBLE   □3                □WAIT FOR JOBS TO FINISH
◇        □TABULATE   □RTIME            □TABULATE TIMES FOR EACH JOB
◇        □BLET       □&JOBS=&JOBS+1    □COUNT JOBS
◇        □TRANSFER   □,BACK
◇SUB13   □ENTER      □WORKER,3
◇        □ADVANCE    □20,9
◇        □LEAVE      □WORKER,3
◇        □SPLIT      □1,SUB34
◇        □ENTER      □WORKER,1         □WORK 3 - 6
◇        □ADVANCE    □25,7
```

```
◇        ▯LEAVE     ▯WORKER,1
◇        ▯ENTER     ▯WORKER,4
◇        ▯ADVANCE   ▯10,3
◇        ▯LEAVE     ▯WORKER,4
◇        ▯TRANSFER  ▯,NODE7
◇SUB34   ▯ENTER     ▯WORKER,2            ▯WORK 3 - 4
◇        ▯ADVANCE   ▯22,5
◇        ▯LEAVE     ▯WORKER,2
◇        ▯TRANSFER  ▯,NODE4
◇        ▯GENERATE  ▯,,,1,10,1PL         ▯PROVIDE 1 TRANSACTION (I.E., WORKER)
◇BACKUP  ▯MARK      ▯1PL                 ▯MARK TIME IN PARAMETER 1
◇        ▯TEST▯NE   ▯MP1PL,0             ▯WAIT UNTIL CLOCK CHANGES
◇        ▯TABULATE  ▯INUSE               ▯MAKE ENTRY IN TABLE (WORKERS USED)
◇        ▯TRANSFER  ▯,BACKUP             ▯GO BACK AND WAIT FOR CLOCK TO CHANGE
◇        ▯GENERATE  ▯,,,1
◇        ▯TEST▯GE   ▯&JOBS,&NJOBS
◇        ▯TERMINATE ▯1
◇        ▯START     ▯1
◇        ▯DO        ▯&I=1,25
◇        ▯PUTSTRING ▯('▯▯')
◇        ▯ENDDO
◇        ▯PUTPIC    ▯LINES=6,&NJOBS,&WORK,TB(RTIME)_
                     SR(WORKER)/10,SM(WORKER),TB(INUSE)
 SIMULATION FOR        ***** JOBS
 NUMBER OF WORKERS       ***
 AVERAGE TIME/JOB        ***.**
 UTIL. OF WORKERS        ***.**
 MAX. WORKERS IN USE      **
 AVG. NO. WORKERS IN USE **.**
◇        ▯PUTSTRING ▯('▯▯')
◇        ▯PUTSTRING ▯('▯▯')
◇        ▯PUTSTRING ▯(' DO AGAIN? (Y/N)')
◇        ▯PUTSTRING ▯('▯▯')
◇        ▯GETLIST   ▯&ANS
◇        ▯IF        ▯(&ANS'E''Y')OR(&ANS'E''y')
◇        ▯CLEAR
◇        ▯LET       ▯&JOBS=0
◇        ▯PUTSTRING ▯('▯▯')
◇        ▯PUTSTRING ▯(' CHANGE ONLY THE NO. OF WORKERS? (Y/N)')
◇        ▯PUTSTRING ▯('▯▯')
◇        ▯GETLIST   ▯&ANS
◇        ▯IF        ▯(&ANS'E''Y')OR(&ANS'E''y')
◇        ▯GOTO      ▯AGAIN2
◇        ▯ENDIF
◇        ▯GOTO      ▯AGAIN1
◇        ▯ENDIF
◇        ▯END
```

Notice how the segment to handle the number of workers is done. The 5 lines of code involved are labeled with comments. A dummy transaction cycles through the lower 4 blocks. The time of the simulation is placed in its first fullword parameter. Whenever the time changes in the program–which happens whenever a job is complete–an entry is made into the table INUSE, which gives the number of workers employed.

The simulation was run until 100 jobs were completed. A summary of the results of the simulation are as follows:

Number of Workers	Avg. Time to Complete Project	Util. of Workers	Avg. No. Working	Maximum No. Working
6	98.27	71.48%	4.58	6
7	81.80	73.70%	5.15	7
8	62.73	83.98%	6.30	8
9	66.13	70.66%	6.33	9
10	65.26	64.60%	6.60	10
11	58.47	65.33%	7.38	11
12	58.44	60.03%	7.39	11
13	59.03	55.11%	7.38	13
..	....	....	....	.....
500	58.92	1.43%	7.37	14

As can be seen, the optimum number of workers to have available appears to be 11. This will depend on being able to keep all the surplus workers busy, so there would have to be further economic study to determine the actual optimum number of workers to have. When an infinite labor supply is available, as indicated by the output data for 500 workers, the maximum number ever in use is 14.

In Schriber's example, the simulation was run for 250 jobs. When 11 workers were available, the utilization was 64.8%, and the average number of workers in use was 7.35. The average time for a job was 59.1 time units. These numbers are in good agreement with those found here.

THE GATHER BLOCK

The GATHER block acts much the same as the ASSEMBLE block in that it holds transactions from the same assembly set until a specified number is reached. This number is given by operand *A*. Whereas the ASSEMBLE block destroys all but 1 transaction when this number is reached, the GATHER block lets all the transactions through to the next sequential block. The general form of the GATHER block is

```
◇        GATHER    A
```

Some examples of this block are

```
◇(a)     GATHER    3
◇(b)     GATHER    &AMOUNT
◇(c)     GATHER    FN(AMT1)
```

In (a), the GATHER block holds transactions from the same assembly set until it has accumulated 3 and then lets all 3 move to the next sequential block. In (b) and (c), operand *A* is specified by the ampervariable &AMOUNT and by the function FN(AMT1), respectively.

There are not too many applications of the GATHER block compared to the ASSEMBLE block since once transactions have been cloned, it is rare that they are allowed to remain in the program for very long. Probably the most common example of the GATHER block comes from manufacturing, as in the next example.

Example 30.3

Parts come along an assembly line and are stacked in front of a single worker who inspects each part. Of these parts, 10% are rejected. It is assumed that there are always enough parts arriving for the worker to do the inspection. The worker inspects a part in 8 ± 4 seconds and sets it aside until 50 parts have passed inspection. When the count becomes 50, the worker places the inspected parts inside of a carton. It takes 1.5 ± 0.5 seconds for the worker to place each part in the box. Determine how many boxes the worker can fill each day. The worker receives no breaks for lunch or coffee.

Solution

```
◇        SIMULATE
◇        INTEGER    &I,&BOXES
◇        DO         &I=1,25
◇        PUTSTRING  ('  ')
◇        ENDDO
◇        GENERATE   ,,,1              DUMMY TRANSACTION
◇BACKUP  SEIZE      WORKER            IS BUSY
◇        ADVANCE    8,4               CHECKS A PART
◇        SPLIT      1,BACKUP          CREATE NEW TRANSACTION
◇        RELEASE    WORKER            FREE, BUT...
◇        TRANSFER   .1,,BADPART       10% ARE BAD
◇        PRIORITY   10                HIGH PRIORITY
◇        GATHER     50                WAIT FOR 50 DONE
◇        SEIZE      WORKER            NOW, SEIZE WORKER AGAIN
◇        ADVANCE    1.5,.5            PLACE PART IN BOX
◇        RELEASE    WORKER            FREE WORKER TO PLACE NEXT PART IN BOX
◇        ASSEMBLE   50                NEED 50 IN THE BOX
◇        BLET       &BOXES=&BOXES+1   COUNT BOXES
◇        TERMINATE
◇BADPART TERMINATE
◇        GENERATE   3600*8            SIMULATE FOR 8 HOURS
◇        TERMINATE  1
◇        START      1
◇        DO         &I=1,25
◇        PUTSTRING  ('  ')
◇        ENDDO
◇        PUTPIC     LINES=1,&BOXES
 BOXES FILLED/DAY     ***
◇        END
```

The worker will produce 55 boxes a day.

THE MATCH BLOCK

The MATCH block is very useful in modeling manufacturing systems. A MATCH block always has a "twin" block that is in another part of the program. The effect of a MATCH block is to cause a pair of transactions to "wait for each other" before both can move on to the next block in the program. The form of the MATCH block is as follows:

```
◇LABEL1 ▯MATCH    ▯LABEL2
```

In another part of the program, there must be a corresponding block that has the following form:

```
◇LABEL2 ▯MATCH    ▯LABEL1
```

Thus, operand *A* of the first MATCH block becomes the label of the second MATCH block and vice versa. A transaction arriving first at either of these blocks will be delayed until a member of its assembly set arrives at the other block. The MATCH blocks can be thought of as "pointing" at each other. Once a member of the same assembly set arrives at the matching block, both transactions are allowed to move to the next sequential blocks.

Example 30.4

Trucks arrive at a repair shop every 18 ± 4 minutes. There are two crews to service a truck. Both can begin work simultaneously. When a truck comes for service, the "blue crew" can begin lubrication of the underside of the truck, which takes 6 ± 1.5 minutes, and the "green crew" does an engine inspection, which takes 6.5 ± 1 minutes. Only when both jobs are done can the truck be repositioned for the next jobs. The blue crew does minor adjustments in 7 ± 3 minutes, and the green crew does minor repairs to the engine in 8 ± 4.5 minutes. When these jobs are done, the blue crew does a quick wash of the truck, which takes 3 ± 1 minutes. Simulate for 100 shifts of 480 minutes each to determine the number of trucks that can be serviced in a shift, the average number of trucks waiting in the queue at the repair shop, and the utilization of both crews.

Solution

```
◇        ▯SIMULATE
◇        ▯INTEGER   ▯&I
◇        ▯DO        ▯&I=1,25
◇        ▯PUTSTRING ▯('▯▯')
◇        ▯ENDDO
◇        ▯GENERATE  ▯18,4             ▯TRUCKS ARRIVE
◇        ▯QUEUE     ▯WAIT
◇        ▯SPLIT     ▯1,AWAY
◇        ▯SEIZE     ▯BLUE             ▯BLUE CREW
◇        ▯DEPART    ▯WAIT
◇        ▯ADVANCE   ▯6,1.5            ▯LUBRICATE
◇        ▯RELEASE   ▯BLUE             ▯BLUE CREW
◇WAITUP  ▯MATCH     ▯OTHERS           ▯WAIT FOR OTHER CREW
◇        ▯SEIZE     ▯BLUE             ▯BLUE CREW
◇        ▯ADVANCE   ▯7,3              ▯MINOR ADJUSTMENTS
◇        ▯RELEASE   ▯FIRST
```

```
◇BACKUP  ▯ASSEMBLE  ▯2
◇        ▯SEIZE     ▯BLUE              ▯BLUE CREW
◇        ▯ADVANCE   ▯3,1               ▯QUICK WASH
◇        ▯RELEASE   ▯BLUE              ▯BLUE CREW
◇TRUCK   ▯TERMINATE
◇AWAY    ▯SEIZE     ▯GREEN             ▯GREEN CREW
◇        ▯ADVANCE   ▯6.5,1             ▯ENGINE INSPECTION
◇        ▯RELEASE   ▯GREEN             ▯GREEN CREW
◇OTHERS  ▯MATCH     ▯WAITUP            ▯WAIT FOR OTHER CREW
◇        ▯SEIZE     ▯GREEN             ▯GREEN CREW
◇        ▯ADVANCE   ▯7,3               ▯FINE TUNING
◇        ▯RELEASE   ▯GREEN             ▯GREEN CREW
◇        ▯TRANSFER  ▯,BACKUP
◇        ▯GENERATE  ▯480*100
◇        ▯TERMINATE ▯1
◇        ▯START     ▯1
◇        ▯DO        ▯&I=1,25
◇        ▯PUTSTRING ▯('▯')
◇        ▯ENDDO
◇        ▯PUTPIC    ▯LINES=4,N(TRUCK)/100.,FR(FIRST)/10.,_
                     FR(SECOND)/10.,QA(WAIT)
 TRUCKS SERVICED PER SHIFT     ***.**
 UTIL. OF BLUE CREW            ***.**%
 UTIL. OF GREEN CREW           ***.**%
 AVERAGE NO. OF TRUCKS WAITING ***.**
◇        ▯END
```

The results of the simulation are

```
TRUCKS SERVICED PER SHIFT      26.59
UTIL. OF BLUE CREW             88.67%
UTIL. OF GREEN CREW            74.95%
AVERAGE NO. OF TRUCKS WAITING   0.07
```

As can be seen from the results, the service area seems to be working all right.

EXERCISES, CHAPTER 30

1. A race car comes into the pits for a routine stop. It needs the following operations done, each of which start at the same time and each of which requires a different crew of mechanics. All times are in seconds.
   a. Refueling: 18 ± 6
   b. New tires: 22 ± 7
   c. Refresh the driver: normal distribution, mean of 15, standard deviation of 3
   d. Engine check: exponential distribution, mean of 19

   Simulate for 5000 pit stops to determine the expected time in the pits.
2. The average time in the pit for the race car in exercise 1 is found to be too long. Suppose that the engine check time can be changed to being from the normal distribution with mean of 19 and standard deviation of 3.1. How does this change the expected time in the pits?

SOLUTIONS, CHAPTER 30

1. The program to do the simulation is:

```
          SIMULATE
          INTEGER     &I,&NO,&NOSIM
          DO          &I=1,25
          PUTSTRING   ('  ')
          ENDDO
          PUTSTRING   ('  HOW MANY PIT STOPS TO SIMULATE FOR?')
          PUTSTRING   ('  ')
          GETLIST     &NOSIM
  WHERE   FUNCTION    N(BLOCK)@3+1,L3
1,BLOCKA/2,BLOCKB/3,BLOCKC
 INPITS   TABLE       MPL1PL,15,1,30
          GENERATE    ,,,1,,1PL
 BACKUP   MARK        1PL
 BLOCK    SPLIT       3,FN(WHERE)
          ADVANCE     18,6    REFUEL
          TRANSFER    ,READY
 BLOCKA   ADVANCE     22,7    NEW TIRES
          TRANSFER    ,READY
 BLOCKB   ADVANCE     RVNORM(1,15,3)  DRIVER REFRESHED
          TRANSFER    ,READY
 BLOCKC   ADVANCE     RVEXPO(1,19)   ENGINE CHECK
 READY    ASSEMBLE    4
          TABULATE    INPITS
          BLET        &NO=&NO+1
          BUFFER
          TRANSFER    ,BACKUP
          GENERATE    ,,,1,1
          TEST GE     &NO,&NOSIM
          TERMINATE   1
          START       1
          DO          &I=1,25
          PUTSTRING   ('  ')
          ENDDO
          PUTPIC      LINES=2,&NOSIM,TB(INPITS)
  NUMBER OF PIT STOPS TO SIMULATE FOR ****
  AVERAGE TIME IN THE PITS              ***.**
          END
          END
```

The results of the simulation are somewhat surprising as they are:

```
NUMBER OF PIT STOPS TO SIMULATE FOR 5000
AVERAGE TIME IN THE PITS              28.67
```

2. The change to the program requires only changing the block:

```
BLOCKC  ADVANCE  RVEXPO(1,19)
```

to

```
BLOCKC  ADVANCE  RVNORM(1,19,3.1)
```

Making this change and running the simulation yields an average expected time of 23.32. This is a considerable time difference from the results of Exercise 30.1.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 31

Macros

MACROS

Anyone seeing a complex GPSS/H program for the first time will most certainly be confused by the cryptic appearing nature of the code. Part of the reason for this is that it is good programming to use as many macros as possible. A macro is a shorthand way of writing lines of code that are repeated in the program with the only differences being in text or numbers. Most of these differences are found in the labels, operands, and the output from BPUTPICS but may be found in other blocks as well. For example, many large programs have the following lines of code repeated many times with the only difference being in the operands:

```
        QUEUE     WAIT1
        SEIZE     MACH1
        DEPART    WAIT1
        ADVANCE   20,5
        RELEASE   MACH1
```

Then in another part of the program, there might be the following lines of code:

```
        QUEUE     WAIT2
        SEIZE     MACH2
        DEPART    WAIT2
        ADVANCE   15,5
        RELEASE   MACH2
```

These 5 lines of code are very similar with the only differences being in the operands. A macro will replace these lines of code by a single line. The macro must first be defined before it is used. Most macros are defined near the top of the program before the first GENERATE block, but this positioning is only for the convenience of the programmer. The statements that specify the macro are known as *macro definition* statements. Text to be replaced when the macro is expanded during compiling are indicated by the number sign, "#," followed by a letter from A to J. Thus, the 5 blocks given previously could be written in the macro definition as

```
◇        QUEUE     #A
◇        SEIZE     #B
◇        DEPART    #A
◇        ADVANCE   #C,#D
◇        RELEASE   #B
```

In the program, one would have a single line of code. This might be as follows:

```
◇MYFIRST MACRO     WAIT1,MACH1,20,5
```

where WAIT1 replaces #A, MACH1 replaces #B, 20 replaces #C, and 5 replaces #D. Since the macro text replacement is indicated by a letter from A to J, this means that a macro can have only 10 such replacements. However, it is possible to have a macro invoke another macro, which in turn can invoke a macro, etc.; so, in practice, there can be many such replacements.

The formal definition of a macro is as follows:

```
◇LABEL   STARTMACRO
◇        ----- .macro definition statements
◇        ----- .macro definition statements
◇        ENDMACRO
```

The "words" STARTMACRO and ENDMACRO must be as shown. When a macro is invoked in the main program, the form is

```
◇LABEL   MACRO     A,B,C . . .
```

where operands *A, B, C,* . . . are a list of values or even text.

Thus, one might have a macro definition as

```
◇MYFIRST STARTMACRO
◇        QUEUE     #A
◇        SEIZE     #B
◇        DEPART    #A
◇        ADVANCE   #C,#D
◇        RELEASE   #A
◇        ENDMACRO
```

In the main program, one might have the following to correspond to the macro definition:

```
◇MYFIRST MACRO     WAIT1,MACH1,20,5
```

In the .LIS file, the macro is shown expanded with plus signs, "+," before the various blocks.

Another example of a macro is

```
◇MYNEXT  STARTMACRO
◇        SEIZE     MACH#A
◇        ADVANCE   RVEXPO(4,#B)
◇        RELEASE   MACH#A
◇        ENDMACRO
```

In the body of the program, if one had the statement

```
◇MYNEXT  ▯MACRO     ▯1,20
```

The macro would be expanded to be

```
◇        ▯SEIZE     ▯MACH1
◇        ▯ADVANCE   ▯RVEXPO(4,20)
◇        ▯RELEASE   ▯MACH1
```

Since the text output in a BPUTPIC is not case sensitive, it is possible to have a macro with lowercase letters as follows:

```
◇        ▯CHAR*11   ▯&J
◇        ▯LET       ▯&J='hello there'
◇        ▯-----
◇        ▯-----
◇MYMAC   ▯STARTMACRO
◇        ▯PUTPIC    ▯AC1,#A
 THE CLOCK TIME IS ***.**  *
◇        ▯ENDMACRO
```

In the body of the program, it would be permissible to have

```
◇MYMAC   ▯MACRO     ▯&J
```

The output from the program might be, for example,

```
 THE CLOCK TIME IS 123.45 hello there
```

Alternatively, one could have the following:

```
◇MYMAC2  ▯STARTMACRO
◇        ▯BPUTPIC   ▯#A,#B
 THE TIME IS ***.**  *
◇        ▯ENDMACRO
```

And then in the program, one could have the following:

```
MYMAC2  ▯MACRO    ▯AC1,' program ok to here'
```

The output from this code might be

```
 THE TIME IS 123.45  program ok to here
```

The advantage of using macros not only is in their economy of lines of code, but they can easily be lined up under each other so that typing errors are easy to detect. The use of macros is illustrated in the next example.

Example 31.1

There are 6 trucks in a mine. A single shovel loads a truck in 2 ± 1 minutes. Trucks haul ore to the crusher and return to the shovel. These other times are

Operation	Time (minutes)
travel to crusher	5 ± 2
dump	1 ± 0.5
return to shovel	3.5 ± 1.2

Only one truck can dump at a time. Each truck has been found to have the following time distributions (in minutes) for when it is working and for how long it is down. For convenience, the trucks are numbered from 1 to 6.

Truck No.	Running Time	Time Down (exponential distribution)
1	100 ± 15	15
2	130 ± 20	13
3	120 ± 15	15
4	100 ± 25	25
5	105 ± 23	23
6	100 ± 20	24

Simulate for 500 shifts of 480 minutes each to determine the average production per shift, the utilization of the shovel, and the utilization of the crusher. Run the program again with none of the trucks ever being out of production.

Solution Using Macros

```
◇        □SIMULATE
◇        □RMULT      □654321
◇        □INTEGER    □&I
◇        □DO         □&I=1,25
◇        □PUTSTRING  □('□□')
◇        □ENDDO
◇MYFIRST □STARTMACRO
◇        □GENERATE   □,,,1
◇#A      □ADVANCE    □#B,#C
◇        □LOGIC□S    □#D
◇        □ADVANCE    □RVEXPO(1,#E)
◇        □LOGIC□R    □#D
◇        □TRANSFER   □,#A
◇        □ENDMACRO
◇MYNEXT  □STARTMACRO
◇#A      □GATE□LR    □STOP#B
◇        □TRANSER    □,BACKUP
◇        □ENDMACRO
◇WHERE   □FUNCTION   □PH1,L6
1,BLOCKA/2,BLOCKB/3,BLOCKC/4,BLOCKD/5,BLOCKE/6,BLOCKF
◇TRUCK   □GENERATE   □6,,0,6              □TRUCKS IN THE MINE
◇        □ASSIGN     □1,N(TRUCK),PH       □NUMBER THEM
◇BACKUP  □QUEUE      □WAIT                □AT SHOVEL
◇        □SEIZE      □SHOVEL
◇        □DEPART     □WAIT
◇        □ADVANCE    □2,1                 □LOAD A TRUCKS
◇        □RELEASE    □SHOVEL
◇        □ADVANCE    □5,2                 □TRAVEL TO CRUSHER
◇        □SEIZE      □CRUSHER
◇        □ADVANCE    □1,.5                □DUMP A LOAD
◇        □RELEASE    □CRUSHER
◇        □ADVANCE    □3.5,1.2             □RETURN TO SHOVEL
◇        □TRANSFER   □,FN(WHERE)          □CHECK EACH TRUCK
```

```
◇MYNEXT  □MACRO     □BLOCKA,1                □TRUCK 1
◇MYNEXT  □MACRO     □BLOCKB,2                □TRUCK 2
◇MYNEXT  □MACRO     □BLOCKC,3                □TRUCK 3
◇MYNEXT  □MACRO     □BLOCKD,4                □TRUCK 4
◇MYNEXT  □MACRO     □BLOCKE,5                □TRUCK 5
◇MYNEXT  □MACRO     □BLOCKF,6                □TRUCK 6
◇MYFIRST □MACRO     □BACK1,100,15,STOP1,15  □RELIABILITY FOR TRUCK 1
◇MYFIRST □MACRO     □BACK2,130,20,STOP2,13  □RELIABILITY FOR TRUCK 2
◇MYFIRST □MACRO     □BACK3,120,15,STOP3,15  □RELIABILITY FOR TRUCK 3
◇MYFIRST □MACRO     □BACK4,100,25,STOP4,25  □RELIABILITY FOR TRUCK 4
◇MYFIRST □MACRO     □BACK5,105,23,STOP5,23  □RELIABILITY FOR TRUCK 5
◇MYFIRST □MACRO     □BACK6,100,20,STOP6,24  □RELIABILITY FOR TRUCK 6
◇        □GENERATE  □480*500
◇        □TERMINATE □1
◇        □START     □1
◇        □DO        □&I=1,25
◇        □PUTSTRING □('□')
◇        □ENDDO
◇        □PUTPIC    □LINES=3,FC(CRUSHER)/500.,FR(SHOVEL)/10.,_
                     FR(CRUSHER)/10.
 LOADS DUMPED PER SHIFT   ***.**
 UTIL. OF SHOVEL         ***.**%
 UTIL. OF CRUSHER        ***.**%
◇        □END
```

Notice how the macros line up underneath each other in the program listing. In the .LIS file, one can see how the program looks when each macro is expanded. The output from the program is as follows:

```
LOADS DUMPED PER SHIFT  199.59
UTIL. OF SHOVEL          83.15%
UTIL. OF CRUSHER         41.54%
```

When the program is run with the trucks never being down, the output is as follows:

```
LOADS DUMPED PER SHIFT   219.59
UTIL. OF SHOVEL           91.48%
UTIL. OF CRUSHER          45.68%
```

In the case of an actual mine, there would be a slight error in the above results since the trucks are only tested at one place in the mine. This error can be corrected by several means, such as testing the trucks at more places or simply adjusting the downtime distributions.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

CHAPTER 32

Other GPSS/H Blocks

As the GPSS simulation language evolved over the years from the original, crude form to its present modern, dynamic form known as GPSS/H, the number and power of the programming blocks changed. At one time, there was a block known as the HELP block whose operand was a compiled program from another language. When a transaction entered a HELP block, the executable program (normally written in Fortran) was executed. The block is still supported by GPSS/H but is considered obsolete and will not be discussed in this chapter. Instead, other blocks that are of use to the mining engineer will be presented with short programs to illustrate their salient features. Every block used in the examples in Part III of this book has been introduced in Part II. However, the blocks presented here are such that they may be useful in developing other programs for mining simulation.

BGETLIST AND BPUTSTRING BLOCKS

These blocks act exactly the same way as the GETLIST and PUTSTRING statements except that now they are blocks. The program will stop execution, and the user will be prompted for input values when a transaction enters the BGETLIST block. The BPUTSTRING puts text onto the screen. It is possible to read from a data file when a BGETLIST block is entered by a transaction. For example, it is possible to have

```
        BPUTSTRING  (' INPUT VALUES FOR &K, &J, AND &L')
        BGETLIST    &K,&J,&L
```

Control would pass to the keyboard, and the screen would show the message

```
INPUT VALUES FOR &K, &J, and &L
```

The cursor would flash until values for &K, &J, and &L have been keyed in. If they are all to be input at the same time, they need to be separated by spaces. One can read a data file by the following code:

```
◇         SIMULATE
◇MYOWN    FILEDEF    'DATA.TXT'
◇         -----
◇         -----
◇         BGETLIST   FILE=MYOWN,&K,&J,&K
```

FAVAIL AND FUNAVAIL BLOCKS

These blocks are used to shut down a facility and allow the transaction the option of leaving the facility when that is desired. The general form of the FUNAVAIL block is quite complex:

```
◇         FUNAVAIL   A,B,C,D,E,F,G,H
```

The *A* operand is the name or number of the facility to be made unavailable. When a facility is made unavailable, if there is a transaction that has seized it, the transaction will give up control of the facility until the facility becomes available again. Consider the following example

```
◇         SIMULATE
◇         GENERATE   ,,,1
◇         SEIZE      FIRST
◇         ADVANCE    10
◇         RELEASE    FIRST
◇         BPUTPIC    AC1
 TIME IS ***.**
◇         TERMINATE  1
◇         GENERATE   ,,,1
◇BACK     ADVANCE    5
◇         FUNAVAIL   FIRST
◇         ADVANCE    20
◇         FAVAIL     FIRST
◇         TRANSFER   ,BACK
◇         START      1
◇         END
```

A transaction is created at time $t = 0$. It uses the facility FIRST and is placed on the FEC for 10 time units, so it normally would come off at time $t = 15$. However, a second transaction is also created at time 0 and placed on the FEC for 5 time units. At this time, it comes off the FEC and makes the facility FIRST unavailable until time 25. The original transaction now retains control of the facility and has 5 minutes left on the FEC. When it releases the facility FIRST, the clock will be at time $t = 30$. The BPUTPIC will indicate this time on the screen.

The *B* operand will be either CO or RE. If it is CO, the transaction that is seizeing the facility when it is made unavailable will continue to control the facility until it is finished. If it is RE, the transaction that is seizeing the facility is removed and routed to the block as specified by the *C* operand. However, the transaction that is so removed no longer controls the facility and does not need to release it. A program to illustrate this is as follows:

```
◇        □SIMULATE
◇        □GENERATE  □,,,1,,10PL
◇        □SEIZE     □FIRST
◇        □ADVANCE   □10
◇        □RELEASE   □FIRST
◇        □TERMINATE □1
◇        □RELEASE   □FIRST
◇        □TERMINATE □1
◇BLOCKA  □BPUTPIC   □AC1
 TIME IS ***.**
◇        □TERMINATE □1
◇        □GENERATE  □,,,1
◇BACK    □ADVANCE   □5
◇        □FUNAVAIL  □FIRST,RE,BLOCKA
◇        □ADVANCE   □20
◇        □FAVAIL    □FIRST
◇        □TRANSFER  □,BACK
◇        □START     □1
◇        □END
```

A transaction is created at time $t = 0$ from the first GENERATE block and seizes the facility FIRST where the transaction is to remain for 10 time units. However, the transaction that is created by the second GENERATE block makes the facility FIRST unavailable at time $t = 5$ and removes the earlier-created transaction from contention for the facility FIRST. The earlier-created transaction is routed to the block with the label BLOCKA. When the program is run, the output shows that the time is 5.00. If the program has the line of code

```
◇        □FUNAVAIL  □FIRST,CO,BLOCKA
```

facility FIRST would *have* to be released or a run-time error would occur.

Operand *D* is the name or number of the parameter of the transaction currently using the facility that has been made unavailable. The time remaining for the transaction to be on the FEC is placed in this parameter. This operand can be used only if the RE option is used for operand *B*. The parameter needs to be floating point or a warning message is sent to the screen. Consider the following program:

```
◇        □SIMULATE
◇        □GENERATE  □,,,1,,10PL
◇        □ADVANCE   □20
◇        □SEIZE     □FIRST
◇        □ADVANCE   □10
◇        □RELEASE   □FIRST
◇        □TERMINATE □1
◇BLOCKA  □BPUTPIC   □PL1
 PARAMETER IS ***.**
◇        □RELEASE   □FIRST
◇        □TERMINATE □1
◇        □GENERATE  □,,,1
```

```
◇BACK    ▯ADVANCE    ▯25
◇        ▯FUNAVAIL   ▯FIRST,CO,BLOCKA,1PL
◇        ▯ADVANCE    ▯20
◇        ▯FAVAIL     ▯FIRST
◇        ▯TRANSFER   ▯,BACK
◇        ▯START      ▯1
◇        ▯END
```

When this code is executed, the value of the first floating-point parameter is 5.00, since this is the time remaining for the transaction that was using the facility when it was made unavailable.

The *E* operand has to do with a transaction that is using the facility via a PREEMPT block, which will be covered later in this chapter. It is coded as either CO or RE and has the same effect as the *B* operand. The *F* operand goes with the *E* operand and gives the name of the block the transaction is to be routed to. The *G* operand has to do with any transaction that might be waiting to seize or preempt the facility that is now unavailable. If omitted, these transactions are removed from competing for the facility (as one would expect). If coded as RE, the transactions will be sent to the block given by the label specified by the operand in position *H*.

LINK AND UNLINK BLOCKS

The LINK block is used when there is a blocking condition and the blocked transactions are to be temporarily removed from the CEC and placed on a different chain, known as the "user chain." Once these transactions are placed on this user chain, the order in which they are placed back in the program is controlled by the programmer, hence the name "user." Normally, the transactions that are placed on the user chain are taken off when the blocking condition no longer exists. Imagine a shop with a single worker. When the worker is busy with a customer, the transactions waiting for service are kept on the CEC and are included in every rescan. Even though this scan will often be only to check what is known as the "scan indicator," this still takes time. Imagine that the transactions are removed from the system instead, just as the people waiting for service are (somehow) transported into another place to wait until the server is free. How they are sent from this room to the shop when the server is free depends on the logic of the program. Transactions are placed on the user chain by the LINK block. Later, when the transactions are to leave this chain and return to the CEC, the UNLINK block is used. The LINK block and the UNLINK block are complementary.

The general form of the LINK block is

```
◇        ▯LINK       ▯A,B,C
```

Operand *A* is the name or number of the user chain. Operand *B* shows the priority criterion; it can be FIFO (first in, first out), which places the transactions on the back of the user chain. This is the normal sequence of people joining a queue and being served in the order which they arrived. Operand *B* could also be LIFO (last in, first out) where transactions are placed on the user chain in front of the other transactions. Thus, when transactions are removed from the

user chain, they are taken in the reverse order from which they were placed on it. The *B* operand could also be an integer parameter. In this case, the transactions are placed on the user chain in ascending order according to the value of the parameter. For example, consider the block

```
◇         LINK        MYCHAIN,3PH
```

Suppose the transactions having the third halfword parameters 1, 5, -8, 16, and 2 were placed on the user chain. They would be placed on it in the order

(end of chain) 16 5 2 1 -8 (front of chain)

If a transaction arrived with the third halfword parameter having a value of 3, the new transaction would be placed on the user chain between the transactions whose third halfword parameters are 5 and 2.

Operand *C* is optional. When it is used, the LINK block is said to be in *conditional mode.* A transaction entering the LINK block either will be routed to the block with the label given by this operand or be placed on the user chain. This block is normally the next sequential block. When is a transaction placed on the user chain and when is it routed to this block? The answer is given by what is known as the user chain's *link indicator*. This is a switch that is either "set" (on) or "reset" (off). The link indicator is originally in a reset (off) position. When a transaction comes to a LINK block, it tests the condition of the link indicator. If it is on, it enters the block and turns the indicator on and then moves to the next sequential block and is not placed on the user chain. If the link indicator is on when the transaction comes to the LINK block, the transaction is removed from the CEC and placed on the user chain in accordance with operands *A* and *B*. Consider the following sequence of blocks:

```
◇         LINK        MYCHAIN,FIFO,NEXT
◇NEXT     SEIZE       MACH1
◇         ADVANCE     20,5.6
◇         RELEASE     MACH1
◇         -----
```

When the first transaction enters the LINK block, the link indicator is off; therefore the transaction turns the link indicator on and moves to the SEIZE block since this is the block with the label NEXT. Suppose that another transaction arrives to find the link indicator on. It will be taken off the CEC and placed on the user chain. When the RELEASE block is executed, something will happen to cause the link indicator to turn off again. This is where the UNLINK block is used. The general form of the UNLINK block is

```
◇         UNLINK      A,B,C,D,E,F
```

Operand *A* is the name or number of the user chain. This is the same name or number as the one used in the corresponding LINK block. Operand *B* specifies the label of the block where the unlinked transactions are to be routed. In the case of the example lines of code given here, it would be NEXT. When a transaction enters the UNLINK block, it checks to see if the user chain's link indicator is off. If so, it turns it on. If there is at least 1 transaction to be removed from the user chain, the link indicator will be on.

Consider the case of a barbershop with one barber. Customers arrive at a rate of 18 ± 6 minutes, and the barber works at a rate of 16 ± 8 minutes. The program to simulate this is as follows:

```
        SIMULATE
        GENERATE  18,6
        QUEUE     WAIT
        LINK      MYFIRST,FIFO,NEXT
NEXT    SEIZE     MACH1
        DEPART    WAIT
        ADVANCE   16,8
        RELEASE   MACH1
        UNLINK    MYFIRST,NEXT,1
        TERMINATE
        GENERATE  480
        TERMINATE 1
        START     1
        END
```

The *C* operand specifies the number of transactions to be removed from the user chain. The *C* operand can be the word ALL, which means that *all* transactions are to be removed from the user chain. Transactions are normally removed from the front of the user chain, which is to be expected. The *D* operand can be the word BACK. In this case, transactions are removed from the back of the user chain, i.e., in reverse order to the way one would expect them to be removed. The *D* operand could also be a Boolean variable. In this case, all the transactions are evaluated. Any transaction for which the Boolean variable is true is removed from the user chain. In the case where the BACK or Boolean variable option is used, the *E* operand must be omitted. The *D* operand can specify the name or number of a parameter to be evaluated by using the *E* operand in the test. In this case, an auxiliary operator is used: G, GE, LE, L, E, or NE. In case the auxiliary operator is omitted, it is taken as E by default. For example,

```
(a)     UNLINK      MYCHN,AWAY,,PH3,2
(b)     UNLINK G    MYFIRST,NEXT,,PH3,0
(c)     UNLINK LE   ABCD,EFGH,,PH1,0
```

The *F* operand specifies the name and number of a block used if the user chain is empty or if a conditional form of unlinking has been used and no transaction has been unlinked.

PREEMPT AND RETURN BLOCKS

The PREEMPT block is used for when a transaction is to take the place of another that is using a facility. In its simplest form, it is

```
        PREEMPT   A
```

where operand *A* is the name or number of a facility. If a facility is being used by another transaction when a transaction enters the PREEMPT block, it replaces the one that has control of the facility. The preempted transaction is placed on a chain known as the interrupt chain until the preempting transaction is removed

from the CEC. It will then continue to use the facility until it, too, is removed from the CEC. For example, a truck repair shop with one service bay will stop work on normal vehicles and fix an emergency vehicle. When the emergency vehicle is repaired, the repair shop will continue with the regular vehicles. When a transaction is finished preempting, the RETURN block is used to return control of the facility. In this simple form, the PREEMPT and RETURN blocks are similar to the SEIZE and RELEASE blocks. For example, consider the program

```
◇        □SIMULATE
◇        □GENERATE  □,,,1
◇        □SEIZE     □JOEB
◇        □ADVANCE   □20
◇        □RELEASE   □JOEB
◇        □BPUTPIC   □AC1
 A:  TIME IS NOW ***.**
◇        □TERMINATE □1
◇        □GENERATE  □,,,1
◇        □ADVANCE   □15
◇        □PREEMPT   □JOEB
◇        □ADVANCE   □20
◇        □RETURN    □JOEB
◇        □BPUTPIC   □AC1
 B:   TIME IS NOW ***.**
◇        □TERMINATE
◇        □START     □1
◇        □END
```

The first GENERATE block creates a transaction at time $t = 0$. This is placed on the FEC until time $t = 20$. The second GENERATE block preempts this transaction at time $t = 15$ and controls the facility JOEB until time $t = 35$. The first transaction is then placed on the FEC at this time for another 5 time units. The output from this program is

```
B: TIME IS NOW 35.00
A: TIME IS NOW 40.00
```

The general form of the PREEMPT block is

```
◇        □PREEMPT   □A,B,C,D,E
```

The *A* operand has been discussed. If the *B* operand is omitted, the PREEMPT block allows preemption on only "one level." This means that, if a transaction enters the PREEMPT block and another is using the facility, preemption will take place. However, if the transaction using the facility did so by preempting the facility and not by seizing it, no preemption will take place. If the *B* operand is PR, then preemption will take place providing the transaction using the facility has a lower priority than the one preempting, no matter how the one using the facility came to use the facility. The *C* operand will direct the preempted transaction to the block with this label. However, the transaction will still be in control of the facility so must either release or return the facility through the use of the RELEASE or RETURN blocks. The program shown next illustrates this procedure:

```
◇         SIMULATE
◇         GENERATE  ,,,1,5
◇         ADVANCE   10
◇         SEIZE     JOEB
◇         ADVANCE   20
◇         RELEASE   JOEB
◇         BPUTPIC   AC1
 A: TIME IS ***.**
◇         TERMINATE 1
◇         GENERATE  ,,,1,10
◇         ADVANCE   15
◇         PREEMPT   JOEB,PR,AWAY
◇         ADVANCE   20
◇         RETURN    JOEB
◇         BPUTPIC   AC1
 B:  TIME IS NOW ***.**
◇         TERMINATE 1
◇AWAY     BPUTPIC   AC1
 C:  TIME IS NOW ***
◇         RELEASE   JOEB
◇         TERMINATE 1
◇         START     1
◇         END
```

The preempting transaction takes control of the facility JOEB at time $t = 15$, so the transaction using it is directed to the block with the label AWAY. The only output from the program is the line

C: TIME IS NOW 15

The *D* operand specifies the name or number of the parameter into which the preempted transaction's remaining time on the FEC is placed when it is preempted. This parameter should be in floating-point mode or a warning message is sent to the screen. The *E* operand can be RE or left blank. If it is used, the preempted transaction is removed from competing for the preempted facility and sent to the block indicated by the operand *C*.

SAVAIL AND SUNAVAIL BLOCKS

The SAVAIL and SUNAVAIL blocks are used to make a storage available and unavailable, respectively, similar to the FAVAIL and FUNAVAIL blocks. When a transaction enters a SUNAVAIL block, the transactions currently using the storage will continue to use it. Only the transactions waiting to use the storage are denied access to the storage until the SAVAIL block is executed. The only operand allowed is the *A* operand, which is the name of the storage. For example,

```
◇         GENERATE  ,,,1
◇BACK     ADVANCE   RVNORM(1,200,30)
◇         SUNAVAIL  TUGS
◇         ADVANCE   RVNORM(1,20,3)
◇         SAVAIL    TUGS
◇         TRANSFER  ,BACK
```

would make the storage TUGS unavailable for a time given by the normal distribution with a mean of 20 and a standard deviation of 3.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

About the Author

John R. Sturgul is recognized as the world's leading authority on mine design using computer simulation models. He has designed more mines worldwide using simulation techniques than any other person. His model of the Lihir mine, located on an island northeast of Papua, New Guinea, is the first mine to be totally designed using a computer simulation model.

Sturgul received a bachelor of science degree in mining engineering from Michigan Technological University; a masters of science in mathematics from the University of Arizona, and a doctorate degree in mining engineering from the University of Illinois. Currently, he is professor of mining engineering at the University of Idaho where he was named 1999 Distinguished Professor.

In addition to his teaching duties, Sturgul organized and co-hosted the First International Symposium on Mine Simulation via the Internet–the first symposium of this type in any field. He is also co-editor of *The International Journal of Surface Mining, Environment, and Reclamation.* He has published extensively on mine system simulation and presented short courses on this subject throughout the world.

Sturgul established and maintains a website that will eventually link every university mining department in the world. The URL for the site is: http://www.uidaho.edu/mining_school. Additional information on mine system simulation can be found at:
http://www.udiaho.edu/~sturgul
His email address is: sturgul@uidaho.edu.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

Acknowledgments

The examples chosen here come from several sources. Some are from classical queueing theory or from early papers that concerned a mining problem that may have been solved by simulation but not by using GPSS. A few examples were done with former students who brought interesting problems back after summer work in mines. Some of the examples in Part II on the GPSS/H language are modifications of ones found in Schriber's classic 1974 textbook (with his kind permission), but the majority represent actual mining operations.

Many of the mining examples are from mines in Australia where I have spent quite a few years both in an academic situation and as a mining consultant. Other examples come from different mining operations in places as diverse as Chile and China. The responsibility of the choice of examples lies solely with me. I hope that the reader will benefit from the variety of examples presented here.

Thanks go to Bob Crain and James O. Henriksen of Wolverine Software for their continued support and comments. James Henriksen is the developer of GPSS/H (which explains the H in GPSS/H). Special thanks to him for not only providing me with the limited version of GPSS/H to be included with this book but also for making GPSS/H into the powerful tool that it is today. Wolverine Software also wrote the README.DOC file that is included on the CD. This file should be read first before attempting to use GPSS/H.

Dr. Thomas J. Schriber, the author of *Simulation with GPSS* and *Introduction to Simulation Using GPSS/H,* deserves special thanks. Few people programming with GPSS have been trained without using his textbooks (the first has been so well received that it is simply known as the "Red Book" or "Big Red"). Many of the examples from this textbook are excellent not only for learning the GPSS/H language but also for learning how to construct simulation models. Discussions with Tom have been highly productive. Some of the examples presented here were inspired by ones from his textbook and are so noted. In

one case, the exact example is used. Many of the animations were painstakingly done by visiting scholars to the University of Idaho from the Otto-von-Guericke-Technological-University, Magdeburg, Germany. These include Thomas Fliess, Chris Ritter, Ingolf Geist, Sebastian Bayer, and Frank Seibt. Prof.-Dr. Peter Lorenz of Otto-von-Gueriche-University also made practical suggestions for part of the book.

My wife, Alison, showed her editing skills by reading the manuscript and making numerous suggestions and changes.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

Mine Design: Examples Using Simulation

Installation

To install the software from the Mine Design CD, simply insert the CD into your CD-ROM drive. If you have disabled automatic recognition of software installation CDs, or if the installation procedure fails to start automatically, you should manually run setup.exe, contained in the root directory of the CD. To do so, click Start, Run, Browse; locate setup.exe on the CD; double click on it; and click OK in the Run dialog box.

The installation procedure will prompt you for the name of a directory to be used as the base directory for the software and examples contained on this CD.

The software will be installed in the base directory.

In addition, the following subdirectories will be created:

Subdirectory	Contents
GPSSH	Source files (.GPS) for GPSS/H mining examples
ANIMATION	Layout (.LAY) and trace (.PTF) files for sample animations
HTML	Problem statements, solutions, and discussion
P4DEMO	A demonstration of Wolverine's Proof Animation (tm) software

Note that installation of the Proof demonstration can be suppressed by performing a "custom" installation when you are given the opportunity during execution of the installation procedure.

The installation procedure will add a "Mining Examples" entry to the Start Menu. Clicking on this entry will reveal lower-level entries for viewing the HTML files and animations, an entry for establishing a command prompt window in which the sample GPSS/H programs can be run, and an entry for invoking the Proof demo, if installed.

Running GPSS/H Examples

To run a GPSS/H program named prog1, open the GPSS/H command prompt window and type the following command:

```
GPSSH PROG1
```

By default, the program will be run writing its interactive output to the screen. Noninteractive output, e.g., program listings and standard statistical output, will be placed in PROG1.LIS. Models can also be run under the control of the GPSS/H debugger, e.g.,

```
GPSSH PROG1 TVTNW
```

Note that successful execution of most of the GPSS/H examples contained on the CD requires reasonable values for model parameters. Failure to specify reasonable values will frequently result in GPSS/H execution error 401. This error occurs when the memory limits of Student GPSS/H are exceeded. For example, if you specify model parameters such that arrivals into a system occur more rapidly than they can be processed by the system, the contents of the system will become larger and larger the longer the model runs. As the contents of the system increase, more memory will be required. The model may run out of memory before a simulation run is completed. If so, error 401 will occur.

System Requirements for Running Proof Animation
==

To run Proof, your system must satisfy two requirements: (1) it must support Microsoft DirectDraw 3.0 or later, and (2) it must have reasonably up-to-date video drivers.

DirectDraw is built into Windows 98, Windows 2000, and Windows NT4 (Service Pack 4 or later required). DirectDraw is not built into Windows 95. For Windows 95, DirectDraw can be downloaded free of charge from Microsoft. Currently, downloads can be obtained from

www.microsoft.com/directx/download.asp

Download the "home user" version. (You are not a software developer.)

If you already have DirectDraw installed on your machine, you needn't download a later version. NT4 supports only DirectDraw 3, and Proof is written using DirectDraw 3 as a least common denominator.

Your video drivers must be "reasonably" up-to-date. If your video drivers are dated prior to 1998, you may encounter problems. Video drivers are generally available from two sources: the manufacturer of your machine's video hardware or the company from which you bought your computer. Video drivers are almost always available for download from a website. Generally, although not always, drivers that have been certified by Microsoft are preferable to uncertified drivers; however, probably less than half of the video drivers available are certified at present. You should not be afraid of running an uncertified driver.

Wolverine Software cannot provide you with DirectDraw, nor can we provide you with video drivers.

The installation procedure for Proof will perform tests to ascertain whether your system has adequate DirectDraw support. Proof will give you a step-by-step explanation of what it's doing. When the test procedure first comes up, it will be in a low resolution, usually 640 X 480. Before exiting the test procedure, you can use its Setup menu to try higher resolutions.

If at some later time you change video hardware or for some other reason need to retest your system's DirectDraw capabilities, you can do so by clicking on the "Test DirectDraw" entry under "Mine Design Examples" in the Start Menu.

If you experience problems with the cursor leaving trails of pixels as you move your mouse, try enabling Advanced Cursor Handling in Proof's Setup menu. If you have changed your default cursor to something other than the standard Windows arrow cursor, and the cursor you have chosen is implemented via software emulation, you may experience problems. If so, try changing back to the Windows standard cursor.

If you are running under Windows 95 or Windows 98, and Proof does not work properly, you may have to change the Hardware Acceleration settings for your video hardware. To do so, click Start, Settings, Control Panel, System, Performance, Graphics. You will see a 4-position switch. If this switch is set to the lowest position, no hardware acceleration is used, and DirectDraw will not work. If this switch is set to the highest position, you may experience problems with cursor motion on some machines.

The resolutions at which Proof can run are determined by available video memory and video driver policies for allocation of video memory. For example, your video hardware may have 2MB of video memory, but your video driver may make only 1MB available for DirectDraw applications. Some video drivers make assumptions based on the color depth you use for your desktop. If you have problems running Proof, it's worth quickly testing whether changing your desktop's color depth helps. For example, changing from 16-bit depth (65536 colors) to 8-bit (256 colors) may help. We have also seen instances in which increasing desktop color depth helps.

If you have difficulties with installation of Proof, send e-mail to mail@wolverinesoftware.com.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

PART III

Examples on the CD

Appendix A: Running the Programs and Animations

Click here to download CD-ROM Installer

THE SIMULATION PROGRAMS

When the CD-ROM that is included with this book is used to install the various files, two subdirectories are created on the hard disk. These are C:\GPSSH and C:\PROOF. In the event that some other subdirectory was selected when the CD-ROM is installed, then the appropriate subdirectories will apply.

The subdirectory GPSSH contains the files to run the examples. These files are

EX1A.GPS	EX9D.GPS	EX13D.GPS	EX22.GPS
EX1B.GPS	EX10A.GPS	EX14.GPS	EX23A.GPS
EX1C.GPS	EX10B.GPS	EX15.GPS	EX23B.GPS
EX2.GPS	EX10C.GPS	EX16A.GPS	EX24A.GPS
EX3.GPS	EX11A.GPS	EX16B.GPS	EX24B.GPS
EX4.GPS	EX11B.GPS	EX16C.GPS	EX25.GPS
EX5.GPS	EX11C.GPS	EX17.GPS	EX26.GPS
EX6A.GPS	EX12A.GPS	EX18.GPS	EX27A.GPS
EX6B.GPS	EX12B.GPS	EX19A.GPS	EX27B.GPS
EX7A.GPS	EX12C.GPS	EX19B.GPS	EX27C.GPS
EX7B.GPS	EX12D.GPS	EX19C.GPS	EX27D.GPS
EX8.GPS	EX12E.GPS	EX19D.GPS	EX28.GPS
EX9A.GPS	EX13A.GPS	EX19E.GPS	EX29.GPS
EX9B.GPS	EX13B.GPS	EX20.GPS	EX30.GPS
EX9C.GPS	EX13C.GPS	EX21.GPS	

The files, EX1A.GPS, EX1B.GPS, and EX1C.GPS correspond to Example 1. Example 2 has the single file EX2.GPS, etc.

To run any of the programs from the DOS prompt, key in the following:

```
C:\GPSS> GPSSH filename NODICT NOXREF <cr>
```

The **filename** is any file given above without the extension. If the .GPS file extension is included, the program will also run. NODICT and NOXREF are compiler directives and are optional. They stand for "no dictionary" and "no cross-reference."

The program will run and the screen will show the input prompts as given in each example. In the event that erroneous data are typed in, such as a letter for a number, an error message will appear on the screen and the program will have to be rerun.

Whenever a GPSS/H program is run, a .LIS file (**filename.LIS**) is created in the subdirectory where the GPSS/H programs are stored. This file is not needed for the examples as output is always sent to the screen and can be deleted. The effect of having NODICT and NOXREF is to cut down on the size of the .LIS file that is always created when the GPSS/H program is run. This .LIS file is created whenever the program is run and can be deleted. It is most useful for debugging purposes. Thus,

```
C:\GPSSH> GPSSH EX21 NODICT NOXREF <cr>
```

will run the program that goes with Example 21. The user will be prompted for input data as shown in the narrative for the example. If the compiler directives are omitted, the program will still run as before but the .LIS file is considerably longer.

One option for running the programs that is quite useful in debugging programs is the "TV" option. This is

```
GPSSH filename TV
```

The screen will be split into several parts. The top shows the program and location of the transaction being moved. The middle shows the current status of the transaction and the bottom is known as the dialog window. When this option is used, there will be the following on the bottom of the screen:

```
Ready !
:
```

Keying in s 1 (for "step 1") will show the movement of the first transaction after 1 step. The lines of code above and below the transaction will also be shown as well as the position of the transaction. Repeated keying of this (or s *n* to advance *n* steps) will show where the transaction being moved by the processor is. For example, there might be a message such as

```
XACT 1 POISED AT BLOCK 10.
     RELATIVE CLOCK: 2.5000
```

This shows that transaction 1 is poised at the 10th block at time 2.5000.

This mode can be very useful in debugging programs. This option will not be needed for running any of the examples here.

RUNNING THE ANIMATIONS

The following animation files are created and are found in the subdirectory C:\PROOF>.

Each of them will have the extensions .LAY and .PTF.

A1ANIM	A9DANIM	A15ANIM	A24AANIM
A2ANIM	A10AANIM	A16AANIM	A24BANIM
A3ANIM	A10BANIM	A16BANIM	A25ANIM
A4ANIM	A11AANIM	A16CANIM	A26ANIM
A5ANIM	A11BANIM	A17ANIM	A27AANIM
A6ANIM	A11CANIM	A18ANIM	A27BANIM
A7AANIM	A12AANIM	A19AANIM	A27CANIM
A7BANIM	A13AANIM	A19BANIM	A27DANIM
A8ANIM	A13BANIM	A20ANIM	A28ANIM
A9AANIM	A13CANIM	A21ANIM	A29ANIM
A9BANIM	A13DANIM	A22ANIM	A30ANIM
A9CANIM	A14ANIM	A23ANIM	

To run any of the animations, key in

```
C:\PROOF> PP/DEMO filename <cr>
```

Only the name of the animation file need be used, not the extension. For example, keying in

```
PP/DEMO A1ANIM <cr>
```

will cause the animation associated with Example 1 to be run.

The animations are run in what is known as "demo" mode. This gives the user the full power of PROOF but any changes that are made to the animation will not be saved. When an animation is run, the layout will be shown. A menu bar appears on the top of the screen. Using the mouse to click on GO starts the animation. By using the menu bar options, the animations can be changed. The menu bar has the following options:

```
TIME  0.0  SPEED  6.00  FASTER  SLOWER PAUSE GO VIEW FILE MODE
```

These options can all be used when viewing the animations.

TIME — This allows the animation to go forward or back in time. When this is used, a bar is shown in the bottom center of the screen to prompt the user to input a time jump.

SPEED — By default, the animations have an initial viewing "speed" of 6.00. This corresponds to having 6 time units of the animation take 1 second of viewing time. Thus, an animation that uses 100 simulated time units will be shown on the screen in 16.67 seconds. If this option is used, the animations can be speeded up or slowed down by keying in a new viewing speed.

FASTER	This causes the animations to run faster. This will have the same effect as the SPEED option, but now the speed increases by about 10% per click of the mouse.
SLOWER	This has the same effect as FASTER but in reverse.
PAUSE	This will pause the animation until GO is clicked on.
VIEW	There are many options with this, and a pull-down menu will appear when this is clicked. Some of these options cannot be used in demo mode.

Select View	This allows you to select any previously defined views (but no previously defined views are included with the animations).
Define View	It is not possible to define new views in demo mode.
Update Existing View	This option will not work in demo mode.
Delete Existing View	This option will not work in demo mode.
Capture New View	This option will not work in demo mode.
/Split View	This lets you split the current view into 2 or more independent windows. Each window can be manipulated independently by using other View menu options. The options are to split the screen horizontally or vertically.

Unsplit	Deletes a window.
Resize	Changes the size of a window.
Undo	Undoes the previous Split, Unsplit, or Resize.

Pan	This allows one to pan the animation.

Left 25%	Shift the window 25% left.
Right 25%	Shift the window 25% right.
Up 25%	Shift the window 25% up.
Down 25%	Shift the window 25% down.

Zoom	Zoom in or out of the animation. This goes from 0.9× to 3.0×.

	Zoom Box	This allows one to zoom by using a zoom box. When this option is selected, a zoom box appears on the screen. The boundaries can be clicked on to increase or decrease the box. There is a cross hair in the center to move the zoom box to a new location.
	Rotate	This rotates the animation 10 degrees at a time.
	Grid, Inspect, and Refresh	These are not used in demo mode.
FILE	This has the following options.	
	Open Layout & Trace	Select from the stored files of .LAY and .PTF files.
	Open Layout Only	Select from the stored .LAY files.
	Change Directory	Not needed for the examples unless they are stored in a directory different from C:\PROOF.
	Change Disk	Not needed for the examples unless they are stored in a directory different from C:\PROOF.
MODE	This has several options but the only ones that are used in demo mode are the following.	
	Run	Starts the animation. The line menu appears on the top of the screen, if it is not there already.
	Exit	Exits the animation.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

Appendix B: SNAs Used in Book

Some of the SNAs given below have "*j*" after them. This is either a positive integer or an opening parenthesis followed by a legal GPSS/H name followed by a closing parenthesis. For example, Qa*j* could be QA5 or QA(WAIT) if the queue was referred to by a 5 for its operand or had the operand WAIT. Alternatively, one could write this as QA(5) or QA$WAIT. The use of the dollar sign is considered obsolete but is still supported in GPSS/H.

When an SNA is "true," this means that the value is 1, otherwise the value is 0.

AC1	Absolute clock in simulation.
BV*j*	The value of the Boolean variable (either 0 or 1).
C1	Relative clock of the simulation.
FN*j*	Value of function.
FRN*j*	Random number in the interval 0 < random number < 1.
K*j*	Integer constant (obsolete).
M1	Transit time of transaction. This is the difference between the absolute clock and the mark time of the transaction. The mark time is when the transaction left the GENERATE block.
MP*j*PL	This is computed as the difference between the current value of the absolute clock and what is stored in the floating-point parameter *j*. This is normally the absolute clock time that the transaction entered a MARK block. One can also have MP*j*PB, M*j*PF, and MP*j*PH, but their use is rare since the absolute clock is stored as a floating-point value.
N*j*	The total number of transactions to have entered the block.
PR	Transaction priority. This can be in the interval −2,147,483,632 to +2,147,483,632.

RN*j*	Random number. This returns a random number. If it is used with a function, the random number is in the interval 0 ≤ random number < 1. If it is used in any other context, the number is in the interval 0 ≤ random number ≤ 999.
TG1	Current value of terminate counter. This is originally specified by the START statement.
V*j*	Value of variable (or Fvariable). This is a floating-point value for both types.
W*j*	The current block count.
XID1	The number of the current transaction. Each transaction has a unique number.

FACILITIES

F*j*	True if captured.
FC*j*	Number of times facility has been captured.
FNS*j*	True if not seizeable.
FNU*j*	True if not in use.
FR*j*	Utilization in parts per thousand.
FS*j*	True if seizeable.
FT*j*	Average time per transaction when in facility.
FU*j*	True when in use.

LOGIC SWITCHES

LR*j*	True when logic switch is reset.
LS*j*	True when logic switch is set.

MATRICES

MB*j*(row, col)	The value of the (row, column) of the byte-word matrix savevalue.
MH*j*(row, col)	The value of the (row, column) of the halfword matrix savevalue.
ML*j*(row, col)	The value of the (row, column) of the floating-point matrix savevalue.
MX*j*(row, col)	The value of the (row, column) of the fullword matrix savevalue.

Caution: Do not use MF*j*(row, col) for the value of the fullword matrix savevalue.

PARAMETERS

P*j*	The value of a parameter where the mode is determined at run time. This is not a recommended use of parameters.
PB*j*	The value of the transaction's byte-word parameter. This is restricted to values between −128 and + 127.
PF*j*	The value of the transaction's fullword parameter. This is restricted to values between −2,147,483,648 and +2,147,483,647.
PL*j*	The value of the transaction's floating-point parameter. This value depends on the computer.

QUEUES

Q*j*	Current content.
QA*j*	Average contents.
QC*j*	Total entry count.
QM*j*	Maximum entry count.
QT*j*	Average time in queue for every transaction, even those that have a zero residence time.
QX*j*	Average time in queue for only those transactions that were held up in the queue.
QZ*j*	Number of transactions that passed through the queue in zero residence time.

STORAGES

R*j*	Remaining storage capacity.
S*j*	Units of storage currently in use. Note that if the original storage was m, then R*j* + S*j* = m. The original storage, m, is not an SNA.
SA*j*	Average contents.
SC*j*	Total number of entries.
SE*j*	True if storage is empty.
SF*j*	True if storage is full.
SM*j*	Maximum contents.
SNE*j*	True if storage is not empty.
SNF*j*	True if storage is not full.
SR*j*	Utilization of storage expressed in parts per thousand.
ST*j*	Average time per unit in storage.

TABLES

TB*j*	The mean value.
TC*j*	The number of entries.
TD*j*	The standard deviation.

SAVEVALUES

X*j*	Value of fullword savevalue. The use of this form is not recommended.
XB*j*	Value of byte-word savevalue. These values are in the range −128 to +127.
XF*j*	Value of fullword savevalue. These values are in the range -2,147,483,648 to +2,147,483,647.
XH*j*	Value of halfword savevalue. These values are in the range -32,768 to +32,767.
XL*j*	Value of floating-point (real) value of savevalue. Maximum and minimum values depend on the size of the computer.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

Appendix C: Historical GPSS/H Format

GENERAL FORM OF A GPSS/H PROGRAM LINE

The format for coding a line of GPSS/H code has evolved over the years. The general form of a line will look as follows:

```
label   operation   operand(s)   comment
```

Since GPSS was introduced when communication with the computer was done by using 80-column punch cards, at one time the coding was very strict. This is known as "fixed" format and the rules were as follows:

- Position 1 is blank.
- The label is in positions 2 through 6 so must be 5 or less characters in length.
- Position 7 is blank.
- The operation is in position 8 through 17.
- Position 18 is blank.
- The operands start in position 19 or further (but not beyond position 25).
- Comments begin anywhere after the operands as long as there is a single space between them.

Today, GPSS/H supports a free format as well as the older fixed format. The main differences between free format and fixed format are that the labels and operands can be up to 8 characters in length and the operands can begin in an arbitrary position. In general, the form is as follows:

- Labels begin in either position 1 or 2 (in this book, they begin in position 2). They can be from 1 to 8 characters in length. Because GPSS/H has many reserved words that are normally from 1 to 3 characters in length with a few as long as 4 characters in length, it is good programming practice to have labels 5 or more characters in length.

- Operations normally start in position 8 but may begin in a position up to OPERCOL -1. OPERCOL is 25 by default. In this book, operations begin in position 11.
- Operands begin in position OPERCOL or less.
- Comments begin separated from operands by one space. However, there is one case–when specifying a MACRO–in which comments must begin after two spaces, so the practice in this book is to always have two spaces between operands and comments.

The OPERCOL statement comes at the start of the program and has the form

```
OPERCOL  n
```

where *n* is an integer greater than 25. This statement tells the compiler that the operands will begin in column *n* or before. Thus,

```
OPERCOL   30
```

tells the compiler that the operands will begin in position 30 or before. Sometimes this statement is useful to align code, as in complicated DO loops.

Copyrighted Materials
Copyright © 2000 Society for Mining, Metallurgy, and Exploration (SME)
Retrieved from www.knovel.com

INDEX

Note: *f* indicates figure, *t* indicates table.

Index Terms	Links		
A			
ADVANCE block	87		
caution in using functions with	137		
exercises	91		
Ampervariables	241		
and DO loop	243		
and ELSE statement	249		
and ENDIF statement	249		
exercises	251		
and GETUST statement	247	250	
and GOTO statement	249	250	
and HERE statement	250		
and IF statement	248	250	
Animations (on CD-ROM)	353		
Arithmetic expressions	153	284	
exercises	155		
Arithmetic operations	47	283	
Art exhibit example	90		
ASSEMBLE block	321		
Assembly line example	49	308	331
Assembly sets	321		
exercises	333		
ASSIGN block	182		
decrement mode	182		
general form	186		
increment mode	182		
Attributete-valued functions	141		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms **Links**

B

Barbershop example 21 35 38 47 49 54 56 67 97 113 154 159
BGETLIST block 341
Blodcs 51
Boolean expressions (rules for evaluation) 293
Boolean operators 290
Boolean variables 289
  and Boolean operators 290
  exercises 297
  and logical operators 292
  referencing 293
  and relational operators 290
BPUTPIC block 83
  See also PUTPIC statement
  exercise 85
BPUTSTRING block 341
Brand loyalty example 275
BUFFER blodc 301
BVARlABLE statements 289
Byte-word savevalues 207

C

Car inspection example 116
Car repair example 102
Cars-and-junction example 81
CD-ROM
  animations 353
  simulation programs 351
CLEAR statement 129
  exercises 131
  and savevalues 211
Coding format 361

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links	

Comparison operators. *See also* Relational operators		
Conditional TRANSFER block	66	
Continuous functions	139	
Control statements	51	
COUNT block	258	
with facilities and storages	261	
with logic switches	259	
Cyclic queues	23	

D

DEPART block	96	
exercises	98	
Discrete functions	133	
Discrete systems	7	19
Distributions		
built-in functions	167	
exercises	171	
exponential and normal ones other than built-in ones	169	
nonsymmetrical	27	
normal	168	
Poisson	167	
symmemcal	27	
triangular	168	
DO loop	243	
Drilled boxes example	322	

E

ELSE statement	249	
END statement	48	
ENDIF statement	249	
ENTER block	111	
exercises	120	
Entities	255	
Entity class	255	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
EQU statement	188		
Examples (on CD-ROM)	351		
F			
Facilities	260		
SNAS	358		
Factory parts example	229	257	309
FAVAIL block	342		
FILEDEF statement	82		
filename	351		
FIX mode conversion	209		
Fixed format	42	361	
Floating-point savevalues	207		
FLT mode conversion	209		
Fortran	10		
Free format	46	361	
Fullword savevalues	207		
FUNAVAIL block	342		
Functions			
attribute-valued	141		
built-in	167		
caution in using with ADVANCE and GENERATE blocks	137		
continuous	139		
discrete	133		
exercises	142	171	
list	141		
normal distribution	168		
Poisson distribution	167		
referencing	135		
triangular dismbution	168		
use with SNAS	140		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms Links

G

Index Terms	Links	
GATE block	219	289
in conditional transfer mode	220	
exercises	224	
GATHER block	330	
Gaussian disttibution. *See also* Normal distribution		
GENERATE block	51	55
caution in using functions with	137	
creating transactions	52	
exercises	59	
GETLIST statement	247	250
GOTO statement	249	250
GPSS	3	
benefits	3	
nonprocedural language	4	
origins	4	
GPSS/H	3	
arithmetic operations	47	
coding format	361	
defined	7	
END statement	48	
fixed format	42	361
and Fortran	10	
free format	46	361
input format	36	
internal clock	51	
learning	9	
modern versions	8	
origins	8	
output	39	
reasons for using	7	10
SIMULATE statement	48	
and traditional languages	10	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
H			
Halfword savevalues	207		
Hand simulation	49		
Harbor example	189		
Hardware store example	115	117	184
HERE statement	250		
I			
IA mode table	202		
IF statement	248	250	
INITIAL statement	211		
Input format	36		
L			
LEAVE block	116		
exercises	120		
LINK block	344		
conditional mode	345		
Link indicator	345		
List functions	141		
LOGIC block	217		
Logic switches. *See also* GATE block, LOGIC block			
SNAs	358		
with SELECT or COUNT block	259		
Logical expressions	289		
Logical operators	292		
LOOP block	187		
M			
M1 and tables	200		
Machine production example	271		
Macro definition statements	335		
Macros	335		
MARK block	201		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links	

MATCH block	332	
Mathematical functions	152	
exercises	155	
Matrices	267	
defining	267	
exercises	278	
initial values	268	
savevalue block	270	
SNAs	358	
transition	275	
types of	267	
Military time	312	
example	312	
MSAVEVALUE block	270	
Multiple servers. *See also* ENTER block, LEAVE block, STORAGE statement		
Multiple START statements	127	

N

Negative times	53	
Newspaper boy's problem example	212	
Nonprocedural language	4	9
Nonsymmemcal distributions	27	
Normal distribution	168	

O

Operands	284	
exercises	287	
Other simulation languages	10	
Output	39	

P

Parameters	179	
ASSIGN block	182	
EQU statement	188	

This page has been reformatted by Knovel to provide easier navigation.

Parameters (*Cont.*)
exercises 193
giving names 183
LOOP block 187
SNAs 359
PERT diagrams 324 325*f* 327*f*
Poisson distribution 167
PREEMPT block 346
PRINT block 286
PUTPIC statement 79
See also BPUTPIC block
exercise 85
PUTSTFUNC statement 77
exercises 83

Q

QTABLE mode 203
QUEUE block 93
exercises 98
Queueing theory 13 19 23
Queues
cyclic 23
SNAs 359
Quickline example 261

R

Random numbers 53
Referencing functions 135
Relational operators 290
RELEASE block 102 347
exercises 105
Repair shop example 302
RESET statement 130
and savevalues 211
exercises 131
RETURN block 347

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			
RMULT statement	130			
exercises	131			
RT mode table	202			

S

Index Terms	Links			
SAVAIL block	348			
SAWVALUE block	208			
Savevalues	207			
byte-word	207			
and CLEAR	211			
exercises	214			
floating-point	207			
FIX mode conversion	209			
FLT mode conversion	209			
fullword	207			
halfword	207			
INITIAL statement	211			
and RESET	211			
SAVEVALUE block	208			
SNAs	360			
types	207			
SEIZE block	99			
exercises	105			
SELECT block	255			
exercises	264			
with facilities and storages	260			
with logic switches	259			
in MIN or MAX mode	259			
Sequential times	90			
Shopping example	136	138		
Shovel-loading truck example	29	60	103	119
	221	337		
SIMULATE statement	48			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	**Links**			

Index Terms	Links			
Simulation				
applications	19			
barbershop example	21	35	38	47
	49	54	56	67
	97	113	154	159
information needed and provided	20			
vs. mathematical solution	24			
mining operations	14			
nonsymmetrical and symmetrical distributions	27			
other languages	10			
and personal computers	15			
programs (on CD-ROM)	351			
queueing theory	13	19	23	
shovel-loading truck example	29	60	103	119
	221	337		
tool crib example	22			
SNAs. *See also* Standard numerical attributes				
Soft drink brand loyalty example	275			
Spare truck problem	119			
SPLIT block	307			
exercises	317			
general form	316			
Standard numerical attributes	79	151		
See also Parameters				
and arithmetic expressions	153			
associated with STORAGE statement	114			
exercises	155			
facilities	358			
list of mathematical functions	152			
logic switches	358			
M1 and tables	200			
matrices	358			
parameters	359			
partial list of	152			
queues	359			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
Standard numerical attributes (*Cont.*)			
savevalues	360		
storages	359		
and tables	197	200	360
used in this book	357		
using functions with	140		
START statement	57	127	
exercises	131		
multiple	128		
Statements	51		
STORAGE statement	112		
exercises	120		
SNAs	359		
Student example	88		
SUNAVAIL block	348		
symmetrical distributions	27		
T			
TABLE statement	197		
Tables			
exercises	203		
IA mode table	202		
and M1	200		
MARK block	201		
QTABLE mode	203		
RT mode table	202		
and SNAs	197	200	360
TABLE statement	197		
TABULATE block	198		
TABULATE block	198		
TERMINATE block	57		
exercise	59		
TEST block	157	217	289
exercises	162		
normal mode	161		
refusal mode	158		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms | Links

Index Terms	Links		
Time caution	90		
Timer example	83		
Tire supply example	244	248	
Transactions	51		
assembly sets	321		
cloning	307		
creating	52		
parameters attached to each	179		
TRANSFER block	65		
in ALL mode	228		
conditional	66		
exercises	68	235	
in function mode	230		
in parameter mode	232		
in pick mode	227		
in simultaneous mode	234		
in subroutine mode	233		
unconditional	65		
TRANSFER BOTH block	67		
TRANSFER SIM block	234		
Triangular distribution	168		
Truck inspection and refueling example	268		
Truck repair example	231	293	332
24-hour time	312		
example	312		

U

Index Terms	Links
Unconditional TRANSFER block	65
UNLINK block	344

W

Index Terms	Links
Writing to a file	82

This page has been reformatted by Knovel to provide easier navigation.