
SIMSCRIPT III[®]

Programming Manual

CACI Products Company

SIMSCRIPT III Programming Manual,
CACI Advanced Simulation Lab

Copyright © 2007 CACI Products Company.

All rights reserved. No part of this publication may be reproduced by any means without written permission from CACI.

For product information or technical support contact:

CACI Products Company
1455 Frazee Road, Suite 700
San Diego, CA 92108
Phone: (619) 881-5806
Email: simscrip@caci.com

The information in this publication is believed to be accurate in all respects. However, CACI cannot assume the responsibility for any consequences resulting from the use thereof. The information contained herein is subject to change. Revisions to this publication or new editions of it may be issued to incorporate such change.

Table of Content

PREFACE	A
1 INTRODUCTION TO SIMSCRIPT III	1
1.01 LANGUAGE BASICS	3
1.02 CHARACTER SET	4
1.03 COMMENTS	5
1.04 SCIENTIFIC NOTATION AND PUNCTUATION	6
1.05 NAMED AND ENUMERATED CONSTANTS	7
1.06 BASIC DATA TYPES.....	8
1.07 TEXT AND ALPHA.....	8
1.08 VARIABLES AND ARRAYS	9
1.09 EXPRESSIONS	9
1.10 BASIC STATEMENTS	10
1.11 LOOPS	11
1.12 FUNCTIONS AND SUBROUTINES	11
1.13 ARGUMENT CHECKING.....	12
1.14 REFERENCE MODE	14
1.15 INPUT AND OUTPUT.....	15
2 OBJECT-ORIENTED PROGRAMMING	19
2.01 CLASSES AND OBJECTS	21
2.02 ATTRIBUTES.....	22
2.03 METHODS	25
2.04 GROUPING OBJECTS IN SETS.....	28
2.05 ARRAYS OF SETS.....	30
2.06 INHERITANCE	31
3 OBJECT-ORIENTED DISCRETE SIMULATION	37
3.01 PROCESS METHOD.....	39
3.02 RANDOM NUMBER GENERATION	41
3.03 STATISTICS	41
4 MODULARITY	43
4.01 SUBSYSTEMS.....	43
4.02 SOURCE CODE ORGANIZATION	47
5 LIBRARY.M	49
5.01 MODE CONVERSION	50
5.02 NUMERIC OPERATIONS.....	52
5.03 TEXT OPERATIONS	57
5.04 INPUT/OUTPUT	59
5.05 RANDOM-NUMBER GENERATION	63
5.06 SIMULATION	67
5.07 MISCELLANEOUS.....	71
6 EXAMPLE PROGRAMS	73
6.01 EXAMPLE 1 - GAS STATION WITH 2 ATTENDANTS.....	73
6.02 EXAMPLE 2 - GAS STATION WITH 2 ATTENDANTS AND 2 GRADES OF GASOLINE.....	77
6.03 EXAMPLE 3 - A BANK WITH SEPARATE QUEUE FOR EACH TELLER	83
6.04 EXAMPLE 4 - A HARBOR MODEL	89
6.05 EXAMPLE 5 - THE MODERN BANK	92
6.06 EXAMPLE 6 - A JOB SHOP MODEL	98
6.07 EXAMPLE 7 - A COMPUTER CENTER STUDY.....	106

PREFACE

This document contains information on CACI's new SIMSCRIPT III, Modular Object-Oriented Simulation Language, designed as a superset of the widely used SIMSCRIPT II.5 system for building high-fidelity simulation models.

CACI publishes a series of manuals that describe the SIMSCRIPT III Programming Language, SIMSCRIPT III Graphics and SIMSCRIPT III SimStudio. All documentation is available on SIMSCRIPT WEB site <http://www.caciasl.com/products/simscript.cfm>

- *SIMSCRIPT III Programming Manual* – this manual - is a short description of the programming language and a set of programming examples.
- *SIMSCRIPT III User's Manual* – is a detailed description of the SIMSCRIPT III development environment: usage of SIMSCRIPT III Compiler and the symbolic debugger from the SIMSCRIPT Development studio - Simstudio, and from the Command-line interface.
- *SIMSCRIPT III Reference Manual* - A complete description of the SIMSCRIPT III programming language constructs in alphabetic order. Graphics constructs are described in SIMSCRIPT III Graphics Manual.
- *SIMSCRIPT III Graphics Manual* — A detailed description of the presentation graphics and animation environment for SIMSCRIPT III

Since SIMSCRIPT III is a superset of SIMSCRIPT II.5, a series of manuals and text books for SIMSCRIPT II.5 language, Simulation Graphics, Development environment, Data Base connectivity, Combined Discrete-Continuous Simulation, can be used for additional information:

- *SIMSCRIPT II.5 Simulation Graphics User's Manual* — A detailed description of the presentation graphics and animation environment for SIMSCRIPT II.5
- *SIMSCRIPT II.5 Data Base Connectivity (SDBC) User's Manual* — A description of the SIMSCRIPT II.5 API for Data Base connectivity using ODBC
- *SIMSCRIPT II.5 Operating System Interface* — A description of the SIMSCRIPT II.5 APIs for Operating System Services
- *Introduction to Combined Discrete-Continuous Simulation using SIMSCRIPT II.5* — A description of SIMSCRIPT II.5 unique capability to model combined discrete-continuous simulations.
- *SIMSCRIPT II.5 Programming Language* — A description of the programming techniques used in SIMSCRIPT II.5.
- *SIMSCRIPT II.5 Reference Handbook* — A complete description of the SIMSCRIPT II.5 programming language, without graphics constructs.

- *Introduction to Simulation using SIMSCRIPT II.5* — A book: An introduction to simulation with several simple SIMSCRIPT II.5 examples.
- *Building Simulation Models with SIMSCRIPT II.5* —A book: An introduction to building simulation models with SIMSCRIPT II.5 with examples.

The SIMSCRIPT language and its implementations are proprietary program products of the CACI Products Company. Distribution, maintenance, and documentation of the SIMSCRIPT language and compilers are available exclusively from CACI.

Free Trial Offer

SIMSCRIPT III is available on a free trial basis. We provide everything needed for a complete evaluation on your computer. **There is no risk to you.**

Training Courses

Training courses in SIMSCRIPT III are scheduled on a recurring basis in the following locations:

San Diego, California
Washington, D.C.

On-site instruction is available. Contact CACI for details.

For information on free trials or training, please contact the following:

CACI Products Company
1455 Frazee Road, suite 700
San Diego, California 92108
Telephone: (619) 881-5806
www.caciasl.com

1 Introduction to SIMSCRIPT III

The SIMSCRIPT III programming language is a superset of SIMSCRIPT II.5 with significant new features to support modular object-oriented simulation programming.

It preserves existing world-view and the powerful data structures: entities, attributes and sets, process and event-oriented discrete simulation of SIMSCRIPT II.5, and adds the new, more elaborated data structures and concepts like classes, methods, objects, multiple inheritance and process-methods, to support object-view and object-oriented process and event discrete simulation. Object types are defined with the class which can be instantiated, they may have methods which describe object behavior, and may contain special process-methods with time elapsing capabilities which can be scheduled for execution in defined instances of time. Both, world-view and object-view can exist in the same model, or a modeler may decide to use entirely object-view or a world-view only.

SIMSCRIPT III model can consist only of main module (preamble and implementation), but larger models should be designed with modularity in mind, as a main module with a set of subsystems to facilitate code reuse and team work development. Modularity can be easily added to an existing SIMSCRIPT II.5 model, defining it as a main module (system) and adding new subordinate modules (subsystems/packages).

SIMSCRIPT III includes all standard language elements and can be used as a general-purpose object-oriented programming language with English-like syntax. In addition, it includes powerful support for building simulation models with interactive GUI, presentation graphics and animation. Building SIMSCRIPT III graphical models is explained in the SIMSCRIPT III Graphics Manual.

The SIMSCRIPT III models are developed inside “Simstudio”, an Integrated Development Environment (IDE) which incorporates automatic project builder, syntax colored text editors and graphical editors for GUI elements: dialog boxes, menus, palettes, icons, graphs. Building SIMSCRIPT III projects using Simstudio is described in SIMSCRIPT III User’s Manual.

Chapter 1 describes basic language elements and related enhancements like support for Latin character set, named constants, argument type checking, multiple-line comments, and reference modes.

Chapter 2 introduces classes, objects, multiple inheritance, object and class methods, and for support of object-oriented programming

Chapter 3 describes a process-method which can be used for process and event-based discrete simulation. It also describes accumulate and tally statements for statistics collection.

Chapter 4 explains how SIMSCRIPT III programs can be designed as a set of modules or “subsystems”, and elaborates on data scope and name resolution. Subsystem may contain of public and private declarations and implementation. Public data and function/method

declaration defines subsystem's interface with the system and other subsystems while private data and method declarations hide implementation details.

Chapter 5 lists the "system" routines, variables, and constants, which are defined by SIMSCRIPT III's **library.m** subsystem and are implicitly imported into every module. Other system modules like GUI.m, SDBC.m, Continuous.m are imported on demand and described in specialized manuals.

Chapter 6 provides SIMSCRIPT III example programs, rewritten from SIMSCRIPT II.5. Original programs are from the book: Building Simulation Models with SIMSCRIPT II.5. These examples illustrate use of classes, objects, inheritance, subsystems, creating simulations with process-methods and collection of statistics on object attributes.

1.01 Language Basics

SIMSCRIPT III is Modular Object-Oriented Language which can be used for general purpose program development. It is especially suited for building discrete-event and process based simulation models.

SIMSCRIPT program consists of a main module and zero or more imported subordinated modules called subsystems or packages. Main module consists of a block of declarations known as the “preamble,” followed by one or more functions and routines, one of which is named **main**. The simplest main module without a preamble in SIMSCRIPT would be:

```
main
  print 1 line thus
    Hello World !
end
```

or with a preamble:

```
preamble
  define Greeting as a text variable
end

main
  Greeting = "Have a nice day!"
  write Greeting as T *,/
end
```

Declarations in the preamble are “global,” i.e., they apply to every routine in the module. Declarations within a routine are “local,” i.e., they apply only to the routine in which they are declared. Other levels of scope: object scope, class scope, public and private scope of the subsystem will be described in the chapters that follow.

Program execution begins with the first statement in **main** and continues until **main** returns or a **stop** statement is executed.

Programmer-defined names and language keywords are case insensitive. A programmer-defined name is a sequence of letters, digits, periods, dollar signs, and underscores. Except for **and**, there are no reserved words.

1.02 Character Set

The character set supported by SIMSCRIPT III is Latin1, more formally ISO 8859-1, which is an 8-bit character encoding that includes ASCII as a subset. Values 0 to 127 are defined by ASCII, and values 128 to 159 are non-printable Latin1 characters. Values 160 to 255 are printable Latin1 characters and include these letters,

À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ ß
à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ

and these special symbols:

¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ × ÷

Words in the following languages can be represented using the Latin1 character set: Afrikaans, Albanian, Basque, Catalan, Danish, Dutch, Faroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Romansh, Scottish Gaelic, Spanish, Swahili, and Swedish.

Latin1 characters can appear in SIMSCRIPT III source code and in program input and output, and can be stored in alpha and text variables. For example:

```
define CAFÉ as a text variable
CAFÉ = "Le Loir dans la Théière"
write CAFÉ as "Le nom du café est ", t *, /
```

The character set supported by SIMSCRIPT II.5 is ASCII, which is a 7-bit character code.

1.03 Comments

SIMSCRIPT III supports single and multiple line comment. A single line comment begins with a pair of consecutive apostrophes and terminates at the end of the line or upon reaching another pair of apostrophes on the same line. The comments are for human readers; the compiler ignores them. This block of code,

```
if N = 0      '' variable N is uninitialized
    read     '' number of elements into '' N
always
reserve X as N '' allocate the array
```

is equivalent to:

```
if N = 0
    read N
always
reserve X as N
```

Multiple line comment which may span several lines begins with slash-tilde */~* and ends with tilde-slash *~/*. It can also be used in a single line as in the example:

```
if N = 0      '' variable N is un-initialized
    read     '' number of elements into '' N
always
reserve X as N /~ allocate the array ~/
```

Single line comments can be nested inside multiple line comments. This makes it convenient to “comment out” a block of code which may itself contain comments:

```
/~ assume the array is already allocated
if N = 0      '' variable N is un-initialized
    read     '' number of elements into '' N
always
reserve X as N /~ allocate the array ~/
~/
```

Comments may be nested to any depth.

1.04 Scientific Notation and Punctuation

A numeric constant is a sequence of digits with an optional period (i.e., decimal point) and optional scientific notation.

Floating point variables and constants can be expressed in scientific notation. For example:

```
Define X, Y, and Z as double variables

X = 3.5026E5           '' assign 350260.0 to X
Y = 1.72e-03          '' assign 0.00172 to Y
Z = -27.641e+2        '' assign -2764.1 to Z
```

The letter **E** may be omitted from an input value (e.g., **4.82-7**), but it is required when expressing the value as a constant (e.g., **4.82e-7**). Space characters are not permitted within the constant.

SIMSCRIPT III permits periods and semicolons to enhance the readability of statements. When used for this purpose, these punctuation characters are ignored by the compiler. In this example a period is placed at the end of the **define** statement and a semicolon after each assignment statement:

```
Define X, Y, and Z as double variables.

X = 3.5026E5;          '' assign 350260.0 to X
Y = 1.72e-03;         '' assign 0.00172 to Y
Z = -27.641e+2;       '' assign -2764.1 to Z
```

1.05 Named and Enumerated Constants

Named constants are defined with a specified value in **define constant** statement. More than one constants can be defined in a single statement, for example:

```
define Max_Capacity = 100 as a constant  
or  
define Min_Capacity = 5 and Max_Capacity = 100 as constants
```

Named constants are not limited to integers, for example:

```
define cm_per_inch = 2.54, cm = "centimeters" as constants  
write 12 * cm_per_inch, cm as d(5,2), " ", t *
```

The above **write** statement writes the number of centimeters in one foot:

```
30.48 centimeters
```

If the value of a named constant is unspecified, it is assigned the integer value that is one greater than the value of the preceding integer constant in the statement, or assigned a value of one if there is no preceding integer constant. In the following example, the constants named **F**, **D**, **C**, **B**, and **A** represent letter grades and are assigned values zero through four, and the constants **Idle**, **Busy**, and **Terminated** are given values one to three.

```
define F = 0, D, C, B, A as constants  
define Idle, Busy, and Terminated as constants
```

Named constants declared in a preamble are “global,” that is, they are accessible to every routine of the module. Named constants declared in a routine are “local,” that is, they are accessible only within the declaring routine.

Similar mechanism for creating named constants is a **define to mean** or **substitute** statement. For example, after the following statement, each occurrence of the name **Max_Capacity** is replaced by the number **100**.

```
define Max_Capacity to mean 100
```

1.06 Basic Data Types

There are several basic data types, called “modes”: **integer**, **real**, **double**, **alpha**, **text**, and **pointer**. **Integer** is implemented as a signed 32-bit or 64-bit value, depending on a platform. **Real** and **double** are single- and double-precision floating-point values, respectively. **Pointer** is a generic (un-typed) reference value, implemented as a 32-bit or 64-bit address, depending on a platform. SIMSCRIPT III supports also Reference mode, fully described in chapter 1.14.

1.07 Text and Alpha

Alpha holds one 8-bit character; an **alpha** constant is surrounded by quotation marks, e.g., "B". **Text** is a dynamic string holding a sequence of zero or more characters; a **text** constant is also surrounded by quotation marks: "Hello, world!". Built-in functions are available for string operations like `concat.f`, `upper.f` and type conversions like `toa.f`, `atof.f`.

A text expression can be assigned to an alpha variable and passed to an alpha argument, and its value is converted automatically by an implicit call of `toa.f`. Likewise, an alpha expression can be assigned to a text variable and passed to a text argument, and its value is converted automatically by an implicit call of `atof.f`. This notational convenience permits, for example, an alpha variable named **A** to be converted to uppercase by

```
A = upper.f(A)
```

A text expression can be compared with an alpha expression as part of a logical expression. The alpha expression is automatically converted to text before the comparison is performed. For example, if **T** is a text variable, the following syntax is valid:

```
if T = A
```

An alpha constant, such as "**x**", can appear in, and be compared with, an arithmetic expression. It can also be assigned to an integer or double variable, and can be used as an array subscript. For these cases, the alpha constant evaluates to its Latin1 character code which ranges from zero to 255.

The binary **+** operator concatenates text and/or alpha operands. For example:

```
define First_Name, Last_Name, Full_Name as text variables
define Middle_Initial as an alpha variable
...
Full_Name = First_Name + " " + Middle_Initial + ". " + Last_Name
```

1.08 Variables and Arrays

An **integer** variable named X is declared by the following statement:

```
define X as an integer variable
```

If the statement is specified in the preamble, the variable is global for that module; if specified within a routine, the variable is local to the routine. All variables are automatically initialized to zero, except **text** variables which are initialized to the zero-length string "".

A one-dimensional **double** array named Y is declared by:

```
define Y as a 1-dimensional double array
```

An array is dynamically allocated, and its number of elements determined at run time, by executing a **reserve** statement, e.g.,

```
reserve Y as 100
```

The number of elements in an array can be obtained by calling the built-in function `dim.f`; for example, `dim.f(Y)` returns 100. The first element of the array is stored at index 1. The elements of Y therefore are Y(1), Y(2), ..., Y(100). Each element is automatically initialized to zero. Multi-dimensional arrays may also be declared. The **release** statement de-allocates an array, i.e., frees its storage.

1.09 Expressions

Arithmetic expressions may use any combination of arithmetic operators: unary + and -; binary +, -, *, /, and ** (exponentiation). Built-in functions may be called to perform other arithmetic operations, including logarithms, modulus, square root, and trigonometric functions.

Logical expressions may use relational operators, =, <>, <, <=, >, >=, and logical operators **and** and **or**. Logical negation is specified by appending **is false** to a logical expression. The expression `J >= 1 and J <= dim.f(Y)` may be abbreviated as `1 <= J <= dim.f(Y)`. Logical expressions use “short-circuit” evaluation; that is, if the first operand of **and** evaluates to false, or the first operand of **or** evaluates to true, the second operand is not evaluated.

1.10 Basic Statements

Multiple statements may appear on one line, and one statement may span multiple lines. A semicolon is not required but is allowed after a statement.

The following statement assigns the value 10 to the variable named X:

```
X = 10
```

The optional **let** keyword can be also used:

```
let X = 10
```

The statement:

```
add 1 to X
```

is equivalent to

```
X = X + 1
```

Likewise, **X** may be decremented by **subtract 1 from X**.

The **read** statement reads free-form and formatted input. The **write** and **print** statements produce formatted output. The **open** and **close** statements open and close files.

The **if** statement specifies a logical expression followed by a sequence of statements to execute if the expression is true, and optionally by **else** and a sequence of statements to execute if the expression is false. It is terminated by the keyword **always**. For example:

```
define J as an integer variable
read J
if 1 <= J <= dim.f(Y)
  write Y(J) as "The value is ", d(7,2), /
else 'invalid entry
  write as "The index is out of bounds!", /
always
```

The **select** statement is a “case” statement in which one of several blocks of statements is chosen for execution based on the value of an expression.

1.11 Loops

A loop is specified by one or more control phrases followed by the body of the loop, which is either a single statement or a sequence of statements between the keywords **do** and **loop**. A **for** phrase causes the body of the loop to be executed once for each value assigned to a control variable, for example, for $J = 1$ to N . A **while** (or **until**) phrase specifies a logical expression and terminates the loop when the expression is false (or true). A **with** (or **unless**) phrase specifies a logical expression and executes the body of the loop for the current iteration when the expression is true (or false). These phrases may be combined to control loop execution. In addition, **leave** and **cycle** statements may be specified in the body of the loop: a **leave** statement terminates the loop, and a **cycle** statement terminates the current iteration of the loop.

A **find** or **compute** statement may be specified in the body of a loop. A **find** statement terminates the loop when the body is executed for the first time and is followed by an **if found** (or **if none**) phrase which evaluates to true if the body of the loop was (or was not) executed. For each execution of the body of the loop, a **compute** statement evaluates an arithmetic expression and computes statistics (e.g., sum, mean, maximum, minimum) from the values of the expression over the life of the loop.

1.12 Functions and Subroutines

A subroutine is a block of code which can be written once and invoked from different places in the program. In SIMSCRIPT, subroutines are recursive, which means the same subroutine can be invoked by itself. A *function* is a routine that returns a function result. A *subroutine* does not return a function result. Functions and subroutines may have one or more *given* arguments; however, only subroutines may have *yielded* arguments. The value of a given argument is an input to the routine, whereas the value of a yielded argument is an output from the routine.

Each function and subroutine is declared by a **define** statement in the preamble, which specifies the number and mode of arguments, and the mode of the function result for functions. To call a function with n given arguments, the function name is followed by a parenthesized list of n expressions, for example, **F(I, J, K)**. A subroutine is invoked by a **call** statement, for example

```
call Analyze given A, B yielding C, D
```

A function is terminated by a **return with** statement, which specifies the function result. A subroutine terminates when a **return** statement is executed or the end of the subroutine is reached.

The following function has three given arguments: a one-dimensional array of **text** values, a **text** key to look up in the array, and a **text** value describing the order of values in the array. The function searches for the key in the array. If it is found, the index of the array element containing the key is returned; otherwise, zero is returned to indicate that the key

was not found. If the third argument is "ascending", the function uses binary search; otherwise, the array is searched sequentially.

```
function Search (T, Key, Order)

  define First, Last, and Index
    as integer variables
  First = 1
  Last = dim.f(T)

  if Order = "ascending"

    'binary search
    Index = (First + Last) / 2
    while First <= Last and Key <> T(Index)
    do
      if Key < T(Index)
        Last = Index - 1
      else
        First = Index + 1
      always
      Index = (First + Last) / 2
    loop

    if First > Last
      Index = 0 'not found
    always

  else 'sequential search

    for Index = First to Last
    with Key = T(Index)
      find the first case
    if none
      Index = 0 'not found
    always

  always

  return with Index

end
```

The function must be declared in the preamble:

```
define Search as an integer function
  given a 1-dimensional text argument
  and 2 text arguments
```

The following is an example of a function call:

```
if Search (A, "Jim", "ascending") > 0
  write as "Found Jim in array A", /
  always
```

1.13 Argument Checking

The **define routine** statement specifies the number of given and yielded arguments of a routine. It is also possible to specify the mode and dimensionality of each argument.

In the following example, a double function named **F** is declared. Its first argument is integer, its second argument is double, and its third argument is integer.

```
define F as a double function
    given an integer argument,
        a double argument, and an integer argument
```

The following statement declares a subroutine named **Test** given a text value and a one-dimensional integer array and yielding two double values.

```
define Test as a routine
    given a text argument and a 1-dimensional integer argument
    yielding 2 double arguments
```

The compiler checks each routine call to verify that the caller's arguments are compatible with the routine's arguments. A caller's given value is converted to the mode of the routine's given argument, and a routine's yielded value is converted to the mode of the caller's yielded argument, if the argument modes differ and mode conversion is possible. For example, a double value passed to an integer argument is automatically converted to integer. If the argument modes differ but mode conversion is not permitted (for example, passing a text value to a double argument), the compiler issues an error message.

When the mode and dimensionality of a routine's arguments have been declared in a **define routine** statement, it is not necessary to define the mode and dimensionality of the arguments within the routine implementation. But, if they are defined within the routine implementation, their definitions must agree with the definitions in the **define routine** statement. For example:

```
function F(M, X, N)
    /~the following statements are optional because
        the argument modes have already been declared
        in a "define routine" statement ~/

    define M and N as integer variables
    define X as a double variable
```

In some cases, the mode of routine arguments is known by the compiler without a **define routine** statement, such as the mode of arguments to function attributes, monitoring functions, and **before/after** routines.

1.14 Reference Mode

In SIMSCRIPT III, a “reference mode” is implicitly defined for each process type and temporary entity type. The name of the mode is the name of the entity type followed by the keyword **reference**. A “reference variable” is a typed pointer variable that can hold the “reference value” or address of an entity.

For example, if **Ship** is a temporary entity type, the mode **Ship reference** is implicitly defined. The following statement defines **Tanker** to be a reference variable that can hold the reference value of a **Ship** entity:

```
define Tanker as a Ship reference variable
```

The following statement creates a **Ship** entity, initializes its attributes to zero, and assigns its reference value to **Tanker**:

```
create Tanker
```

This entity is destroyed by:

```
destroy Tanker
```

When a reference variable is used to access an attribute, the compiler verifies that the attribute is an attribute of the entity type. For example:

```
C = Capacity(Tanker) /~ if Ship does not have a Capacity,
                        an error is reported ~/
```

The compiler also validates set operations when reference variables are used. For example:

```
define Captain as a Shiphand reference variable
...
file Captain in Crew(Tanker)    /~ compiler error unless
                                every Shiphand belongs to a Crew and
                                every Ship owns a Crew ~/
```

A reference variable of one entity type cannot be assigned to a reference variable of another entity type. For backward compatibility with SIMSCRIPT II.5, a reference variable can be assigned to an integer or pointer variable, and an integer or pointer variable can be assigned to a reference variable.

A variable can be checked at runtime to determine if it contains a reference value of a particular reference mode. For example, if **P** is a pointer variable that refers to a **Ship** entity, the logical condition, **P is a Ship reference**, is true:

```
if P is a Ship reference
    /~ it is safe to access a Ship attribute using P ~/
    C = Capacity(P)
    /~ and it is safe to assign P to a Ship reference variable ~/
    Tanker = P
```

always

More than one **Ship** entity can be created and destroyed at a time:

```
define S1, S2, S3 as Ship reference variables
create S1, S2, S3  '' create three Ships
destroy S1, S2, S3  '' destroy three Ships
```

An array of reference values can be defined and initialized:

```
define Armada as a 1-dimensional Ship reference array
define J as an integer variable

reserve Armada as 1000
for J = 1 to 1000
    create Armada(J)
```

Attributes, global variables, local variables, and arguments can be reference variables. A function that returns a reference value has a reference mode. The background mode, set by a **normally** statement, can be a reference mode. Preamble declarations may specify a reference mode before the entity type is declared; for example, **Ship reference** may appear in statements that precede the declaration of the **Ship** entity type.

1.15 Input and Output

Input data can be entered to the SIMSCRIPT III model interactively using GUI form, standardin unit or can be read from Database or a file. Output from a model can be presented in the same way. I/O operations using GUI are described in SIMSCRIPT III Graphics Manual, Database operations and interfaces are explained in SDBC Manual. This section describes basic input/output statements for using file system and standard input/output units.

The **open** statement associates a SIMSCRIPT I/O unit with a file. Its general form is

```
open [ unit ] EXPRESSION1
[ for ] { input | output } < comma >
[ [ file ] name is TEXT1 |
  binary |
  recordsize is EXPRESSION2 |
  noerror |
  append |
] < comma >
```

EXPRESSION1 specifies the unit number. If **input** is specified, the **unit** may appear in **use for input** statements. If **output** is specified, the unit may appear in **use for output** statements.

TEXT1 specifies the name of the file associated with the unit. If the **name** phrase is omitted, the filename **SIMU_{nn}** is assumed, where **nn** is the unit number. For example, for unit 3, the default filename is **SIMU03**.

The default file type is an ASCII file containing variable-length records. If **binary** is specified, the file is treated as a binary file containing fixed-length records. The free-form **read**, formatted **read**, **print**, **write** and **list** statements are used with ASCII files. The **read as binary** and **write as binary** statements are used with binary files.

Expression2 specifies the size of records in bytes. If the **recordsize** phrase is omitted, the size of records is assumed to be 132. For binary files, this is the actual length of each record. For files with variable length records, this is the maximum length of a record. Note that the “newline” character is not counted as part of the record length. Examples are:

```
open unit 1 for input, recordsize is 132
open 7 for output, binary, name is "datafile"
```

Normally, if a file cannot be opened for some reason, such as the file does not exist or the filename is invalid, the program will be aborted with a runtime error. If **noerror** is specified, however, the program will not be aborted. Instead, a global variable, **ropenerr.v** for the current input unit, or **wopenerr.v** for the current output unit, will be assigned a non-zero value which may be tested by the program. For example:

```
open unit 12 for input,
    file name is INPUT.FILENAME, noerror
use unit 12 for input
if ropenerr.v <> 0
    write INPUT.FILENAME as "Unable to open ", T *, /
    close unit 12
always
```

Note: **Ropenerr.v** and **wopenerr.v** will be set after the **use unit** statement, not after the **open statement**.

If a **unit**, which has not been opened, appears in a **use** statement, the following statement will open it automatically:

```
open UNIT-NUMBER for input
```

The standard units — 5, 6 and 98 — are opened automatically by the system and may not appear in an **open** statement. The record size of each is 132. Unit 5 is **stdin**, the standard input unit. It is opened **for input** and is the current input unit when a program begins execution. Unit 6 is **stdout**, the standard output unit. It is opened **for output** and is the current output unit when a program begins execution. Unit 98 is **stderr**, the standard error unit. It is opened **for output** and is used for writing system error messages. Each of the standard units is associated with the terminal unless it has been redirected.

The units 1-4 and 7-97 have no predefined meaning and are available for general use. Unit 99 is the **buffer**. This unit may also appear in an **open** statement, but the **name** phrase is ignored and no physical file is associated with it. The **recordsize** phrase is also ignored. The record size for **the buffer** is obtained from the global variable, **buffer.v**, with a default value of 132.

The **close** statement dissociates a SIMSCRIPT I/O unit from a file. Its general form is:

```
close [ unit ] EXPRESSION1
```

where **EXPRESSION1** specifies the unit number.

If the current input unit is closed, unit 5 becomes the current input unit. If the current output unit is closed, unit 6 becomes the current output unit.

A unit, which is open when a program terminates, is closed automatically. All units, including unit 99, may be closed, except for the standard units, which must remain open at all times.

The global variable, **lines.v**, indicates whether pagination is enabled for the current output unit. By default, **lines.v = 0** which indicates that pagination is disabled. To enable pagination, initialize **lines.v** to a non-zero value indicating the desired number of lines per page. For example, to produce paginated output on unit 1, with 60 lines per page, specify:

```
use unit 1 for output
let lines.v = 60
```

A record read from a file containing variable-length records will automatically have blanks appended to it so that it is as long as the record size specified for the unit. Furthermore, each tab character found in the record will be expanded into one or more blanks, i.e. tab stops are set every 8 columns, starting with column 1. The global variable **rreclen.v** contains the length of the record last read from the current input unit before blanks are appended but after tabs have been expanded.

2 Object-Oriented Programming

SIMSCRIPT III object-oriented programs are organized around object types, i.e. classes. Each object type has two interrelated sets of properties. These respective properties are **fields** and **methods**. The state of an object instance at any instant is described by the values in a series of fields, similar to those of a record. Its behavior, or the actions it is capable of performing, are described in its methods which are executable routines with special characteristics associated with the object type.

This formal association of data and code provides an inherent encapsulation of the data, because the information in an object's fields can be changed *only* by a method which belongs to the object or by a method which it has inherited.

In other words, the fields of an object may not be directly modified except by the object itself. Other parts of a SIMSCRIPT III program external to a particular object can change the value of the object's fields only indirectly. For instance a **controller** object might want a vehicle object's speed to be zero. The **controller** object could **call vehicle stop method**. The vehicle object's **stop** method would then set the vehicle's **speed** field to zero.

Other parts of a SIMSCRIPT III program may request the value of an object's fields by invoking the appropriate object method.

Essentially, an object's fields are “read only” from the perspective of code not included in the object, but are “read and write” from within the object itself.

Experience with the object-oriented approach shows that it is even more effective at modularizing the interactions of a program than the structured programming techniques. The part of the program which invokes some behavior or action can always invoke an action in the same way over a wide range of object types, car aircraft, etc. Each object, however, may have a different behavior in response to this same method invocation.

Note that, in any program, there will likely be a number of different methods which have the same name. Since each is encapsulated within an object, this does not lead to any ambiguity. Each invocation of a method is accomplished by addressing the specific object instance, requesting it to execute one of its methods.

Thus, the calling entity need not have detailed knowledge of how an object will accomplish the action it is being told to do. It simply asks for a generic action to be performed and the object which receives the request has its own, tailored routine to perform that action.

New object types can be defined in terms of existing object types. When this is done, all of the fields and methods of the existing object are incorporated as a proper subset of the new object type. This capability is known as **inheritance**. Where an inherited method is inappropriate, it can be **overridden** and a more appropriate method substituted.

The appropriate use of inheritance encourages software reusability more effectively than any library of procedures. Unlike procedural libraries, a properly-structured object library allows selective redefinition of some of the code while incorporating other code unchanged.

The design of a SIMSCRIPT III object-oriented program, then, encourages a careful separation of the data representation and behaviors for each object type, as well as the declaration of related types using a common inheritance.

2.01 Classes and Objects

A class is defined by one or more **begin class** blocks appearing in a preamble. The following block defines a class named **Vehicle**:

```
begin class Vehicle
...
end
```

Definitions of attributes, methods, and sets are placed within these blocks.

A class also defines **reference mode** of the same name, so a reference variables of that mode can be declared, like:

```
define Car as a Vehicle reference variable
```

The following statement allocates a **Vehicle** type object, initializes its attributes to zero, and assigns its reference value to the reference variable named **Car**:

```
create Car
```

The following statement de-allocates the object whose reference value is stored in **Car**:

```
destroy Car
```

An array of objects can be created and destroyed:

```
define Fleet as a 1-dimensional Vehicle reference array

reserve Fleet as 50
for J = 1 to 50
    create Fleet(J)
...
for J = 1 to 50
    destroy Fleet(J)
release Fleet
```

2.02 Attributes

“Object attributes” are declared in **every** statement within **begin class** blocks. In the following example, every **Vehicle** object has an integer attribute named **ID**, a text attribute named **Manufacturer**, and double attributes named **Maximum_Speed** and **Current_Speed**:

```
begin class Vehicle

    every Vehicle
    has      an ID,
           a Manufacturer,
           a Maximum_Speed,
    and      a Current_Speed

    define ID as an integer variable
    define Manufacturer as a text variable
    define Maximum_Speed and Current_Speed as double variables

end
```

An object attribute is accessed like an attribute of a temporary entity, by placing a reference value expression in parentheses after the attribute name. For example:

```
ID(Car) = 781
Manufacturer(Car) = "Chrysler"
Maximum_Speed(Car) = 100
Current_Speed(Car) = Maximum_Speed(Car) / 2
```

It reads “ID of Car is 781”, “Manufacturer of Car is Chrysler”, etc.

“Class attributes” are declared in **the class** statements within **begin class** blocks. Whereas each object has its own copy of each object attribute, there is only one copy of each class attribute in the program. In our example, a class attribute named **Count** can be used to keep track of the current number of **Vehicle** objects in the program, and a class attribute named **Last_ID** can hold the ID of the last **Vehicle** created by the program.

```
begin class Vehicle

    the class has a Count and a Last_ID
    define Count and Last_ID as integer variables

end
```

A class attribute is accessed by specifying its qualified name, which is the class name followed by an apostrophe and the attribute name, with no intervening spaces. For example:

```
write Vehicle'Count as "The number of vehicles is ", i *, /
```

Object attributes and class attributes are automatically initialized to zero. Their names must be unique within the class.

The mode of an object attribute or class attribute must be specified by a **define variable** statement after the **has** phrase that names the attribute and within the same **begin class** block.

```
begin class Vehicle

    every Vehicle
    has an ID,
        a Manufacturer,
        a Maximum_Speed,
    and a Current_Speed

    define ID, Manufacturer, Maximum_Speed, and Current_Speed
        as integer variables
    define Manufacturer as a text variable

    the class has a Count and a Last_ID
        define Count and a Last_ID as integer variables
end
```

Statement **normally mode is** may appear within a **begin class** block to establish a background mode, and attributes defined by subsequent **has** phrases will have the background mode if their mode is not specified by a **define variable** statement. In the following example, all of the attributes have the background mode of integer except **Manufacturer**:

```
begin class Vehicle

    normally mode is integer

    every Vehicle
    has an ID,
        a Manufacturer,
        a Maximum_Speed,
    and a Current_Speed

    define Manufacturer as a text variable

    the class has a Count and a Last_ID

end
```

After the **begin class** block, the background mode reverts to its setting before the block. The background settings inside the block are independent of the background settings outside the block. Substitutions defined by **define to mean** and **substitute** statements within a **begin class** block have effect only within the block.

The dimensionality of an object attribute or class attribute is zero by default, which means the attribute contains a scalar value. However, a dimensionality greater than zero may be specified in a **define variable** statement or **normally dimension is** statement to define an array attribute. Let us add to our example an object attribute named **Tire_Pressure** that is an array of real values, where each element of the array contains the air pressure of one tire of the **Vehicle**.

```

begin class Vehicle

    every Vehicle has a Tire_Pressure

    define Tire_Pressure as a 1-dimensional real array

end

```

When accessing an element of an array attribute of an object, the array subscripts appear in parentheses *after* the parenthesized reference value expression. The following statements allocate and initialize the **Tire_Pressure** array for the **Vehicle** object whose reference value is stored in **Car**:

```

reserve Tire_Pressure(Car) as 4
for J = 1 to 4
    Tire_Pressure(Car)(J) = 30

```

Suppose that a **Vehicle** object is assumed to have four tires. A named constant may be defined within a **begin class** block and is called a “class constant”:

```

begin class Vehicle

    define Num_Tires = 4 as a constant

end

```

A class constant is accessed by specifying its qualified name:

```

reserve Tire_Pressure(Car) as Vehicle'Num_Tires
for J = 1 to Vehicle'Num_Tires
    Tire_Pressure(Car)(J) = 30

```

Statistical attributes may be defined by **accumulate** and **tally** statements appearing within a **begin class** block. A statistical attribute is an object attribute (or class attribute) whose value is computed based on the values assigned to another object attribute (or class attribute). We add to our example an object attribute named **Trip_Distance** and a statistical attribute named **Odometer** containing the sum of the values assigned to **Trip_Distance**.

```

begin class Vehicle

    every Vehicle has a Trip_Distance

    define Trip_Distance as a real variable

    tally Odometer as the sum of Trip_Distance

end

```

Object attributes and class attributes may be reference variables, random variables, and monitored variables.

2.03 Methods

A method is a routine associated with a class. It may have given arguments, and it may be a function which returns a function result, or a subroutine which does not return a function result but may have yielded arguments.

An “object method” is invoked on behalf of an object and performs some operation using the object. A “class method” is related to the class but is not invoked on behalf of an object.

Object methods are declared in **every** statements, and class methods are declared in **the class** statements, within **begin class** blocks. The mode and dimensionality of a method’s arguments, and the mode of the method’s function result if the method is a function, are specified by a **define method** statement after the method’s declaration and within the same **begin class** block. A **define method** statement is similar to a **define routine** statement. If the **define method** statement is omitted, the method is assumed to be a subroutine with no arguments.

The names of methods and attributes must be unique within the class; however, these names may be defined elsewhere in the program, including in other classes.

If an object method is a subroutine with no arguments, it may be specified in an **after creating** statement within a **begin class** block, which causes the method to be invoked implicitly on behalf of an object after a **create** statement has allocated the object and initialized its attributes to zero. Since this method cannot accept arguments, the program can define and explicitly call another object method that accepts arguments and uses them to initialize attributes of the new object to nonzero values.

If an object method is a subroutine with no arguments, it may be specified in a **before destroying** statement within a **begin class** block, which causes the method to be invoked implicitly on behalf of an object before a **destroy** statement has de-allocated the object.

In our **Vehicle** example, we define five object methods and one class method. The object method **Construct** is invoked automatically after a **Vehicle** is created, and the object method **Destruct** is invoked automatically before a **Vehicle** is destroyed. The object method **Initialize** is given three arguments which are used to initialize a **Vehicle** object. The object method **Flat_Tires** is a function that returns the number of under-inflated tires. The object method **Print** writes a description of a **Vehicle**, and the class method **Print_Count** writes the current number of **Vehicle** objects.

```

begin class Vehicle

    every Vehicle
    has      a Construct method,
            a Destruct method,
            an Initialize method,
            a Flat_Tires method,
            and a Print method

    after creating a Vehicle, call Construct
    before destroying a Vehicle, call Destruct

    define Initialize as a method given
        a text argument,    '' name of manufacturer
        a double argument,  '' maximum speed
    and      a real argument    '' initial tire pressure

    define Flat_Tires as an integer method given
        a real argument    '' minimum tire pressure

    the class has a Print_Count method

end

```

An object method is may be invoked with given and yielded arguments. A reference value expression is specified in parentheses after an object method name and before any given arguments. A class method name must be qualified. The following statements invoke the methods of the **Vehicle** class and the Chevy object methods:

```

define Chevy as a Vehicle reference variable

create Chevy '' implicit call Construct(Chevy)
call Initialize(Chevy) given "Chevrolet", 90, 32

if Flat_Tires(Chevy)(25) is zero
    write as "Tires are okay", /
always

call Print(Chevy)
call Vehicle'Print_Count

destroy Chevy '' implicit call Destruct(Chevy)

```

The reference value of an object is passed implicitly by value to an object method and must be nonzero. It is accessible within the object method in an implicitly-defined local reference variable that has the same name as the class. Because a class method is not invoked on behalf of an object, a reference value is not passed to a class method and this local reference variable is not defined within a class method.

A method implementation begins with the keyword **method**. The following is an implementation of the **Construct** object method:

```

method Vehicle'Construct

    add 1 to Count
    add 1 to Last_ID
    ID(Vehicle) = Last_ID
    Manufacturer(Vehicle) = "Unknown"
    reserve Tire_Pressure(Vehicle) as Num_Tires

end

```

As shown above, the names of class attributes, **Count** and **Last_ID**, and the name of the class constant, **Num_Tires**, do not need to be qualified within a method of the class. However, the method name, **Vehicle'Construct**, must be qualified unless it follows a **methods** heading that names the class. The object attributes, **ID**, **Manufacturer**, and **Tire_Pressure**, are subscripted by the implicitly-defined local reference variable named **Vehicle** that contains the reference value of the **Vehicle** object for which the method was invoked. However, these subscripts may be omitted and are implicit when accessing object attributes and calling object methods. With these changes, here is an equivalent implementation of the **Construct** method followed by implementations of the other **Vehicle** methods:

```

methods for the Vehicle class

method Construct  ' ' called after a Vehicle object has been created

    add 1 to Count
    add 1 to Last_ID
    ID = Last_ID
    Manufacturer = "Unknown"
    reserve Tire_Pressure as Num_Tires

end

method Initialize given Maker, Max_Speed, Initial_Pressure

    Manufacturer = Maker
    Maximum_Speed = Max_Speed

    define J as an integer variable
    for J = 1 to Num_Tires
        Tire_Pressure(J) = Initial_Pressure
    end

end

method Flat_Tires(Min_Pressure)

    define Count and J as integer variables

    for J = 1 to Num_Tires with Tire_Pressure(J) < Min_Pressure
        add 1 to Count  ' ' increment local variable
    end

    return with Count  ' ' return number of under-inflated tires

end

method Print

    print 3 lines with ID, Manufacturer, Current_Speed,
        Maximum_Speed, Odometer, Flat_Tires(10) thus
    Vehicle # *** manufactured by *****
    Its current and maximum speeds are *** and *** mph.
    Its odometer reads *****.* miles. It has * flat tires.

end

```

```

method Destruct  '' called before a Vehicle object is destroyed

    write as "Destroying:", /
    call Print
    release Tire_Pressure
    subtract 1 from Count

end

method Print_Count

    write Count as "There are ", i *, " Vehicle objects in existence.", /

end

```

A method that is a function may have left and/or right implementations. A left implementation begins with the keywords **left method**, whereas a right implementation begins with the keywords **method** or **right method**.

An object method (or class method) that is a function is implicitly defined for a monitored object attribute (or class attribute). This method has the same name and mode as the attribute, and is given n integer arguments where n is the dimensionality of the attribute. It has left and/or right implementations depending on whether the attribute is monitored on the left and/or the right.

A method may not be represented as a subprogram literal and called using a subprogram variable.

2.04 Grouping Objects in Sets

Objects as well as entities can be grouped in sets. A *set* is a doubly-linked list with a programmer-defined name. The *owner* of a set of objects named List has three *owner attributes*: reference variables F.List and L.List, which identify the first and last objects in the set, and N.List, which holds the number of objects in the set. A *member* of this set has three *member attributes*: reference variables P.List and S.List, which identify the predecessor and successor objects in the set, and M.List, which indicates whether this object is in a set named List.

An object may own and belong to any number of sets. Each **belongs** phrase in an **every** statement names a set in which an object may be a member. Each **owns** phrase in an **every** statement names a set owned by an object. An **owns** phrase in **the class** statement names a set owned by the class. The set named in an **owns** phrase is qualified by the name of the member class.

A **belongs** phrase in an **every** statement appearing *inside* a **begin class** block defines a set that contains objects of the class. Member attributes **p.set_name**, **s.set_name**, and **m.set_name** are implicitly defined as 0-dimensional (scalar) object attributes. A **define set** statement may appear inside the block after the **belongs** phrase to specify the ordering of members of the set, either FIFO (first-in first-out, which is the default), LIFO (last-in first-

out), or ranked based on the values of one or more 0-dimensional object attributes (and values returned by object methods that are functions with no arguments).

An **owns** phrase in an **every** statement (or **the class** statement) appearing *inside* a **begin class** block refers to a set of entities or set of objects owned by an object of the class (or owned by the class). Owner attributes **f.set_name**, **l.set_name**, and **n.set_name** are implicitly defined as object attributes (or class attributes) with the background dimensionality. If the background dimensionality is nonzero, the owner attributes are array attributes and the object (or class) owns an array of sets.

Unless the owner and member class are the same class, an **owns** phrase must refer to a set of objects by its qualified name, i.e., the name of the member class, followed by an apostrophe and the set name. However, only the set name appears in the name of owner attributes.

In the following example, the **owns** phrase indicates that every **Repair_Shop** object owns a set of **Vehicle** objects named **Service_Queue**. The set of objects is defined by the **belongs** phrase and **define set** statement.

```
begin class Repair_Shop
    every Repair_Shop owns a Vehicle'Service_Queue
end

begin class Vehicle
    every Vehicle belongs to a Service_Queue
    define Service_Queue as a FIFO set
end
```

The implicitly-defined member set attributes of a **Vehicle** object are **p.Service_Queue**, **s.Service_Queue**, and **m.Service_Queue**. The implicitly-defined owner set attributes of a **Repair_Shop** object are **f.Service_Queue**, **l.Service_Queue**, and **n.Service_Queue**. The mode of attributes **p.Service_Queue**, **s.Service_Queue**, **f.Service_Queue**, and **l.Service_Queue** is **Vehicle reference**.

A **file** statement inserts an object into a set. Variations of this statement permit the object to be inserted first or last in the set, or immediately before or after a specified object. If the position is unspecified, the object is placed into the set according to the “set discipline,” which may be FIFO, LIFO, or “ranked,” i.e., ordered according to attribute values of the members. The set discipline is declared by a **define** statement in the **begin class** block of the member class and is FIFO by default.

A **remove** statement removes an object from a set. Variations of this statement remove the first or last object, or a specific object from the set. A **for each** loop control phrase traverses a set in the forward or reverse direction, executing the body of the loop once for each member of the set. Special logical expressions test whether an object is in a set and whether a set is empty. For example:

The following statements illustrate operations involving the **Service_Queue** set:

```

define Car and MyCar as Vehicle reference variables
define EZ_Auto and Ferrari_Depot as Repair_Shop reference variables
create MyCar, EZ_Auto, and Ferrari_Depot
...

for each Car in Service_Queue(EZ_Auto) with Manufacturer(Car) = "Ferrari"
do
    remove Car from Service_Queue(EZ_Auto)
    file Car in Service_Queue(Ferrari_Depot)
    write as "Transferred:", /
    call Print(Car)
loop

if Service_Queue(EZ_Auto) is empty
    write as "Time for a coffee break", /
always

if MyCar is in Service_Queue
    write as "My car is in the shop", /
always

```

An object may belong to any number of sets. An object or class may own any number of sets and arrays of sets. A set contains either objects or entities but not a mixture of the two. An object method (or class method) can be invoked automatically **before/after filing/removing** an entity or object into a set owned by an object (or class).

A **belongs** phrase in an **every** statement appearing *outside* a **begin class** block defines a set of entities (temporary entities, permanent entities, and/or resources).

An **owns** phrase in an **every** statement (or **the system** statement) appearing *outside* a **begin class** block refers to a set of entities or set of objects owned by an entity (or owned by **the system**).

SIMSCRIPT III supports sets of objects and sets of entities. It also supports array of sets.

2.05 Arrays of Sets

An array of sets can be declared, as illustrated by the following example:

```

every Ship belongs to a Fleet

normally dimension is 1
the system owns the Fleet

```

The following statements reserve and release an array of sets Fleet:

```

reserve Fleet as 100
release Fleet

```

The number of elements in this array of sets is obtained by **dim.f(Fleet)**.

2.06 Inheritance

A new class similar to the existing classes defined in the model can be derived from one or more existing classes by inheriting their attributes and methods. This language property is named inheritance.

In single inheritance, a class is derived from one base class. In multiple inheritance, a class is derived from two or more base classes. SIMSCRIPT III supports both, single and multiple inheritance.

A derived class inherits the object attributes of each of its base classes. This means that an object of a derived class has a copy of each object attribute defined or inherited by its base classes. In addition, the derived class may define object attributes of its own.

In the following example, a class named **Gas_Vehicle** is derived from the **Vehicle** class, which is indicated by the **is a** phrase of the **every** statement. Each **Gas_Vehicle** object has the object attributes of a **Vehicle**, such as **ID**, **Manufacturer**, etc., and the object attributes defined here: **Miles_Per_Gallon**, **Fuel_Capacity**, and **Current_Gallons**.

```
begin class Gas_Vehicle
    every Gas_Vehicle is a Vehicle and
    has a Miles_Per_Gallon,
        a Fuel_Capacity,
    and a Current_Gallons
    define Miles_Per_Gallon, Fuel_Capacity, and Current_Gallons
        as real variables
end
```

A derived class also inherits the object methods of each of its base classes. This means that each object method defined or inherited by its base classes may be invoked on behalf of an object of the derived class. In addition, the derived class may define object methods of its own.

In our example, the object methods of the **Vehicle** class, such as **Initialize**, **Flat_Tires**, etc., may be invoked on behalf of a **Gas_Vehicle** object. This is appropriate because the **Gas_Vehicle** *is a* **Vehicle**: it has all of the object attributes of a **Vehicle** and can be operated upon by these methods as if it were a **Vehicle** object. The **Gas_Vehicle** class may define object methods of its own, for example, a **Fuel_Level** method that returns the value of **(Current_Gallons / Fuel_Capacity)**. Note that an object method defined by the **Gas_Vehicle** class may not be invoked on behalf of a **Vehicle** object because a **Vehicle** object lacks the object attributes defined by the **Gas_Vehicle** class. A **Vehicle** is *not* a **Gas_Vehicle**.

A derived class cannot alter the definition of an inherited object attribute or object method. For example, the **Gas_Vehicle** class cannot change the mode of the inherited **ID** attribute. A derived class may define an attribute or method having the same name as an inherited attribute or method, but it does not replace or change the inherited attribute or method. The result is that the derived class has *two* definitions of the name, one defined by the class and the other inherited from a base class.

In the following example, the **Gas_Vehicle** defines a text object attribute named **ID** and an object method named **Initialize** which accepts three more given arguments than the inherited **Initialize** method.

```
begin class Gas_Vehicle
    every Gas_Vehicle has an ID and an Initialize method
    define ID as a text variable
    define Initialize as a method given
        2 text arguments,  ' ' VIN and manufacturer name
        1 double argument, ' ' maximum speed
        and 3 real arguments 'initial tire pressure, mpg, and fuel capacity
end
```

When a name has been inherited from two or more base classes, or has been defined by the derived class and inherited from one or more base classes, each inherited definition must be accessed using its qualified name. A **Gas_Vehicle** object has an inherited integer attribute named **Vehicle'ID** and a defined text attribute named **ID** or **Gas_Vehicle'ID**.

The **Initialize** method defined by the **Gas_Vehicle** class is called on behalf of a **Gas_Vehicle** object. The following implementation of this method calls the inherited **Initialize** method on behalf of the **Gas_Vehicle** object to initialize its inherited attributes, **Manufacturer**, **Maximum_Speed**, and **Tire_Pressure**. It then initializes three of its defined attributes, **ID**, **Miles_Per_Gallon**, and **Fuel_Capacity**.

```
methods for the Gas_Vehicle class
method Initialize
    given VIN, Maker, Max_Speed, Initial_Pressure, MPG, Tank_Size
    call Vehicle'Initialize given Maker, Max_Speed, Initial_Pressure
    ID = VIN
    Miles_Per_Gallon = MPG
    Fuel_Capacity = Tank_Size
end
```

The inherited **after creating** and **before destroying** methods, **Construct** and **Destruct**, are invoked implicitly:

```
define Buick as a Gas_Vehicle reference variable
create Buick      ' ' invokes Vehicle'Construct
call Initialize(Buick)  ' ' invokes Gas_Vehicle'Initialize
    given "5A2TY461T", "Buick", 95, 35, 22.5, 15
call Print(Buick)  ' ' invokes Vehicle'Print
destroy Buick     ' ' invokes Vehicle'Destruct
```

A derived class can provide an object method implementation that “overrides” an inherited one. For example, the **Gas_Vehicle** class can override the inherited **Print** method:

```
begin class Gas_Vehicle
    every Gas_Vehicle overrides the Print
end
```

The new implementation calls the overridden implementation to print attributes inherited from the **Vehicle** class. It then prints attributes defined by the **Gas_Vehicle** class.

```
methods for the Gas_Vehicle class
method Print
    call Vehicle'Print '' invoke the overridden implementation
    print 2 lines with ID, Miles_Per_Gallon,
    Fuel_Capacity, Current_Gallons thus
    ***** gets *.* miles per gallon.
    Its *.*-gallon tank contains *.* gallons.
end
```

Because a **Gas_Vehicle** object can be treated as a **Vehicle** object, a **Gas_Vehicle** reference value can be assigned (or passed) to a **Vehicle** reference variable (or argument). However, a **Vehicle** reference value cannot be assigned (or passed) to a **Gas_Vehicle** reference variable (or argument). When the **Print** method is called using a **Vehicle** reference variable that contains a **Gas_Vehicle** reference value, **Gas_Vehicle'Print** is invoked. For example:

```

define V as a Vehicle reference variable
create V      '' create a Vehicle object
call Print(V) '' invoke Vehicle'Print
destroy V     '' destroy the Vehicle object

define GV as a Gas_Vehicle reference variable
create GV     '' create a Gas_Vehicle object
call Print(GV) '' invoke Gas_Vehicle'Print

V = GV       '' assign Gas_Vehicle reference value
              '' to Vehicle reference variable
call Print(V) '' invoke Gas_Vehicle'Print
destroy V    '' destroy the Gas_Vehicle object

create V     '' create a Vehicle object
GV = V      '' not allowed! this is flagged by the compiler

```

A variable can be checked at runtime to determine if it contains a reference value of an object belonging to a particular class. The following logical condition is true if the variable **P** refers to a **Vehicle** object or to an object of a class derived from **Vehicle** such as a **Gas_Vehicle** object.

```

if P is a Vehicle reference

```

A **Service_Queue** set may contain not only **Vehicle** objects but also objects of classes derived from **Vehicle**. A **Gas_Vehicle** object has inherited the ability to be a member of a **Service_Queue** set. It has inherited the member attributes, **p.Service_Queue**, **s.Service_Queue**, and **m.Service_Queue**, from the **Vehicle** class.

```

define Shop as a Repair_Shop reference variable
define V as a Vehicle reference variable
define GV as a Gas_Vehicle reference variable
create Shop, V, GV

file V in Service_Queue(Shop)
file GV in Service_Queue(Shop)

for each V in Service_Queue(Shop)
    call Print(V)

```

The body of the loop invokes **Vehicle'Print** or **Gas_Vehicle'Print** depending on whether reference variable **V** holds the reference value of a **Vehicle** or **Gas_Vehicle** object. This capability is called polymorphism and is one of the properties of Object-Oriented languages.

Suppose each vehicle in the service queue must be driven to another repair shop ten miles away:

```

for each V in Service_Queue(Shop)
    schedule a Trip(V) given 10, 30 in 0 days

```

If the **Gas_Vehicle** class overrides the **Trip** process method, then **Gas_Vehicle'Trip** is scheduled for each **Gas_Vehicle** object in the queue and **Vehicle'Trip** is scheduled for each **Vehicle** object.

A class derived from the **Repair_Shop** class inherits the ability to own a **Service_Queue** set. It inherits the owner attributes, **f.Service_Queue**, **l.Service_Queue**, and **n.Service_Queue**.

A derived class may specify **accumulate** and **tally** statements that compute statistics based on the values assigned to inherited object attributes. An inherited object method that is a function, including the method associated with a monitored object attribute, is overridden by naming it an **overrides** phrase and providing left and/or right implementations of the method.

The class attributes, class methods, and class constants of a base class may be accessed without qualification within a method of a derived class. A class method cannot be overridden. Substitutions defined by **define to mean** and **substitute** statements within a **begin class** block of a base class are not inherited.

“Cyclic” inheritance is not permitted, for example, **every A is a B** and **every B is an A**, or **every A is a B**, **every B is a C**, and **every C is an A**.

Suppose class **D** is derived from classes **B** and **C**, and that class **A** is a base class of both **B** and **C**. That is, **every D is a B and a C**, **every B is an A**, and **every C is an A**. This is known as “diamond-shaped” inheritance. There is only one occurrence of **A**'s object attributes in a **D** object. If both **B** and **C** override an object method **M** inherited from **A**, then **D** must override **M**; the implementation of **D'M** may invoke any combination of **A'M**, **B'M**, and **C'M**.

3 Object-Oriented Discrete Simulation

There are two general categories of computer simulation: *continuous simulation* and *discrete-event simulation*.

Continuous simulation describes events using sets of equations which are solved numerically with respect to time. Examples of problems in this area are fluid-flow or hydraulics problems and financial modeling. Typically a time step is chosen. The continuous simulation program then steps forward by the increment of time chosen for the time step and recalculates all equations which describe the model.

Discrete-event simulation describes a system in terms of logical relationships which cause changes of state at discrete points in time rather than continuously over time. Examples of problems in this area are most queuing situations: Objects (customers in a gas station, aircraft on a runway, jobs in a computer) arrive and change the state of the system instantaneously. Varying amounts of time elapse between events.

In discrete-event simulation, large or small amounts of simulation time can pass between events, but the state of the system is only of interest when one of its component parts changes state. SIMSCRIPT III takes the capabilities of discrete systems modeling languages like Simula and SIMSCRIPT II.5 and adds object-oriented programming capability and the modularity.

Process-Oriented vs Event-Oriented Simulation

The classical approach to discrete-event simulation is event-oriented. In this approach, routines are written to describe discrete events in the operation of a system. For instance, in a simple bank model the event routines might be:

- Customer arrives
- Customer enters queue
- Customer engages services of teller
- Customer leaves.

No time passes *during* any event routine. Instead, passage of time is handled by scheduling the next event for the object currently being manipulated. In the simple bank model, the event “Customer engages services of teller” would schedule the next event, “Customer leaves”, at some future time.

This event-oriented approach is adequate for smaller models, but in larger models it is often difficult to follow or modify the flow of logic which describes the behavior of an object, such as a customer. Consider the simple bank model if we added a janitor, a security guard and some management functions. There would be many unrelated event routines. The logic flow which describes the behavior of a customer would not be encapsulated in one process routine.

The process approach simplifies larger models by allowing many aspects of an object's behavior in a model (e.g. bank customers) to be described in one method which allows for the passage of time at one or more points in its code.

There is a further advantage to the process technique. Once the actions of a class of objects (such as customers in a bank) have been gathered together in an object, the simulation program can create multiple, concurrent *instances* of the object. In our bank, for example, the simulation program would generate a new instance of the customer object each time a customer arrived. It could also pass information about the customer in the parameter list of the object's initialization method. Perhaps it would pass in information about the sort of customer (*young* or *elderly*) and the expected service time for the customer. While there would be multiple, distinct copies of the customer object operating simultaneously, each could have different values of their fields to describe the particular customer's properties.

Finally, objects can interact. In our example, an instance of the customer object with the *young* attribute might yield its place in the queue to a customer object with the *elderly* attribute.

This process approach is supported in SIMSCRIPT III. It exploits object-oriented programming features to simplify both the original development and the subsequent maintenance of large models.

A simulation model written in SIMSCRIPT III defines a system in terms of objects and their time-elapsing activities i.e. processes-methods which can be scheduled at certain instance of simulation time. Each SIMSCRIPT III object is capable of carrying on multiple, concurrent activities each of which elapses simulation time. An **activity** is scheduled by an object instance using a **wait/work** statement in a **process method**. An **activity** is what occurs in the model as time elapses. Any or all activities of an object can be interrupted, if necessary. An **event** is a point in time at which the state of the model changes in some way. Any process method which does not contain wait statement is an event.

The process technique provides a powerful structure for expressing most categories of simulation problems, and provides significant advantages over the direct use of discrete events. The advantages of processes are both conceptual and labor-saving.

3.01 Process Method

Any method that is a subroutine may be declared as a “process method,” which can be invoked directly by a **call** statement or scheduled by a **schedule** statement for execution at some future simulation time. In our example, let us define a process method named **Trip** given the trip distance and average speed and yielding the duration of the trip.

```
begin class Vehicle

  every Vehicle has a Trip process method

  define Trip as a process method
    given      2 double arguments  '' trip distance in miles and
                                '' average speed in mph
    yielding  1 double argument   '' trip duration in hours

end

methods for the Vehicle class

process method Trip given Distance, Average_Speed yielding Duration

  define Start_Time as a double variable
  Start_Time = time.v

  Current_Speed = min.f(Average_Speed, Maximum_Speed)
  wait Distance / Current_Speed hours
  Current_Speed = 0

  Duration = (time.v - Start_Time) * hours.v

  Trip_Distance = Distance '' update Odometer

end
```

This process method can be called directly, for example:

```
call Trip(Chevy) given 600, 55 yielding Trip_Duration.
```

In this case, the caller waits for the trip to complete and receives the duration of the trip in the yielded argument.

However, a trip can be scheduled to begin now,

```
schedule a Trip(Chevy) given 600, 55 in 0 days
```

or to begin sometime in the future:

```
schedule a Trip(Chevy) given 600, 55 in 3 days.
```

The routine that executes the **schedule** statement does not wait for the trip to complete and continues on without delay to the next statement of the routine. Upon completion of

the trip, argument values yielded by the process method are discarded. In this example, there is no one waiting to receive the duration of the trip; however, this information could be saved by the process method in an attribute.

If the process method is an object method, then an explicit or implicit reference value subscript must follow the method name. If the process method is a class method, however, the method is scheduled without a reference value expression.

A **schedule a** statement creates an instance of the process method:

```
schedule a Trip(Chevy) called Midwest_Trip given 600, 55 in 3 days.
```

The given arguments, and the reference value of the object, are saved in attributes of the process notice for this process method instance. The **time.a** attribute of the notice is assigned the simulation time at which the process method is to begin execution.

The process notice is filed into the event set **ev.s**, where it co-exists with other process notices. The event set is an array of sets and each process method type is assigned a unique index into the array.

The scheduled execution of a process method can be canceled and rescheduled by **cancel** and **schedule the** statements that refer to the process method instance. The reference value of the process notice may be stored in the implicitly-defined attribute,

```
cancel the Trip(Chevy)
schedule the Trip(Chevy) in 7 days
```

or stored in a **pointer** variable:

```
cancel the Midwest_Trip
schedule the Midwest_Trip in 7 days.
```

A process method in a wait state can be interrupted and later resumed:

```
interrupt the Trip(Chevy)
...
resume the Trip(Chevy)
```

or

```
interrupt the Midwest_Trip
...
resume the Midwest_Trip.
```

A process method can check the value of global variable **process.v** to determine if a simulation is running. If **process.v** is nonzero, then a simulation is running and **process.v** contains the reference value of the current process notice, and the process method is permitted to suspend execution using a **wait**, **suspend**, or **request** statement. However, if **process.v** is zero, then no simulation is running and it is a runtime error to suspend execution. Note that resources are requested and owned by the current process notice.

A process method can call or schedule itself or other process methods. A process method that is an object method is invoked on behalf of an object and can be thought of as an activity of the object. The event set can contain more than one scheduled invocation of the same or different process methods on behalf of a single object to model concurrent activities of the object.

A method can be invoked automatically **before/after scheduling/canceling** a process method. All process methods are scheduled internally (endogenously); however, an externally-scheduled process routine can call a process method to achieve the effect of exogenous scheduling.

A **priority** statement inside a **begin class** block specifies the priority order of the process methods of the class. A **priority** statement outside a **begin class** block may specify the priority order of process methods in different classes, and the priority order of processes. A **break ties** statement may not be specified for a process method.

3.02 Random Number Generation

SIMSCRIPT III utilizes a linear congruential generator (LCG) to produce uniform pseudo-random 31-bit values ranging from zero to 2,147,483,647. A predefined array named `seed.v` contains ten seed values equally spaced throughout the period of the LCG; however, any seed values may be assigned by the program to this array. A “stream” number between 1 and 10 selects a seed value from this array.

The values from the LCG are transformed by built-in functions into pseudo-random numbers from the following probability distributions: beta, binomial, Erlang, exponential, gamma, lognormal, normal, Poisson, triangular, uniform (continuous and discrete), and Weibull.

3.03 Statistics

An **accumulate** or **tally** statement specifies one or more statistics to compute automatically from the values assigned to an object attribute (or class attribute). A name is given to each statistic, and an object method (or class method) by that name is generated that returns the value of the statistic. Any of the following statistics may be computed: the maximum, minimum, number, sum, mean, mean square, sum of squares, variance, and standard deviation of the values assigned to the attribute. A histogram of the values may also be computed.

The statistics are weighted by simulation time if specified by an **accumulate** statement and are unweighted if the **tally** statement is used. The statistics can be computed for the

entire simulation, or for particular time intervals, for example, every day or every week of simulation time. The **reset** statement is used to initialize the statistics at the beginning of a time interval.

Suppose in our example we wish to measure how well a repair shop is doing its job, and assume that after each vehicle is serviced, the time required to service the vehicle is assigned to an object attribute named `Service_Time`. A **tally** statement specifies that the average and maximum service time is to be computed from the values assigned to this attribute. An **accumulate** statement indicates that the time-weighted average of the length of the service queue is to be computed. The number of vehicles in the queue is maintained in the implicitly-defined object attribute named `N.Service_Queue`, which is automatically updated whenever a vehicle is inserted into the queue by a **file** statement or removed from the queue by a **remove** statement. A **Print_Statistics** method displays the results.

```
begin class Repair_Shop

    every Repair_Shop
        has a Service_Time and
           a Print_Statistics method, and
        owns a Vehicle'Service_Queue

    define Service_Time as a double variable

    tally Avg_Service_Time as the mean and
         Max_Service_Time as the maximum
         of Service_Time

    accumulate Avg_Queue_Length as the mean
         of N.Service_Queue

end

methods for the Repair_Shop class

method Print_Statistics
    print 3 lines with
        Avg_Service_Time, Max_Service_Time, and
        Avg_Queue_Length as follows
    Average service time is **.**
    Maximum service time is **.**
    Average queue length is **.**
end
```

4 Modularity

It is easier to develop and maintain a large program that has been divided into meaningful units. A SIMSCRIPT III program consists of a main module and zero or more subordinate modules called “subsystems.” Subsystems promote better source code organization and facilitate the reuse of the code.

Main module consists of a preamble followed by one or more routines, including a main routine. The preamble declarations are visible only to the routines of the main module. A SIMSCRIPT II.5 program can be viewed as a SIMSCRIPT III main module.

4.01 Subsystems

Subsystem is a named module consisting of a public preamble followed by an optional private preamble and zero or more routines. The declarations in the public preamble are visible to the private preamble and routines of the subsystem, and to every module that “imports” this subsystem. The declarations in the private preamble are visible only to the routines of the subsystem.

The public preamble of a subsystem defines the interface to the subsystem, and the implementation is hidden in the private preamble and routines of the subsystem. A module may import any number of subsystems, and a subsystem may be imported by any number of modules.

A subsystem may be distributed as a source file containing only the public preamble, and one or more binary object files obtained by compiling the subsystem. The source file documents the subsystem interface and is read by the compiler when compiling a module that imports the subsystem. An executable program is built by linking the binary object files that were produced by compiling the main module and each of its subsystems.

Separate compilation is supported. If a subsystem’s private preamble or routines are modified, only the subsystem needs to be recompiled. However, each program that uses the subsystem must be re-linked.

A module imports a subsystem by specifying its name in an **importing** phrase appended to a preamble heading.

Not only can a main module import a subsystem, but a subsystem **A** can import a subsystem **B**. If the public preamble of subsystem **A** imports subsystem **B**, then a module that imports subsystem **A** will automatically import subsystem **B**.

```

public preamble for the X system
    importing subsystem A
end

public preamble for the A subsystem
    importing subsystem B
end

```

However, if the private preamble of subsystem **A** imports subsystem **B**, then a module that imports subsystem **A** is unaware of subsystem **B**.

```

public preamble for the X system
    importing subsystem A
end

public preamble for the A subsystem
end

private preamble for the A subsystem
    importing subsystem B

end

```

If the name of an imported definition is the same as a name defined by the importing module, or if the same name is imported from two or more subsystems, then the name of an imported definition must be qualified by pre-pending the name of the defining subsystem followed by a colon, with no intervening spaces. For example, if module **M** imports subsystems **S1** and **S2**, and the name **C** is defined in module **M** and in the public preambles of **S1** and **S2**, then the three definitions may be accessed within module **M** by using the qualified names, **M:C**, **S1:C**, and **S2:C**. The local definition may be accessed without qualification, that is, **C** and **M:C** are synonymous. Suppose **S1:C** is a class that has a class attribute named **A**. This attribute may be accessed within module **M** by using the qualified name, **S1:C'A**. If such a name is unwieldy, a substitution can be defined for it, for example:

```
define CA to mean S1:C'A
```

The method implementations of a class must appear within the module that defines the class. A “private” class is defined by one or more **begin class** blocks within the preamble of a main module or within the private preamble of a subsystem. A private class is visible only to the defining module.

A “public” class is defined by one or more **begin class** blocks within the public preamble of a subsystem and by zero or more **begin class** blocks within the private preamble of the subsystem. The public part of a public class is specified in the public preamble, whereas the private part of a public class is hidden in the private preamble. This makes it possible for a class to have a public interface yet also have private attributes, methods, and sets, and even private base classes.

Substitutions defined by **define to mean** and **substitute** statements, and the settings established by **normally** and **suppress/resume** statements, in effect at the end of the public preamble of a subsystem, are in effect at the beginning of the private preamble of the subsystem, and those in effect at the end of the private preamble apply to the routines of the subsystem. A module that imports the subsystem, however, does not import, nor is affected by, the substitutions and settings defined by the subsystem. Although it is not possible to import substitutions, named constants defined in the public preamble of the subsystem are imported.

In subsystems, each public routine, whether function or subroutine, must be defined in a public preamble, and each private function and subroutine must be defined in a private preamble. Full definition is encouraged, including specification of the mode and dimensionality of its arguments.

“System” attributes are defined by **the system** statements in the preamble of a main module. “Subsystem” attributes are analogously defined by **the subsystem** statements appearing in the public and private preambles of a subsystem.

A subsystem may provide a special **initialize** routine which is called once automatically before the main routine is executed. This routine can be used to initialize subsystem attributes, global variables, and class attributes defined by the subsystem. If more than one subsystem in a program has an **initialize** routine, the sequence in which these routines are executed is undefined.

The following example shows a subsystem and a main module that imports the subsystem.

```
public preamble for the Transportation subsystem

begin class Vehicle  '' public part of public class
    the class has a Count  '' public class attribute
    ...
end

'' public subroutine
define Check as a subroutine given a double argument

'' public subsystem attributes
the subsystem has an X and a Y
define X and Y as double variables
end

private preamble for the Transportation subsystem

begin class Moving_Object  '' private class
    ...
end

begin class Vehicle          '' private part of public class
    every Vehicle is a Moving_Object  '' private base class
    the class has a Last_ID          '' private class attribute
    ...
end
```

```

        ' private subsystem attribute
        the subsystem has a Z
        define Z as a double variable
    end

    methods for the Moving_Object class
    ...

    methods for the Vehicle class
    ...

    subroutine Check(Arg)
        ...
    end

    initialize ' called before main

        X = 1.0; Y = 1.0; Z = 1.0; Vehicle'Last_ID = 100;

    end

' main module

preamble for the City system
    importing the Transportation subsystem

    begin class City_Vehicle
        every City_Vehicle is a Vehicle
        ...
    end

    the system has a Y
    define Y as a text variable

end

/~
by importing the Transportation subsystem, routines of this module
can:
    create Vehicle objects
    access the public attributes of Vehicle such as Vehicle'Count
    call the public methods of Vehicle
    call the public subroutine Check
    access the public subsystem attributes X and Transportation:Y
    (qualification of Y is required
    to distinguish it from the system attribute
    named Y defined by this module)
but cannot:
    refer to class Moving_Object
    access the private attributes of Vehicle such as Vehicle'Last_ID
    call the private methods of Vehicle
    access the private subsystem attribute Z
~/

methods for the City_Vehicle class
...

main
...
end

```

4.02 Source Code Organization

A SIMSCRIPT III program consists of a main module or a main module and several subordinate modules called “subsystems.” The keywords **subsystem**, **module**, and **package** are synonymous.

A main module may have an optional preamble followed by one or more routines and **methods** headings. One of the routines must be named **main**. Preamble contains definitions of data structures used in the program like: classes, entities, global variables, constants and sets. All statements in a preamble are non-executable. A main module can be given a name and can import subsystems.

```
Preamble for the X system
    importing the A subsystem

end

methods for the X system
    main
end

Rout1
end
```

A subsystem begins with a public preamble and is followed by an optional private preamble and zero or more routines and **methods** headings.

SIMSCRIPT III compiler supports source code organization in which main module and subsystem may span multiple files; however, a preamble or routine may not span multiple files.

Building SIMSCRIPT III projects is facilitated by Interactive Development Environment (IDE) Simstudio, fully described in SIMSCRIPT III User Manual.

For the SIMSCRIPT III Simstudio Release 1, the following conventions are suggested: The public preamble of the main module should be placed in a file *name.sim*, where *name* is name of the main module. For example main module of system **shipping** should be placed in **shipping.sim** file. The implementation of the main module i.e. methods and routines should be placed in a file **shipping_i.sim**

Subsystem module imported by main module should be placed also in two files: *subsysname.sim* and *subsysname_i.sim*. The file *subsysname.sim* should contain public preamble of the subsystem and *subsysname_i.sim* should contain implementation of the subsystem starting with the private preamble and followed by the methods of the subsystem. For example subsystem **resource** would be placed in **resource.sim** and **resource_i.sim** files.

SIMSCRIPT III projects can also be built using command-line interface described in SIMSCRIPT III User Manual.

5 Library.m

Library.m is a special module that is implicitly imported by every preamble. This module defines routines, variables, and constants which are accessible to every module. These definitions may be accessed without qualification (for example, **time.v**) or with qualification (for example, **library.m:time.v**). The **library.m** definitions are described in the sections of this chapter:

- *5.01 Mode Conversion*
- *5.02 Numeric Operations*
- *5.03 Text Operations*
- *5.04 Input/Output*
- *5.05 Random-Number Generation*
- *5.06 Simulation*
- *5.07 Miscellaneous*

5.01 Mode Conversion

atot.f (*alpha_arg*)

A text function that returns a text value of length one containing *alpha_arg* as its only character. For example, **atot.f("B")** converts an alpha "B" to a text "B".

int.f (*double_arg*)

An integer function that returns the value obtained by rounding *double_arg* to the nearest integer. If the argument is positive, the rounded value is computed by adding 0.5 to the argument and truncating the result. If the argument is negative, the value is obtained by subtracting 0.5 from the argument and truncating. For example, **int.f(3.5)** returns 4 and **int.f(-3.5)** returns -4.

itoa.f (*integer_arg*)

An alpha function that returns the character representation of *integer_arg*. The argument must be in the range 0 to 9. The return value is in the range "0" to "9".

itot.f (*integer_arg*)

A text function that returns the text representation of *integer_arg*. For example, **itot.f(100)** returns "100" and **itot.f(-5)** returns "-5".

real.f (*integer_arg*)

A double function that returns the floating-point representation of *integer_arg*. For example, **real.f(3)** returns 3.0.

trunc.f (*double_arg*)

An integer function that returns the value obtained by truncating *double_arg* to remove its fractional part. For example, **trunc.f(3.5)** returns 3 and **trunc.f(-3.5)** returns -3.

ttoa.f (*text_arg*)

An alpha function that returns the first character of ***text_arg*** or returns a blank if ***text_arg*** is the null string. For example, **ttoa.f("yes")** returns "y" and **ttoa.f("")** returns " ".

5.02 Numeric Operations

abs.f (*numeric_arg*)

A function that returns the absolute value of an integer or double argument. If the argument is integer, the function returns an integer result. If the argument is double, the function returns a double result. For example, **abs.f(-5)** returns 5 and **abs.f(12.3)** returns 12.3.

and.f (*integer_arg1*, *integer_arg2*)

An integer function that returns the value obtained by performing a bitwise AND of *integer_arg1* and *integer_arg2*. For example, **and.f(23, 51)** returns 19 because the bitwise AND of binary 010111 (23) and binary 110011 (51) is binary 010011 (19).

arccos.f (*double_arg*)

A double function that returns the arc cosine of *double_arg* in radians. The argument must be in the range -1 to $+1$. The return value is in the range zero to π .

arcsin.f (*double_arg*)

A double function that returns the arc sine of *double_arg* in radians. The argument must be in the range -1 to $+1$. The return value is in the range $-\frac{\pi}{2}$ to $+\frac{\pi}{2}$.

arctan.f (*double_argY*, *double_argX*)

A double function that returns the arc tangent of (*double_argY* / *double_argX*) in radians. Either argument may be zero but not both. If *double_argY* is positive, the return value is in the range zero to π . If *double_argY* is negative, the return value is in the range $-\pi$ to zero. If *double_argY* is zero and *double_argX* is positive, the return value is zero. If *double_argY* is zero and *double_argX* is negative, the return value is π .

cos.f (*double_arg*)

A double function that returns the cosine of *double_arg*. The argument is specified in radians. The return value is in the range -1 to $+1$.

dim.f (array_arg)

An integer function that returns the number of elements in **array_arg**. The argument is normally an array pointer. However, if the argument names an array of sets, then the **f.set** array pointer is implicitly passed in its place. If the argument is zero, then zero is returned.

div.f (integer_arg1, integer_arg2)

An integer function that returns the truncated result of (**integer_arg1 / integer_arg2**). **integer_arg2** must be nonzero. For example, **div.f(17, 5)** returns 3 and **div.f(-12, 8)** returns -1.

exp.c

A double constant equal to the value of e , 2.718281828459045.

exp.f (double_arg)

A double function that returns the value of e^x where **double_arg** is the exponent.

frac.f (double_arg)

A double function that returns the fractional part of **double_arg**. It is computed by subtracting the truncated value of the argument from the original value. If the argument is positive, the return value is positive. If the argument is negative, the return value is negative. For example, **frac.f(3.45)** returns 0.45 and **frac.f(-3.45)** returns -0.45.

inf.c

An integer constant equal to the largest integer value. On 32-bit computers, this value is $2^{31} - 1 = 2,147,483,647$. The smallest integer value is **-inf.c-1**.

log.e.f (double_arg)

A double function that returns the natural logarithm (i.e., the base e logarithm) of **double_arg**. The argument must be positive.

log.10.f (*double_arg*)

A double function that returns the base 10 logarithm of ***double_arg***. The argument must be positive.

max.f (*numeric_arg1*, *numeric_arg2*, ...)

A function that returns the maximum value of two or more integer or double arguments. If every argument is integer, the function returns an integer result; otherwise, the function returns a double result.

min.f (*numeric_arg1*, *numeric_arg2*, ...)

A function that returns the minimum value of two or more integer or double arguments. If every argument is integer, the function returns an integer result; otherwise, the function returns a double result.

mod.f (*numeric_arg1*, *numeric_arg2*)

A function that computes ***numeric_arg1*** divided by ***numeric_arg2*** and returns the remainder. If both arguments are integer, the function returns an integer result; otherwise, the function returns a double result. ***Numeric_arg2*** must be nonzero. If ***numeric_arg1*** is positive, the return value is positive. If ***numeric_arg1*** is negative, the return value is negative. For example, **mod.f(14.5, 3)** returns 2.5 and **mod.f(-14.5, 3)** returns -2.5.

or.f (*integer_arg1*, *integer_arg2*)

An integer function that returns the value obtained by performing a bitwise inclusive OR of ***integer_arg1*** and ***integer_arg2***. For example, **or.f(23, 51)** returns 55 because the bitwise inclusive OR of binary 010111 (23) and binary 110011 (51) is binary 110111 (55).

pi.c

A double constant equal to the value of π , 3.141592653589793.

radian.c

A double constant equal to the number of degrees per radian, which is $\frac{180}{\pi}$ or 57.29577951308232.

rinf.c

A double constant equal to the largest real value. On 32-bit computers, this value is approximately 3.4×10^{38} ; however, a double value may be as large as 10^{308} . The smallest real value is **-rinf.c**.

shl.f (integer_arg1, integer_arg2)

An integer function that returns the value of *integer_arg1* shifted left by *integer_arg2* bit positions. For example, **shl.f(23, 2)** returns 92 because binary 00010111 (23) shifted left two positions is binary 01011100 (92). The value of *integer_arg1* is returned if *integer_arg2* is zero. The result is undefined if *integer_arg2* is negative.

shr.f (integer_arg1, integer_arg2)

An integer function that returns the value of *integer_arg1* shifted right by *integer_arg2* bit positions. For example, **shr.f(23, 2)** returns 5 because binary 010111 (23) shifted right two positions is binary 000101 (5). An arithmetic shift is performed with the sign bit copied to the most significant bit positions. The value of *integer_arg1* is returned if *integer_arg2* is zero. The result is undefined if *integer_arg2* is negative.

sign.f (double_arg)

An integer function that returns the sign of *double_arg*: +1 if the argument is positive, -1 if the argument is negative, and zero if the argument is zero.

sin.f (double_arg)

A double function that returns the sine of *double_arg*. The argument is specified in radians. The return value is in the range -1 to +1.

sqrt.f (*double_arg*)

A double function that returns the square root of ***double_arg***. The argument must be nonnegative.

tan.f (*double_arg*)

A double function that returns the tangent of ***double_arg***. The argument is specified in radians.

xor.f (*integer_arg1*, *integer_arg2*)

An integer function that returns the value obtained by performing a bitwise exclusive OR of ***integer_arg1*** and ***integer_arg2***. For example, **xor.f(23, 51)** returns 36 because the bitwise exclusive OR of binary 010111 (23) and binary 110011 (51) is binary 100100 (36).

5.03 Text Operations

concat.f (*text_arg1*, *text_arg2*, ...)

A text function that returns the concatenation of two or more text arguments. For example, **concat.f("Phi", "ladelp", "hia")** returns **"Philadelphia"**.

fixed.f (*text_arg*, *integer_arg*)

A text function that returns the value obtained after appending space characters to, or removing trailing characters from, the value of ***text_arg*** to make its length equal the value of ***integer_arg***. For example, **fixed.f("abcd", 2)** returns **"ab"** and **fixed.f("abcd", 5)** returns **"abcd "**. ***Integer_arg*** must be nonnegative; if it is zero, a null string is returned.

length.f (*text_arg*)

An integer function that returns the number of characters in ***text_arg***. For example, **length.f("Chicago")** returns 7 and **length.f("")** returns zero.

lower.f (*text_arg*)

A text function that returns the value of ***text_arg*** with each uppercase letter converted to lowercase. All other characters are unchanged. For example, **lower.f("Chicago")** returns **"chicago"** and **lower.f("CAFÉ")** returns **"café"**.

match.f (*text_arg1*, *text_arg2*, *integer_arg*)

An integer function that returns the position of the first occurrence of ***text_arg2*** in ***text_arg1*** excluding the first ***integer_arg*** characters of ***text_arg1***, or returns zero if there is no such occurrence. Zero is returned if ***text_arg1*** or ***text_arg2*** is the null string. ***Integer_arg*** must be nonnegative. For example, **match.f("Philadelphia", "hi", 2)** returns 10 and **match.f("Chicago", "hi", 2)** returns zero.

repeat.f (*text_arg*, *integer_arg*)

A text function that returns the concatenation of ***integer_arg*** copies of ***text_arg***. For example, **repeat.f("AB", 3)** returns **"ABABAB"**. ***Integer_arg*** must be nonnegative. A null string is returned if ***text_arg*** is a null string or ***integer_arg*** is zero.

substr.f (*text_arg*, *integer_arg1*, *integer_arg2*)

A text function that returns a substring of ***text_arg*** when called as a right function, or modifies a substring of ***text_arg*** when called as a left function. The substring begins with the character at position ***integer_arg1*** and continues until the substring is ***integer_arg2*** characters long or until the end of ***text_arg*** is reached. (The first character of ***text_arg*** is at position 1.) For example, the statement,

```
T = substr.f("Philadelphia", 6, 5)
```

assigns "delph" to **T**. When called as a left function, the text value assigned to the function replaces the specified substring of ***text_arg***, which must be an unmonitored text variable. The following assignment changes the value of **T** from "delph" to "delta":

```
substr.f(T, 4, 2) = "ta"
```

If the value assigned to the substring is not the same length as the substring, then space characters are appended to, or trailing characters are removed from, the assigned value. ***integer_arg1*** must be positive and ***integer_arg2*** must be nonnegative. If ***integer_arg1*** is greater than the length of ***text_arg***, or ***integer_arg2*** is zero, then a null string is returned when **substr.f** is called as a right function, and no modification is made to ***text_arg*** when **substr.f** is called as a left function.

trim.f (*text_arg*, *integer_arg*)

A text function that returns the value obtained by removing leading and/or trailing blanks, if any, from the value of ***text_arg***. If ***integer_arg*** is zero, leading *and* trailing blanks are removed; if ***integer_arg*** is negative, only leading blanks are removed; and if ***integer_arg*** is positive, only trailing blanks are removed. If ***text_arg*** is the null string or contains all blanks, then a null string is returned. For example, **trim.f(" Hello ", 0)** returns "Hello".

upper.f (*text_arg*)

A text function that returns the value of ***text_arg*** with each lowercase letter converted to uppercase. All other characters are unchanged. For example, **upper.f("Chicago")** returns "CHICAGO" and **upper.f("café")** returns "CAFÉ".

5.04 Input/Output

buffer.v

An integer variable that specifies the length of “the buffer” when the first **use the buffer** statement is executed. Its default value is 132.

efield.f

An integer function that returns the ending column number of the next value to be read by a free-form **read** statement using the current input unit, or returns zero if there are no more input values.

eof.v

An integer variable that specifies the action to take when an attempt is made to read data from the current input unit beyond the end of file. If the value of the variable is zero (which is the default), the program is terminated with a runtime error. However, if the value of the variable is nonzero (typically the program sets it to 1), the variable is assigned a value of 2 to indicate that end-of-file has been reached. Each input unit has its own copy of this variable.

heading.v

A subprogram variable that specifies a routine to be called for each new page written to the current output unit when pagination is enabled (**lines.v** is greater than zero), or contains zero (which is the default) if no routine is to be called. The routine typically writes a page heading but may perform other tasks. Each output unit has its own copy of this variable.

line.v

An integer variable that contains the number of the current line for the current output unit. It is initialized to 1. If pagination is enabled (**lines.v** is greater than zero), then the first line of each page is number 1. Each output unit has its own copy of this variable.

lines.v

An integer variable that enables pagination for the current output unit if containing a positive value indicating the maximum number of lines per page, or disables pagination if zero (which is the default) or negative. Each output unit has its own copy of this variable.

mark.v

An alpha variable that specifies the character that marks the end of input data describing an external process or random variable. Its default value is "*" (asterisk).

out.f (*integer_arg*)

An alpha function that returns (when called as a right function), or modifies (when called as a left function), the specified character of the current output line. *Integer_arg* is the column number of the character, which must be between 1 and the record size. For example, the statement, **A = out.f(4)**, assigns the character in column four to the variable **A**. The statement, **out.f(4) = "s"**, changes the character in column four to "s". This function may not be used if the current output unit has been opened for writing binary data.

page.v

An integer variable that contains the number of the current page for the current output unit. It is initialized to 1 and is incremented for each new page when pagination is enabled (**lines.v** is greater than zero). Each output unit has its own copy of this variable.

pagecol.v

An integer variable that specifies for the current output unit, a positive starting column number at which the word "Page," followed by the current page number, will be written as the first line of each page (preceding lines written by a **heading.v** routine) when pagination is enabled (**lines.v** is greater than zero); or the variable is zero (which is the default) or negative to disable this feature. Each output unit has its own copy of this variable.

rcolumn.v

An integer variable that contains the column number of the last character read from the current input line, or zero if no character has been read. Each input unit has its own copy of this variable.

read.v

An integer variable that contains the unit number of the current input unit. Its initial value is 5 because unit 5 (standard input) is the current input unit when a program begins execution. The assignment, **read.v = N**, changes the current input unit and has the same effect as the statement, **use N for input**.

record.v (integer_arg)

An integer function that returns the number of lines read from, or written to, the specified I/O unit. *Integer_arg* must be a valid unit number.

ropenerr.v

An integer variable that equals 1 to indicate that an error occurred when opening the file associated with the current input unit, or equals zero if no error occurred. If the **Open** statement for the unit specifies the **noerror** keyword, then the program can check the value of this variable after a **use** statement to determine whether an error occurred when opening the file; otherwise, such an error causes the program to terminate. Each input unit has its own copy of this variable.

rreclen.v

An integer variable that contains the number of characters read in the current input line, excluding the end-of-line character. Each input unit has its own copy of this variable.

rrecord.v

An integer variable that contains the number of lines read from the current input unit. Each input unit has its own copy of this variable.

sfield.f

An integer function that returns the starting column number of the next value to be read by a free-form **read** statement using the current input unit, or returns zero if there are no more input values.

wcolumn.v

An integer variable that contains the column number of the last character written to the current output line, or zero if no character has been written. Each output unit has its own copy of this variable.

wopenerr.v

An integer variable that equals 1 to indicate that an error occurred when opening the file associated with the current output unit, or equals zero if no error occurred. If the **Open** statement for the unit specifies the **noerror** keyword, then the program can check the value of this variable after a **use** statement to determine whether an error occurred when opening the file; otherwise, such an error causes the program to terminate. Each output unit has its own copy of this variable.

wrecord.v

An integer variable that contains the number of lines written to the current output unit. Each output unit has its own copy of this variable.

write.v

An integer variable that contains the unit number of the current output unit. Its initial value is 6 because unit 6 (standard output) is the current output unit when a program begins execution. The assignment, **write.v = N**, changes the current output unit and has the same effect as the statement, **use N for output**.

5.05 Random-Number Generation

beta.f (double_arg1, double_arg2, integer_arg)

A double function that returns a random number in the range zero to one from the beta distribution having shape parameters α_1 equal to **double_arg1** and α_2 equal to **double_arg2**, and mean μ equal to $\frac{\alpha_1}{\alpha_1 + \alpha_2}$, where $\alpha_1 > 0$ and $\alpha_2 > 0$. **Integer_arg** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

binomial.f (integer_arg1, double_arg, integer_arg2)

An integer function that returns a random number in the range zero to n from the binomial distribution having parameters n equal to **integer_arg1** and p equal to **double_arg**, and mean μ equal to np , where $n > 0$ and $p > 0$. The return value represents a random number of successes in n independent trials where p is the probability of success for each trial. **Integer_arg2** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

If n equals 1, the binomial distribution is the same as the Bernoulli distribution.

erlang.f (double_arg, integer_arg1, integer_arg2)

A double function that returns a nonnegative random number from the Erlang distribution having mean μ equal to **double_arg**, shape parameter α equal to **integer_arg1**, and scale parameter β equal to $\frac{\mu}{\alpha}$, where $\mu > 0$ and $\alpha > 0$. **Integer_arg2** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

exponential.f (double_arg, integer_arg)

A double function that returns a nonnegative random number from the exponential distribution having mean μ equal to **double_arg**, where $\mu > 0$. **Integer_arg** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

gamma.f (double_arg1, double_arg2, integer_arg)

A double function that returns a nonnegative random number from the gamma distribution having mean μ equal to **double_arg1**, shape parameter α equal to **double_arg2**, and scale parameter β equal to $\frac{\mu}{\alpha}$, where $\mu > 0$ and $\alpha > 0$. **Integer_arg** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

If α equals 1, the gamma distribution is the same as the exponential distribution. If α is an integer, the gamma distribution is the same as the Erlang distribution. If μ is an integer and α equals $\frac{\mu}{2}$, the gamma distribution is the same as the chi-square distribution with μ degrees of freedom.

log.normal.f (double_arg1, double_arg2, integer_arg)

A double function that returns a nonnegative random number from the lognormal distribution having mean μ equal to **double_arg1** and standard deviation σ equal to **double_arg2**, where $\mu > 0$ and $\sigma > 0$. **Integer_arg** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

normal.f (double_arg1, double_arg2, integer_arg)

A double function that returns a random number from the normal distribution having mean μ equal to **double_arg1** and standard deviation σ equal to **double_arg2**, where $\sigma > 0$. **Integer_arg** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

poisson.f (double_arg, integer_arg)

An integer function that returns a nonnegative random number from the Poisson distribution having mean μ equal to **double_arg**, where $\mu > 0$. **Integer_arg** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

randi.f (integer_arg1, integer_arg2, integer_arg3)

An integer function that returns a random number in the range m to n from the discrete uniform distribution having parameters m equal to **integer_arg1** and n equal to **integer_arg2**, and mean μ equal to $\frac{m+n}{2}$, where $m \leq n$. **Integer_arg3** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

random.f (integer_arg)

A double function that returns a uniform random number in the range 0 to 1. **Integer_arg** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate equal to $1 - \text{random.f}(-\text{integer_arg})$.

seed.v

A one-dimensional integer array that contains the current seed value for each random number stream. A stream number is used as an index into the array. The number of array elements returned by **dim.f(seed.v)** is the number of streams and is initially 10; however, the program may **release** the array and **reserve** it to change the number of streams.

triang.f (double_arg1, double_arg2, double_arg3, integer_arg)

A double function that returns a random number in the range m to n from the triangular distribution having parameters m equal to **double_arg1**, peak k (the mode) equal to **double_arg2**, and n equal to **double_arg3**, and mean μ equal to $\frac{m+k+n}{3}$, where $m \leq k \leq n$. **Integer_arg** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

uniform.f (*double_arg1*, *double_arg2*, *integer_arg*)

A double function that returns a random number in the range m to n from the continuous uniform distribution having parameters m equal to ***double_arg1*** and n equal to ***double_arg2***, and mean μ equal to $\frac{m+n}{2}$, where $m \leq n$. ***integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

weibull.f (*double_arg1*, *double_arg2*, *integer_arg*)

A double function that returns a nonnegative random number from the Weibull distribution having shape parameter α equal to ***double_arg1*** and scale parameter β equal to ***double_arg2***, where $\alpha > 0$ and $\beta > 0$. ***integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

If α equals 1, the Weibull distribution is the same as the exponential distribution. If α equals 2, the Weibull distribution is the same as the Rayleigh distribution.

5.06 Simulation

between.v

A subprogram variable that specifies a routine to be called by the timing routine before each process method or process routine is executed, or contains zero (which is the default) if none is to be called. The process notice is removed from the event set (**ev.s**), and the simulation time (**time.v**) and event set index (**event.v**) are updated, before this routine is called; however, the pointer to the process notice (**process.v**) is not yet assigned.

date.f (integer_arg1, integer_arg2, integer_arg3)

An integer function that returns the number of days from the origin date (established by a prior call of **origin.r**) to the specified date, where month m equals **integer_arg1**, day d equals **integer_arg2**, and year y equals **integer_arg3**. The arguments must satisfy $1 \leq m \leq 12$, $1 \leq d \leq 31$, and $y \geq 100$.

day.f (double_arg)

An integer function that returns the day of the month in the range 1 to 31 for the date that is **double_arg** days after the origin date (established by a prior call of **origin.r**). The argument must be nonnegative.

ev.s

A one-dimensional array of sets called the “event set.” Each process method and process type in the program is assigned a unique index into this array. A smaller index value gives higher priority to the process method or process type. The set at an index contains a process notice for each scheduled invocation of the process method or process type associated with the index. The process notices are ranked within the set by increasing time of occurrence (**time.a**). The number of elements in this array is contained in **events.v**.

event.v

An integer variable that contains the event set index, in the range 1 to **events.v**, of the current process method or process type during a simulation.

events.v

An integer variable that contains the largest event set index, which is equal to the total number of process methods and process types defined by the program.

f.ev.s

A one-dimensional pointer array that contains in each element the reference value of the process notice for the most imminent invocation (smallest **time.a**) of a process method or process type, or is zero if there are no scheduled invocations. The number of elements in this array is contained in **events.v**.

hour.f (double_arg)

An integer function that returns the hour part, in the range 0 to **hours.v-1**, of the number of days specified by **double_arg**, which must be nonnegative.

hours.v

A double variable that specifies the number of hours per day. Its default value is 24.0.

l.ev.s

A one-dimensional pointer array that contains in each element the reference value of the process notice for the least imminent invocation (largest **time.a**) of a process method or process type, or is zero if there are no scheduled invocations. The number of elements in this array is contained in **events.v**.

minute.f (double_arg)

An integer function that returns the minute part, in the range 0 to **minutes.v-1**, of the number of days specified by **double_arg**, which must be nonnegative.

minutes.v

A double variable that specifies the number of minutes per hour. Its default value is 60.0.

month.f (*double_arg*)

An integer function that returns the month in the range 1 to 12 for the date that is ***double_arg*** days after the origin date (established by a prior call of **origin.r**). The argument must be nonnegative.

n.ev.s (*integer_arg*)

An integer function that returns the number of process notices in **ev.s(*integer_arg*)**. The argument must be in the range 1 to **events.v**.

nday.f (*double_arg*)

An integer function that returns the day part of the number of days specified by ***double_arg***, which must be nonnegative.

origin.r (*integer_arg1*, *integer_arg2*, *integer_arg3*)

A subroutine that establishes the specified date as the origin, where month *m* equals ***integer_arg1***, day *d* equals ***integer_arg2***, and year *y* equals ***integer_arg3***. The arguments must satisfy $1 \leq m \leq 12$, $1 \leq d \leq 31$, and $y \geq 100$.

process.v

A pointer variable that contains the reference value of the process notice for the current process method or process routine during a simulation, or zero if no process method or process routine is active.

time.v

A double variable that contains the current simulation time. Its initial value is zero, which corresponds to the start of the day of origin.

weekday.f (*double_arg*)

An integer function that returns the weekday, in the range 1 to 7 representing Sunday through Saturday, for the date that is ***double_arg*** days after the origin date. If no origin date has been established by a prior call of **origin.r**, the origin is assumed to be a Sunday. The argument must be nonnegative.

year.f (*double_arg*)

An integer function that returns the year for the date that is ***double_arg*** days after the origin date (established by a prior call of **origin.r**). The argument must be nonnegative.

5.07 Miscellaneous

batchtrace.v

An integer variable that specifies the action to take when a runtime error occurs. The debugger is invoked unless the value of the variable is 1 or 2. If the value is 1, a traceback is written to a file named “simerr.trc” and **snap.r** is called. If the value is 2, the program exits without a traceback or **snap.r** invocation. The default value is zero, which invokes the debugger.

date.r yielding *text_arg1*, *text_arg2*

A subroutine that returns the current date in the form **MM/DD/YYYY** in *text_arg1* and the current time in the form **HH:MM:SS** in *text_arg2*.

exit.r (*integer_arg*)

A subroutine that terminates the program with an exit status of *integer_arg*.

parm.v

A one-dimensional text array that contains the command-line arguments given to the program when it was invoked. **Dim.f(parm.v)** is the number of command-line arguments and is zero if no arguments were provided.

snap.r

A subroutine that may be provided by the program which is invoked when a runtime error occurs and the value of **batchtrace.v** is 1. The subroutine may write to the file named “simerr.trc” by writing to the current output unit.

6 Example Programs

All example programs from this manual can be found in SIMSCRIPT III installation directory `sim_examples`.

6.01 Example 1 - Gas Station with 2 attendants

This example describes a small, full service gas station where customers arrive randomly, queue up for service, receive service, and leave. Our goal might be to determine the effects of adding or deleting gas pumps or attendants from the system. For the moment, however, our real goal is merely to construct and execute a very simple SIMSCRIPT III model.

To introduce randomness, assume that we have a source of uniformly distributed random numbers for which we can establish bounds. In SIMSCRIPT III this is referenced as `uniform.f`. This function has three parameters: the lower bound, the upper bound, and a random number stream. Each time the function is executed, a new sample from the interval is computed.

Customers arrive, request service, wait if no server is available, occupy the server for awhile, and then depart. A customer is modeled as object with two process methods, one for customer generation and one to fill-up the gas. For this model, it is sufficient to model the attendant (server) as a resource. Resource subsystem is imported for this reason, and attendant inherits all the properties of the `RESOURCE` class and adds the `PRINT.STATISTICS` method.

Two measures that are commonly desired for this type of model are statistics on the queue (average, variance, maximum, etc.) and the utilization of the resources. These can be easily obtained in SIMSCRIPT III by using the `accumulate` statement. The `accumulate` statement provides a simple means of specifying which measurements are desired without requiring detailed specification of the method of measurement. An `accumulate` statement is placed in the `preamble`. As the variable of interest changes during the course of the simulation, the system automatically captures the pertinent information, in this case average and maximum queue length and attendant utilization.

```
preamble for the GAS.STATION system 'Example 1
  importing the RESOURCE subsystem

  begin class CUSTOMER

    the class
      has a FILL.UP process method
      and a GENERATOR process method

  end
```

```

begin class ATTENDANT
    every ATTENDANT
        is a RESOURCE and
        has a PRINT.STATISTICS method

    accumulate AVG.QLEN as the average,
        MAX.QLEN as the maximum of N.QUEUE
    accumulate AVG.BUSY as the average of ACQUIRED.UNITS

end

end

process method CUSTOMER'FILL.UP

    if AVAILABLE.UNITS(ATTENDANT) = 0 'no attendants available
        call WAIT.FOR(ATTENDANT)(1, 0) 'wait for an attendant
    else
        add 1 to ACQUIRED.UNITS(ATTENDANT)
    always

    work UNIFORM.F(5.0, 15.0, 2) minutes 'fill up

    subtract 1 from ACQUIRED.UNITS(ATTENDANT)

end

process method CUSTOMER'GENERATOR

    define I as an integer variable

    for I = 1 to 1000
    do
        schedule a FILL.UP now
        wait UNIFORM.F(2.0, 8.0, 1) minutes
    loop

end

method ATTENDANT'PRINT.STATISTICS

    print 3 lines with AVG.QLEN, MAX.QLEN, 100 * AVG.BUSY / TOTAL.UNITS thus
    AVERAGE CUSTOMER QUEUE LENGTH IS      *.***
    MAXIMUM CUSTOMER QUEUE LENGTH IS      *
    THE ATTENDANTS WERE BUSY  **.** PER CENT OF THE TIME.

end

main

    create ATTENDANT 'reference value stored in global variable
    TOTAL.UNITS(ATTENDANT) = 2

    schedule a CUSTOMER'GENERATOR now
    start simulation

    print 1 line thus
SIMPLE GAS STATION MODEL WITH 2 ATTENDANTS
    call PRINT.STATISTICS(ATTENDANT)
    Read as / using unit 5
end

```

```

public preamble for the RESOURCE subsystem

begin class RESOURCE

    every RESOURCE
        has a TOTAL.UNITS,
            an ACQUIRED.UNITS,
            an AVAILABLE.UNITS method,
            a WAIT.FOR method,
        and a CLEAN.UP method, and
        owns a REQUEST'QUEUE

        define TOTAL.UNITS      as an integer variable
        define ACQUIRED.UNITS   as an integer variable monitored on the left
        define AVAILABLE.UNITS  as an integer method
        define WAIT.FOR        as a method
            given 2 integer values 'requested units and priority
            before destroying a RESOURCE, call CLEAN.UP

    end

begin class REQUEST

    every REQUEST
        has a UNITS,
            a PRIORITY,
        and a PROCESS.NOTICE, and
        belongs to a QUEUE

        define UNITS, PRIORITY as integer variables
        define PROCESS.NOTICE  as a pointer variable
        define QUEUE as a set ranked by high PRIORITY

    end

end

methods for the RESOURCE class

left method ACQUIRED.UNITS

    define ACQ as an integer variable
    define REQ as a REQUEST reference variable

    enter with ACQ

    while QUEUE is not empty and UNITS(F.QUEUE) <= TOTAL.UNITS - ACQ
    do
        remove first REQ from QUEUE
        add UNITS(REQ) to ACQ
        schedule the PROCESS.NOTICE(REQ) now
        destroy REQ
    loop

    move from ACQ

end

method AVAILABLE.UNITS

    return with TOTAL.UNITS - ACQUIRED.UNITS

end

```

```
method WAIT.FOR(REQ.UNITS, REQ.PRIORITY)

  define REQ as a REQUEST reference variable

  create REQ
  UNITS(REQ) = REQ.UNITS
  PRIORITY(REQ) = REQ.PRIORITY
  PROCESS.NOTICE(REQ) = PROCESS.V
  file REQ in QUEUE
  suspend

end

method CLEAN.UP

  define REQ as a REQUEST reference variable

  while QUEUE is not empty
  do
    remove first REQ from QUEUE
    destroy PROCESS.NOTICE(REQ)
    destroy REQ
  loop

end
```

6.02 Example 2 - Gas Station with 2 attendants and 2 grades of gasoline

To illustrate the ease of changing the modeling requirements and explore alternatives in design representing them in SIMSCRIPT III, the Example 1 model will be modified as follows:

- a. Customers choose whether they require premium or regular gas. They must wait for the proper pump to become available. We will arbitrarily assume that 70% require **premium** and 30% require **regular**. Initially, we will arbitrarily have one regular pump and three premium pumps.
- b. An attendant is not required until the pump is available. The attendant starts the pump and then is free until the pump stops. He is then required in order to complete the service.
- c. The queues and utilization for the attendants and two types of pumps will be measured.

```
preamble for the GAS.STATION system 'Example 2
importing the RESOURCE subsystem

begin class CUSTOMER

  the class
    has a FILL.UP process method
    and a GENERATOR process method
end

begin class GAS.STATION.RESOURCE

  every GAS.STATION.RESOURCE
    is a RESOURCE and
    has a REQUEST method,
    a RELINQUISH method,
    and a UTILIZATION method

  accumulate AVG.QLEN as the average,
    MAX.QLEN as the maximum of N.QUEUE
  accumulate AVG.BUSY as the average of ACQUIRED.UNITS

  define UTILIZATION as a double method
end

begin class ATTENDANT

  every ATTENDANT
    is a GAS.STATION.RESOURCE and
    has a PRINT.STATISTICS method
end
```

```

begin class PUMP

  every PUMP
    is a GAS.STATION.RESOURCE and
    has a PRINT.STATISTICS method
  define PRINT.STATISTICS as a method
    given a text argument 'name of grade

  the class
    has a REGULAR,
      a PREMIUM,
      a PRINT.ALL.STATISTICS method,
    and a SELECT method

  define REGULAR, PREMIUM as PUMP reference variables
  define SELECT as a PUMP reference method

end

end

methods for the CUSTOMER class

process method FILL.UP

  define PUMP as a PUMP reference variable

  PUMP = PUMP'SELECT

  call REQUEST(PUMP)

  call REQUEST(ATTENDANT)
  work UNIFORM.F(2, 4, 2) minutes ''insert nozzle
  call RELINQUISH(ATTENDANT)

  work UNIFORM.F(5, 9, 2) minutes ''fill up unattended

  call REQUEST(ATTENDANT)
  work UNIFORM.F(3, 5, 2) minutes ''remove nozzle
  call RELINQUISH(ATTENDANT)

  call RELINQUISH(PUMP)

end

process method GENERATOR

  define I as an integer variable

  for I = 1 to 1000
  do
    schedule a FILL.UP now
    wait UNIFORM.F(2, 8, 1) minutes
  loop

end

```

methods for the GAS.STATION.RESOURCE class

method REQUEST

```
    if AVAILABLE.UNITS = 0
        call WAIT.FOR(1, 0)
    else
        add 1 to ACQUIRED.UNITS
    always
```

end

method RELINQUISH

```
    subtract 1 from ACQUIRED.UNITS
```

end

method UTILIZATION

```
    return with 100 * AVG.BUSY / TOTAL.UNITS
end
```

methods for the ATTENDANT class

method PRINT.STATISTICS

```
    print 3 lines with AVG.QLEN, MAX.QLEN, UTILIZATION thus
    AVERAGE QUEUE WAITING FOR ATTENDANTS IS    *.* CUSTOMERS
    MAXIMUM   "   "   "   "   "   "   *
    THE ATTENDANTS WERE BUSY    *.* PER CENT OF THE TIME.
```

end

methods for the PUMP class

method PRINT.STATISTICS(GRADE)

```
    print 1 line with GRADE, AVG.QLEN, MAX.QLEN, UTILIZATION thus
*****:      *.*          *          *.* PERCENT
```

end

method PRINT.ALL.STATISTICS

```
    print 3 lines thus
```

```
    THE QUEUES FOR THE PUMPS WERE AS FOLLOWS:
    GRADE      AVERAGE      MAXIMUM      UTILIZATION
```

```
    call PRINT.STATISTICS(REGULAR)("REGULAR")
    call PRINT.STATISTICS(PREMIUM)("PREMIUM")
end
```

```
method SELECT
    if RANDOM.F(3) > 0.70
        return with REGULAR
    otherwise
        return with PREMIUM
end

main
    create ATTENDANT
    TOTAL.UNITS(ATTENDANT) = 2

    create PUMP'REGULAR
    create PUMP'PREMIUM
    TOTAL.UNITS(PUMP'REGULAR) = 1
    TOTAL.UNITS(PUMP'PREMIUM) = 3

    schedule a CUSTOMER'GENERATOR now
    start simulation

    print 2 line thus
SIMPLE GAS STATION WITH TWO ATTENDANTS
    AND TWO GRADES OF GASOLINE
    call PRINT.STATISTICS(ATTENDANT)
    call PUMP'PRINT.ALL.STATISTICS

    Read as / using unit 5
end
```

```

public preamble for the RESOURCE subsystem

begin class RESOURCE

    every RESOURCE
        has a TOTAL.UNITS,
            an ACQUIRED.UNITS,
            an AVAILABLE.UNITS method,
            a WAIT.FOR method,
        and a CLEAN.UP method, and
        owns a REQUEST'QUEUE

        define TOTAL.UNITS      as an integer variable
        define ACQUIRED.UNITS   as an integer variable monitored on the left
        define AVAILABLE.UNITS  as an integer method
        define WAIT.FOR        as a method
            given 2 integer values 'requested units and priority
            before destroying a RESOURCE, call CLEAN.UP

    end

begin class REQUEST

    every REQUEST
        has a UNITS,
            a PRIORITY,
        and a PROCESS.NOTICE, and
        belongs to a QUEUE

        define UNITS, PRIORITY as integer variables
        define PROCESS.NOTICE  as a pointer variable
        define QUEUE as a set ranked by high PRIORITY

    end

end

methods for the RESOURCE class

left method ACQUIRED.UNITS

    define ACQ as an integer variable
    define REQ as a REQUEST reference variable

    enter with ACQ

    while QUEUE is not empty and UNITS(F.QUEUE) <= TOTAL.UNITS - ACQ
    do
        remove first REQ from QUEUE
        add UNITS(REQ) to ACQ
        schedule the PROCESS.NOTICE(REQ) now
        destroy REQ
    loop

    move from ACQ

end

```

```
method AVAILABLE.UNITS
    return with TOTAL.UNITS - ACQUIRED.UNITS
end

method WAIT.FOR(REQ.UNITS, REQ.PRIORITY)
    define REQ as a REQUEST reference variable

    create REQ
    UNITS(REQ) = REQ.UNITS
    PRIORITY(REQ) = REQ.PRIORITY
    PROCESS.NOTICE(REQ) = PROCESS.V
    file REQ in QUEUE
    suspend
end

method CLEAN.UP
    define REQ as a REQUEST reference variable

    while QUEUE is not empty
    do
        remove first REQ from QUEUE
        destroy PROCESS.NOTICE(REQ)
        destroy REQ
    loop
end
```

6.03 Example 3 - A Bank with Separate Queue for Each Teller

This example illustrates modeling a bank in which customers arrive, go immediately to any available teller, receive service, and leave. If all tellers are busy, the arriving customer joins the shortest line, and waits there until served. While it is an unreasonable assumption that customers remain in one line when another teller becomes available, it will give an extreme value for comparing this model to the newer-style bank in which all customers form a single line to be served by the next available teller. When jockeying is permitted, the single line is the optimum equivalent.

The measures required are the average and maximum queues, the utilization of each separate teller, and the mean waiting time for all customers.

This system will be modeled using processes methods. The customer **GENERATOR** process method will inject customers into the system with an exponential distribution of inter-arrival times until a pre-specified closing time. The model will terminate after all customers who arrived before closing time finish being served.

The **CUSTOMER** object with its process methods will contain the complete description of the actions of one customer from arrival at the bank until departure.

Input parameters for one model run will be read from the file ex3.dat, which contains number of tellers, mean inter-arrival time, mean service time and the day length. The input data can be introduced to the model using interactive graphical input forms, described in the SIMSCRIPT III Graphics Manual.

```
Input data in file ex3.dat
```

```
2  
5.0  
10.0  
8.0
```

```

preamble for the BANK system 'Example 3
  importing the RESOURCE subsystem

begin class CUSTOMER

  the class
    has a WAITING.TIME, 'in minutes,
      a BANK.VISIT process method,
      and a GENERATOR process method

  define WAITING.TIME as a real variable
  tally MEAN.WAITING.TIME as the mean of WAITING.TIME

  define GENERATOR as a process method
    given 2 real values 'day length in hours and
                        'mean interarrival time in minutes

end

begin class TELLER

  every TELLER
    is a RESOURCE,
    has an ID.NUMBER
    and an ENGAGE method, and
    belongs to the TELLER.POOL

  define ID.NUMBER as an integer variable

  define ENGAGE as a method
    yielding 1 real value 'waiting time in minutes

  accumulate UTILIZATION as the mean of ACQUIRED.UNITS
  accumulate AVG.QLEN as the mean,
    MAX.QLEN as the maximum of N.QUEUE

  the class
    has a MEAN.SERVICE.TIME, 'in minutes
    an INITIALIZE method,
    a SELECT method,
    and a PRINT.STATISTICS method, and
    owns the TELLER.POOL

  define MEAN.SERVICE.TIME as a real variable

  define INITIALIZE as a method
    given 1 integer value 'number of tellers
    and 1 real value 'mean service time in minutes

  define SELECT as a TELLER reference method

end

end

methods for the CUSTOMER class

process method BANK.VISIT

  call ENGAGE(TELLER'SELECT) yielding WAITING.TIME

end

```

```

process method GENERATOR(DAY.LENGTH, MEAN.INTERARRIVAL.TIME)

    define TIME.TO.CLOSE as a real variable

    TIME.TO.CLOSE = DAY.LENGTH / HOURS.V

    until TIME.V >= TIME.TO.CLOSE
    do
        schedule a BANK.VISIT now
        wait EXPONENTIAL.F(MEAN.INTERARRIVAL.TIME, 1) minutes
    loop
end

methods for the TELLER class

method ENGAGE yielding WAIT

    if ACQUIRED.UNITS = 1 'teller is busy
        define START.TIME as a real variable
        START.TIME = TIME.V
        call WAIT.FOR(1, 0)
        WAIT = (TIME.V - START.TIME) * HOURS.V * MINUTES.V
    else
        ACQUIRED.UNITS = 1
    always

    work EXPONENTIAL.F(MEAN.SERVICE.TIME, 2) minutes

    ACQUIRED.UNITS = 0 'free the teller

end

method INITIALIZE(NO.OF.TELLERS, MST)

    define ID as an integer variable
    define TELLER as a TELLER reference variable

    for ID = 1 to NO.OF.TELLERS
    do
        create TELLER
        ID.NUMBER(TELLER) = ID
        TOTAL.UNITS(TELLER) = 1
        file TELLER in TELLER.POOL
    loop

    MEAN.SERVICE.TIME = MST
end

method SELECT

    define TELLER, CHOICE as TELLER reference variables

    for each TELLER in TELLER.POOL with ACQUIRED.UNITS(TELLER) = 0
        find the first case
    if found
        return with TELLER
    otherwise

    for each TELLER in TELLER.POOL
        compute CHOICE as the minimum(TELLER) of N.QUEUE(TELLER)
    return with CHOICE

end

```

```

method PRINT.STATISTICS

    define TELLER as a TELLER reference variable

    print 4 lines thus

TELLER          UTILIZATION          QUEUE LENGTH
                AVERAGE             MAXIMUM

    for each TELLER in TELLER.POOL
        print 1 line with ID.NUMBER(TELLER), UTILIZATION(TELLER),
            AVG.QLEN(TELLER), MAX.QLEN(TELLER) thus
    *           *.**                 *.**                 *
end

main

    define NO.OF.TELLERS as an integer variable
    define MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME, DAY.LENGTH
        as real variables

    open unit 1 for input, name is "ed_ex3.dat"
    use unit 1 for input

    read NO.OF.TELLERS, MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME, DAY.LENGTH
    call TELLER'INITIALIZE(NO.OF.TELLERS, MEAN.SERVICE.TIME)

    schedule a CUSTOMER'GENERATOR(DAY.LENGTH, MEAN.INTERARRIVAL.TIME) now
    start simulation

    print 10 lines with NO.OF.TELLERS, MEAN.INTERARRIVAL.TIME,
        MEAN.SERVICE.TIME, DAY.LENGTH, TIME.V * HOURS.V,
        CUSTOMER'MEAN.WAITING.TIME thus
SIMULATION OF A BANK WITH * TELLERS
(EACH WITH A SEPARATE QUEUE)
CUSTOMERS ARRIVE ACCORDING TO AN EXPONENTIAL DISTRIBUTION
OF INTER ARRIVAL TIMES WITH A MEAN OF *.* MINUTES.
SERVICE TIME IS ALSO EXPONENTIALLY DISTRIBUTED
WITH A MEAN OF *.* MINUTES.
THE BANK DOORS ARE CLOSED AFTER *.* HOURS.
(BUT ALL CUSTOMERS INSIDE ARE SERVED.)
THE LAST CUSTOMER LEFT THE BANK AT *.* HOURS.
THE AVERAGE CUSTOMER DELAY WAS *.* MINUTES.

    call TELLER'PRINT.STATISTICS

    Read as / using unit 5 '' to keep the window open

end

```

```

public preamble for the RESOURCE subsystem

begin class RESOURCE

    every RESOURCE
        has a TOTAL.UNITS,
            an ACQUIRED.UNITS,
            an AVAILABLE.UNITS method,
            a WAIT.FOR method,
        and a CLEAN.UP method, and
        owns a REQUEST'QUEUE

        define TOTAL.UNITS as an integer variable
        define ACQUIRED.UNITS as an integer variable monitored on the left
        define AVAILABLE.UNITS as an integer method
        define WAIT.FOR as a method
            given 2 integer values 'requested units and priority
            before destroying a RESOURCE, call CLEAN.UP

    end

begin class REQUEST

    every REQUEST
        has a UNITS,
            a PRIORITY,
        and a PROCESS.NOTICE, and
        belongs to a QUEUE

        define UNITS, PRIORITY as integer variables
        define PROCESS.NOTICE as a pointer variable
        define QUEUE as a set ranked by high PRIORITY

    end

end

methods for the RESOURCE class

left method ACQUIRED.UNITS

    define ACQ as an integer variable
    define REQ as a REQUEST reference variable

    enter with ACQ

    while QUEUE is not empty and UNITS(F.QUEUE) <= TOTAL.UNITS - ACQ
    do
        remove first REQ from QUEUE
        add UNITS(REQ) to ACQ
        schedule the PROCESS.NOTICE(REQ) now
        destroy REQ
    loop

    move from ACQ

end

method AVAILABLE.UNITS

    return with TOTAL.UNITS - ACQUIRED.UNITS

end

```

```
method WAIT.FOR(REQ.UNITS, REQ.PRIORITY)

  define REQ as a REQUEST reference variable

  create REQ
  UNITS(REQ) = REQ.UNITS
  PRIORITY(REQ) = REQ.PRIORITY
  PROCESS.NOTICE(REQ) = PROCESS.V
  file REQ in QUEUE
  suspend

end

method CLEAN.UP

  define REQ as a REQUEST reference variable

  while QUEUE is not empty
  do
    remove first REQ from QUEUE
    destroy PROCESS.NOTICE(REQ)
    destroy REQ
  loop

end
```

6.04 Example 4 - A Harbor Model

This example models the unloading of ships in a harbor using two cranes mounted on a track. There is room at dockside for two ships. When two ships are there, each crane unloads one of them. If only one ship is there, both cranes work on it, reducing its unloading time by a factor of two. If another ship arrives while both cranes are serving one ship, one crane will immediately begin to serve the new ship. Waiting ships are unloaded first-come-first-served.

Assume that ships arrive on the average of three in every four days, but inter-arrival times are exponentially distributed (the mean is thus 4/3 days). The unloading time is uniformly distributed between 0.5 and 1.5 days.

The desired results are the maximum and average queue length and the cycle times of ships (queuing plus unloading)—minimum, maximum and average. These results should be reported after 80 days of continuous operation.

This comprehensive example will illustrate the concepts of process methods interactions and sets.

The ship will be modeled as an object with UNLOAD process method, method for RESCHEDULE.UNLOAD and DONE.WAITING, to model the interactions between ships by allocating the cranes and adjusting unloading times from ships.

A separate GENERATOR process method will model the arrival of ships. At first glance, it might seem natural to model the cranes as a single resource with one subgroup and two units. However, the rules for operating cranes do not readily translate into the simple operations of the resources (**request/ relinquish**). Two sets are introduced, a set called **DOCK** to contain ships being unloaded. There can be a maximum of two and a set called **QUEUE** to contain ships waiting for the **DOCK**.

```
preamble for the HARBOR system 'Example 4
begin class SHIP
  every SHIP
    has an UNLOAD process method and
      a DONE.WAITING method, and
      a RESCHEDULE.UNLOAD method, and
    belongs to a QUEUE and a DOCK
  define RESCHEDULE.UNLOAD as a method
    given a real argument 'time scale factor
```

```

the class
  has a CYCLE.TIME,
      a GENERATOR process method,
  and a STOP.SIMULATION process method, and
  owns the QUEUE and the DOCK

define CYCLE.TIME as a real variable
tally NO.OF.SHIPS      as the number,
      MIN.CYCLE.TIME   as the minimum,
      MAX.CYCLE.TIME   as the maximum,
      MEAN.CYCLE.TIME  as the mean of CYCLE.TIME

accumulate MAX.QLENGTH as the maximum,
      MEAN.QLENGTH as the mean of N.QUEUE

end

end

methods for the SHIP class

process method UNLOAD

  define ARRIVE.TIME, UNLOADING.TIME as real variables

  ARRIVE.TIME = TIME.V
  UNLOADING.TIME = UNIFORM.F(0.5, 1.5, 2)

  if N.DOCK < 2
    if N.DOCK = 1 'an existing ship is using both cranes
      call RESCHEDULE.UNLOAD(F.DOCK)(2) 'give up one crane
    else 'no existing ships, so this ship will use both cranes
      UNLOADING.TIME = UNLOADING.TIME / 2
    always
    file SHIP in DOCK
  else 'no room at the dock, must wait in the queue
    file SHIP in QUEUE
    suspend
  always

  work UNLOADING.TIME days

  remove SHIP from DOCK
  destroy SHIP
  CYCLE.TIME = TIME.V - ARRIVE.TIME

  if QUEUE is not empty
    call DONE.WAITING(F.QUEUE)
  else
    if N.DOCK = 1
      call RESCHEDULE.UNLOAD(F.DOCK)(0.5) 'gain a crane
    always
  always

end

method DONE.WAITING

  remove SHIP from QUEUE
  file SHIP in DOCK
  schedule the UNLOAD now

end

```

```

method RESCHEDULE.UNLOAD(SCALE.FACTOR)

    interrupt UNLOAD
    TIME.A(UNLOAD) = TIME.A(UNLOAD) * SCALE.FACTOR
    resume UNLOAD

end

process method GENERATOR

    define SHIP as a SHIP reference variable

    until TIME.V > 80
    do
        create SHIP
        schedule an UNLOAD(SHIP) now
        wait EXPONENTIAL.F(4/3, 1) days
    loop

end

process method STOP.SIMULATION

    print 5 lines with NO.OF.SHIPS, TIME.V, MIN.CYCLE.TIME,
    MAX.CYCLE.TIME, MEAN.CYCLE.TIME thus
        SHIP AND CRANE MODEL
    * SHIPS WERE UNLOADED IN *.* DAYS
    THE MINIMUM TIME TO UNLOAD A SHIP WAS *.*
    " MAXIMUM " " " " " " *.*
    " MEAN " " " " " " *.*

    skip 3 lines

    print 2 lines with MEAN.QLENGTH, MAX.QLENGTH thus
    THE AVERAGE QUEUE OF SHIPS WAITING TO BE UNLOADED WAS *.*
    THE MAXIMUM QUEUE WAS *

    'stop

end

main

    schedule a SHIP'GENERATOR now
    schedule a SHIP'STOP.SIMULATION in 80 days
    start simulation

    read as / using unit 5 ' ' to keep text window open

end

```

6.05 Example 5 - The Modern Bank (Single-Queue-Multiple-Server)

This example represents the model of a bank with a single queue and a varying number of tellers. The task is to make a decision on the “optimum” number of tellers to employ. This will be determined by making multiple experimental runs in a single execution of the program. To increase confidence in the results, several replications will be made with each number of tellers. The results for each replication will be reported, as well as the average over all the replications.

The measurements required are the average and maximum daily queues, the average utilization of tellers, and mean waiting time for all customers. The queue-length and cycle-time histograms will be generated for the composite (replicated) runs.

All of the replication and reporting operations are incorporated in the `main` program. The daily and overall statistics are segregated by use of a qualifier “`DAILY.`” Input data are read from file `ex5.dat`

Input data in file `ex5.dat`

```

1 3
5
5.00
10.00
8.00
(SHOULD BE 1.00      24.97      58      251.57 )
(SHOULD BE .92      4.61      28      25.25 )
(SHOULD BE .67      .60      6      2.99 )

```

```

preamble for the BANK system 'Example 5
importing the RESOURCE subsystem

begin class CUSTOMER

  the class
    has a WAITING.TIME, 'in minutes,
        a BANK.VISIT process method,
        and a GENERATOR process method

  define WAITING.TIME as a real variable
  tally DAILY.MEAN.WAITING.TIME as the DAILY mean,
        MEAN.WAITING.TIME as the mean,
        WAIT.HISTOGRAM(0 to 100 by 5) as the histogram
        of WAITING.TIME

  define GENERATOR as a process method
    given 2 real values 'day length in hours and
                        'mean interarrival time in minutes

end

```

```

begin class TELLER

    every TELLER
        is a RESOURCE and
        has an ENGAGE method

    define ENGAGE as a method
        yielding 1 real value 'waiting time in minutes

    accumulate DAILY.AVG.BUSY as the DAILY mean,
        AVG.BUSY as the mean
        of ACQUIRED.UNITS

    accumulate DAILY.AVG.QLEN as the DAILY mean,
        DAILY.MAX.QLEN as the DAILY maximum,
        AVG.QLEN as the mean,
        MAX.QLEN as the maximum,
        QLEN.HISTOGRAM(0 to 20 by 1) as the histogram
        of N.QUEUE

    the class
        has a MEAN.SERVICE.TIME 'in minutes

    define MEAN.SERVICE.TIME as a real variable

end

define SIMULATE.BANK as a routine
    given 4 integer values 'no. of tellers, no. of replications,
        'stream 1 seed, stream 2 seed,
        and 3 real values    'mean interarrival time in minutes,
        'mean service time in minutes,
        'day length in hours

end

process method CUSTOMER'BANK.VISIT

    call ENGAGE(TELLER) yielding WAITING.TIME

end

process method CUSTOMER'GENERATOR(DAY.LENGTH, MEAN.INTERARRIVAL.TIME)

    define TIME.TO.CLOSE as a real variable

    TIME.TO.CLOSE = TIME.V + DAY.LENGTH / HOURS.V

    until TIME.V >= TIME.TO.CLOSE
    do
        schedule a BANK.VISIT now
        wait EXPONENTIAL.F(MEAN.INTERARRIVAL.TIME, 1) minutes
    loop

end

```

```

method TELLER'ENGAGE yielding WAIT

  if AVAILABLE.UNITS = 0
    define START.TIME as a real variable
    START.TIME = TIME.V
    call WAIT.FOR(1, 0)
    WAIT = (TIME.V - START.TIME) * HOURS.V * MINUTES.V
  else
    add 1 to ACQUIRED.UNITS
  always

  work EXPONENTIAL.F(MEAN.SERVICE.TIME, 2) minutes

  subtract 1 from ACQUIRED.UNITS 'free the teller

end

routine SIMULATE.BANK
  given NO.OF.TELLERS, NO.OF.REPLICATIONS, SEED1, SEED2,
        MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME, DAY.LENGTH

  define I as an integer variable
  define START.TIME as a real variable

  TIME.V = 0
  SEED.V(1) = SEED1
  SEED.V(2) = SEED2
  reset totals of CUSTOMER'WAITING.TIME

  create TELLER
  TOTAL.UNITS(TELLER) = NO.OF.TELLERS
  TELLER'MEAN.SERVICE.TIME = MEAN.SERVICE.TIME

  skip 2 lines
  print 5 lines with NO.OF.TELLERS thus
NUMBER OF TELLERS = *

FINISH      TELLER      QUEUE LENGTH      AVERAGE CUSTOMER
TIME  UTILIZATION  AVERAGE  MAXIMUM      WAITING TIME
(HOURS)                                     (MINUTES)

  for I = 1 to NO.OF.REPLICATIONS
  do
    START.TIME = TIME.V
    reset DAILY totals of CUSTOMER'WAITING.TIME,
    ACQUIRED.UNITS(TELLER), N.QUEUE(TELLER)
    schedule a CUSTOMER'GENERATOR(DAY.LENGTH, MEAN.INTERARRIVAL.TIME) now
    start simulation
    print 1 line with (TIME.V - START.TIME) * HOURS.V,
    DAILY.AVG.BUSY(TELLER) / NO.OF.TELLERS, DAILY.AVG.QLEN(TELLER),
    DAILY.MAX.QLEN(TELLER), CUSTOMER'DAILY.MEAN.WAITING.TIME thus
*.**          *.*          *.*          *          *.*
  loop

print 4 lines with AVG.BUSY(TELLER) / NO.OF.TELLERS, AVG.QLEN(TELLER),
  MAX.QLEN(TELLER), CUSTOMER'MEAN.WAITING.TIME thus

AVERAGE OVER ALL REPLICATIONS:

          *.*          *.*          *          *.*

skip 3 lines

```

```

print 3 lines with CUSTOMER'WAIT.HISTOGRAM(1),
QLEN.HISTOGRAM(TELLER)(1) / TIME.V thus
  WAITING TIME   NO. WHO WAITED   QUEUE LENGTH   PERCENTAGE
  (MINUTES)     THIS TIME                       OF TIME
      T < 5                *                       0                *.****
for I = 2 to 20
  print 1 line with 5 * (I - 1), 5 * I, CUSTOMER'WAIT.HISTOGRAM(I),
  I - 1, QLEN.HISTOGRAM(TELLER)(I) / TIME.V thus
  * <= T < *                *                       *                *.****
print 1 line with CUSTOMER'WAIT.HISTOGRAM(21),
QLEN.HISTOGRAM(TELLER)(21) / TIME.V thus
100 <= T                *                       20                *.****

destroy TELLER

end

main

define MIN.TELLERS, MAX.TELLERS, NO.OF.TELLERS, NO.OF.REPLICATIONS,
  SEED1, SEED2 as integer variables
define MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME, DAY.LENGTH
  as real variables

open unit 1 for input, name is "ex5.dat"
use unit 1 for input

read MIN.TELLERS, MAX.TELLERS, NO.OF.REPLICATIONS,
MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME, DAY.LENGTH

print 9 lines with MIN.TELLERS, MAX.TELLERS, NO.OF.REPLICATIONS,
MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME, DAY.LENGTH thus
  SIMULATION OF A SINGLE-QUEUE BANK
  THE NO. OF TELLERS RANGES FROM * TO *
  ( * REPLICATIONS FOR EACH NO. OF TELLERS)
CUSTOMERS ARRIVE ACCORDING TO AN EXPONENTIAL DISTRIBUTION
  OF INTER ARRIVAL TIMES WITH A MEAN OF *.* MINUTES.
SERVICE TIME IS ALSO EXPONENTIALLY DISTRIBUTED
  WITH A MEAN OF *.* MINUTES.
THE BANK DOORS ARE CLOSED AFTER *.* HOURS (EACH DAY).
  (BUT ALL CUSTOMERS INSIDE ARE SERVED.)

SEED1 = SEED.V(1)
SEED2 = SEED.V(2)

for NO.OF.TELLERS = MIN.TELLERS to MAX.TELLERS
do
  call SIMULATE.BANK given NO.OF.TELLERS, NO.OF.REPLICATIONS, SEED1,
  SEED2, MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME, DAY.LENGTH
  start new page
loop

read as / using unit 5 '' to keep text window open

end

```

```

public preamble for the RESOURCE subsystem

begin class RESOURCE

  every RESOURCE
    has a TOTAL.UNITS,
        an ACQUIRED.UNITS,
        an AVAILABLE.UNITS method,
        a WAIT.FOR method,
    and a CLEAN.UP method, and
    owns a REQUEST'QUEUE

    define TOTAL.UNITS      as an integer variable
    define ACQUIRED.UNITS   as an integer variable monitored on the left
    define AVAILABLE.UNITS  as an integer method
    define WAIT.FOR         as a method
        given 2 integer values 'requested units and priority
    before destroying a RESOURCE, call CLEAN.UP

  end

begin class REQUEST

  every REQUEST
    has a UNITS,
        a PRIORITY,
    and a PROCESS.NOTICE, and
    belongs to a QUEUE

    define UNITS, PRIORITY as integer variables
    define PROCESS.NOTICE as a pointer variable
    define QUEUE as a set ranked by high PRIORITY

  end

end

methods for the RESOURCE class

left method ACQUIRED.UNITS

  define ACQ as an integer variable
  define REQ as a REQUEST reference variable

  enter with ACQ

  while QUEUE is not empty and UNITS(F.QUEUE) <= TOTAL.UNITS - ACQ
  do
    remove first REQ from QUEUE
    add UNITS(REQ) to ACQ
    schedule the PROCESS.NOTICE(REQ) now
    destroy REQ
  loop

  move from ACQ

end

```

```
method AVAILABLE.UNITS

  return with TOTAL.UNITS - ACQUIRED.UNITS
end

method WAIT.FOR(REQ.UNITS, REQ.PRIORITY)

  define REQ as a REQUEST reference variable

  create REQ
  UNITS(REQ) = REQ.UNITS
  PRIORITY(REQ) = REQ.PRIORITY
  PROCESS.NOTICE(REQ) = PROCESS.V
  file REQ in QUEUE
  suspend

end

method CLEAN.UP

  define REQ as a REQUEST reference variable

  while QUEUE is not empty
  do
    remove first REQ from QUEUE
    destroy PROCESS.NOTICE(REQ)
    destroy REQ
  loop

end
```

6.06 Example 6 - A Job Shop Model

This example models a typical job shop in which several types of jobs are done. The system is described as follows:

There are any number of different “machines” in the shop. There may be several copies of identical machines clustered in groups. There are any number of prototype jobs. Jobs arrive according to a Poisson process at which time their types are randomly chosen according to an arbitrary distribution. Each job prototype consists of a sequence of tasks. Each task has a specified machine group and mean completion time. The actual task times are sampled from exponential distributions using the given mean.

The purpose of the study is to evaluate the performance of the shop for a particular workload, measuring the utilization of each machine group, and overall delays in the shop. Experiments which might typically be conducted with such a model include evaluating the impact of reconfiguring the machines in the shop (adding or deleting), deciding whether **the system** could handle additional job types, whether reordering the tasks for a certain job type would improve turnaround, etc.

The model is implemented using objects MACHINE, JOB and TASK. Input data is read from ex6.dat.

Input data in file ex6.dat

```

6
14 CASTING_UNITS
5 LATHES
4 PLANES
8 DRILL_PRESSES
16 SHAPERS
4 POLISHING_MACHINES
3
FIRST
2.0833 CASTING_UNITS
0.5833 PLANES
0.3333 LATHES
1.0 POLISHING_MACHINES
SECOND
1.75 SHAPERS
1.5 DRILL_PRESSES
1.0833 LATHES
THIRD
3.9166 CASTING_UNITS
4.1666 SHAPERS
0.8333 DRILL_PRESSES
0.5 PLANES
0.4166 POLISHING_MACHINES
END
0.16 40
.241 1 .44 2 .32 3 *
```

```

preamble for the JOB.SHOP system 'Example 6
importing the RESOURCE subsystem

begin class MACHINE

    every MACHINE
        is a RESOURCE,
        has a NAME,
            a STREAM,
        and a USE.UNIT method, and
        belongs to the SHOP

    define NAME as a text variable
    define STREAM as an integer variable

    define USE.UNIT as a method
        given a real argument    'mean time needed using a unit
        yielding a real argument 'time waiting for a unit

    accumulate AVG.BUSY as the mean of ACQUIRED.UNITS
    accumulate AVG.BACKLOG as the mean,
        MAX.BACKLOG as the maximum of N.QUEUE

    the class
        has a LOOKUP method,
            a READ.SHOP method,
        and a PRINT.STATISTICS method, and
        owns the SHOP

    define LOOKUP as a MACHINE reference method
        given a text argument 'machine name
    end

begin class JOB

    every JOB
        has a NAME,
            a PROBABILITY,
            a DELAY.TIME,
        and a PERFORM process method, and
        owns a TASK'SEQUENCE

    define NAME as a text variable
    define PROBABILITY, DELAY.TIME as real variables
    tally NO.COMPLETED as the number,
        AVG.DELAY as the mean of DELAY.TIME

    the class
        has a NO.OF.JOBS,
            a REPERTOIRE,
            a SELECTION random step variable,
            a GENERATOR process method,
            a READ.REPERTOIRE method,
            a READ.PROBABILITIES method,
            a PRINT.REPERTOIRE method,
            a PRINT.PROBABILITIES method,
        and a PRINT.STATISTICS method

    define NO.OF.JOBS as an integer variable
    define REPERTOIRE as a 1-dim JOB reference array
    define SELECTION as an integer, stream 9 variable

```

```

    define GENERATOR as a process method
      given 2 real arguments 'mean inter-arrival time and stop time

end

begin class TASK

  every TASK
    has a MACHINE
    and a MEAN.TIME, and
    belongs to a SEQUENCE

    define MACHINE as a MACHINE reference variable
    define MEAN.TIME as a real variable

end

define HOURS to mean units

end

methods for the MACHINE class

method USE.UNIT given MEAN.TIME yielding TIME.WAITED

  if AVAILABLE.UNITS = 0
    define START.TIME as a real variable
    START.TIME = TIME.V
    call WAIT.FOR(1, 0)
    TIME.WAITED = TIME.V - START.TIME
  else
    add 1 to ACQUIRED.UNITS
  always

  work EXPONENTIAL.F(MEAN.TIME, STREAM) HOURS

  subtract 1 from ACQUIRED.UNITS

end

method LOOKUP(MACHINE.NAME)

  define MACHINE as a MACHINE reference variable

  for each MACHINE in SHOP with NAME(MACHINE) = MACHINE.NAME
    find the first case
  if found
    return with MACHINE
  otherwise

  return with 0

end

```

```

method READ.SHOP

    define NO.OF.MACHINES, I as integer variables
    define MACHINE as a MACHINE reference variable

    read NO.OF.MACHINES
    for I = 1 to NO.OF.MACHINES
    do
        create MACHINE
        read TOTAL.UNITS(MACHINE), NAME(MACHINE)
        STREAM(MACHINE) = I
        file MACHINE in SHOP
    loop

end

method PRINT.STATISTICS
    define MACHINE as a MACHINE reference variable

    print 5 lines thus

    DEPARTMENT INFORMATION

NAME                NO.OF MACHINES    UTILIZATION    AVG. NO. OF JOBS    MAXIMUM
                   NO. OF MACHINES    IN BACKLOG    IN BACKLOG          BACKLOG

    for each MACHINE in SHOP
        print 1 line with NAME(MACHINE), TOTAL.UNITS(MACHINE),
        AVG.BUSY(MACHINE) / TOTAL.UNITS(MACHINE),
        AVG.BACKLOG(MACHINE), MAX.BACKLOG(MACHINE) thus
*****            *                *.*          *.*          *
end

methods for the JOB class

process method PERFORM

    define TASK as a TASK reference variable
    define TOTAL.WAIT, WAIT as real variables

    for each TASK in SEQUENCE
    do
        call USE.UNIT(MACHINE(TASK)) given MEAN.TIME(TASK) yielding WAIT
        add WAIT to TOTAL.WAIT
    loop

    DELAY.TIME = TOTAL.WAIT

end

process method GENERATOR(MEAN.INTERARRIVAL.TIME, STOP.TIME)

    until TIME.V >= STOP.TIME
    do
        schedule a PERFORM(REPERTOIRE(SELECTION)) now
        wait EXPONENTIAL.F(MEAN.INTERARRIVAL.TIME, 10) HOURS
    loop

    call PRINT.STATISTICS

```

```

    ''stop
end

method READ.REPERTOIRE

    define I as an integer variable
    define JOB as a JOB reference variable
    define TASK as a TASK reference variable
    define MACHINE.NAME as a text variable

    read NO.OF.JOBS
    reserve REPERTOIRE as NO.OF.JOBS

    for I = 1 to NO.OF.JOBS
    do
        create JOB
        read NAME(JOB)
        until mode is alpha
        do
            create TASK
            read MEAN.TIME(TASK), MACHINE.NAME
            MACHINE(TASK) = MACHINE'LOOKUP(MACHINE.NAME)
            if MACHINE(TASK) = 0
                print 1 line with MACHINE.NAME, NAME(JOB) thus
TASK ***** FOR JOB TYPE ***** IS NOT DEFINED
                destroy TASK
            else
                file TASK in SEQUENCE(JOB)
            always
        loop
        REPERTOIRE(I) = JOB
    loop

    start new input line
end

method READ.PROBABILITIES

    define I, J as integer variables

    for I = 1 to NO.OF.JOBS
        read PROBABILITY(REPERTOIRE(I)), J

    read as B 1 ''to reread the current input line
    read SELECTION

end

method PRINT.REPERTOIRE
define I as an integer variable

    define JOB as a JOB reference variable
    define TASK as a TASK reference variable

    print 2 lines thus

    THE JOB TYPE DESCRIPTIONS

```

```

for I = 1 to NO.OF.JOBS
do
  JOB = REPERTOIRE(I)
  print 3 lines with NAME(JOB) thus
  JOB NAME *****
          TASK SEQUENCE
          MACHINE           MEAN TIME
  for each TASK in SEQUENCE(JOB)
    print 1 line with NAME(MACHINE(TASK)), MEAN.TIME(TASK) thus
          *****           *.**
  loop
end

method PRINT.PROBABILITIES

define I as an integer variable
define JOB as a JOB reference variable

print 3 lines thus

THE JOBS WERE DISTRIBUTED AS FOLLOWS:
  NAME           PROBABILITY

for I = 1 to NO.OF.JOBS
do
  JOB = REPERTOIRE(I)
  print 1 line with NAME(JOB), PROBABILITY(JOB) thus
  *****           *.***
loop
end

method PRINT.STATISTICS

define I as an integer variable
define JOB as a JOB reference variable

print 4 lines with TIME.V thus

RESULTS AFTER      *.** HOURS OF CONTINUOUS OPERATION
JOB TYPE           NO. COMPLETED      AVERAGE DELAY
                  (HOURS)

for I = 1 to NO.OF.JOBS
do
  JOB = REPERTOIRE(I)
  print 1 line with NAME(JOB), NO.COMPLETED(JOB), AVG.DELAY(JOB) thus
  *****           *           *.**
loop

call MACHINE'PRINT.STATISTICS

end

```

main

```

define MEAN.INTERARRIVAL.TIME, STOP.TIME as real variables

open unit 1 for input, name is "ex6.dat"
use unit 1 for input

call MACHINE'READ.SHOP
call JOB'READ.REPERTOIRE
read MEAN.INTERARRIVAL.TIME, STOP.TIME
call JOB'READ.PROBABILITIES

print 1 line thus
    E X A M P L E   J O B   S H O P   S I M U L A T I O N

call JOB'PRINT.REPERTOIRE
call JOB'PRINT.PROBABILITIES

schedule a JOB'GENERATOR(MEAN.INTERARRIVAL.TIME, STOP.TIME) now
start simulation

read as / using unit 5 '' keep text window open
end

```

public preamble for the RESOURCE subsystem

```

begin class RESOURCE

    every RESOURCE
        has a TOTAL.UNITS,
            an ACQUIRED.UNITS,
            an AVAILABLE.UNITS method,
            a WAIT.FOR method,
        and a CLEAN.UP method, and
        owns a REQUEST'QUEUE

        define TOTAL.UNITS      as an integer variable
        define ACQUIRED.UNITS   as an integer variable monitored on the left
        define AVAILABLE.UNITS  as an integer method
        define WAIT.FOR         as a method
            given 2 integer values 'requested units and priority
        before destroying a RESOURCE, call CLEAN.UP

    end

begin class REQUEST

    every REQUEST
        has a UNITS,
            a PRIORITY,
        and a PROCESS.NOTICE, and
        belongs to a QUEUE

        define UNITS, PRIORITY as integer variables
        define PROCESS.NOTICE  as a pointer variable
        define QUEUE as a set ranked by high PRIORITY

    end

end

end

```

```

methods for the RESOURCE class

left method ACQUIRED.UNITS

  define ACQ as an integer variable
  define REQ as a REQUEST reference variable

  enter with ACQ

  while QUEUE is not empty and UNITS(F.QUEUE) <= TOTAL.UNITS - ACQ
  do
    remove first REQ from QUEUE
    add UNITS(REQ) to ACQ
    schedule the PROCESS.NOTICE(REQ) now
    destroy REQ
  loop

  move from ACQ

end

method AVAILABLE.UNITS

  return with TOTAL.UNITS - ACQUIRED.UNITS

end

method WAIT.FOR(REQ.UNITS, REQ.PRIORITY)

  define REQ as a REQUEST reference variable

  create REQ
  UNITS(REQ) = REQ.UNITS
  PRIORITY(REQ) = REQ.PRIORITY
  PROCESS.NOTICE(REQ) = PROCESS.V
  file REQ in QUEUE
  suspend

end

method CLEAN.UP

  define REQ as a REQUEST reference variable

  while QUEUE is not empty
  do
    remove first REQ from QUEUE
    destroy PROCESS.NOTICE(REQ)
    destroy REQ
  loop

end

```

6.07 Example 7 - A Computer Center Study

This example models a computer center in which jobs are processed on a priority basis according to a priority chosen by the user. Higher-priority jobs are processed sooner, but at a premium in cost. Small interactive jobs arrive very frequently and have a short execution time. These will be modeled as internally generated according to the assumptions of a Poisson arrival process and exponential service time (truncated to the pre-specified limit on small interactive jobs). Larger jobs will be completely specified through data.

Let us assume that contention for storage is also a problem, and that each job requires a pre-specified amount of storage. If the storage is available, the program can run immediately. If not, the program will be en-queued according to its priority. For multiple jobs with the same priority, the amount of storage required will be used to break the tie (i.e., smaller jobs run sooner)

Small jobs will be generated by a JOB.GENERATOR process method. Large jobs will be represented as external processes. The CPU and the storage will be modeled as resources.

The model will run for an arbitrary period of time and stop abruptly to produce the desired measurements, which are:

- Number of jobs processed
- Average job dwell time
- Utilization of each resource
- Average queue for each resource.

Input data are read from ex7.dat, external processes are invoked from the file ex7_x.dat.

Input data in file ex7.dat

```
1
6
2.0
0.8
12.0
```

Input data in file ex7_x.dat

```
JOB 1.00 3 1 5.00 *
JOB 2.46 1 2 7.00 *
JOB 3.78 3 3 10.00 *
JOB 9.28 2 2 30.00 *
JOB 10.48 1 4 40.00 *
JOB 24.22 1 5 60.00 *
```

```

preamble for the COMPUTER.CENTER system 'Example 7
importing the RESOURCE subsystem

begin class COMPUTER

  the class
    has a CPU,
      a MEMORY,
      a JOB.TIME, 'in minutes
      a JOB process method,
      a JOB.GENERATOR process method,
      and a STOP.SIMULATION process method

  define CPU, MEMORY as COMPUTER.RESOURCE reference variables

  define JOB.TIME as a real variable
  tally NO.PROCESSED as the number,
      AVG.JOB.TIME as the average of JOB.TIME

  define JOB as a process method
    given 2 integer values 'priority, required units of memory,
      and 1 real value      'processing time in minutes

  define JOB.GENERATOR as a process method
    given 3 real values    'mean interarrival time in minutes,
                          'mean processing time in minutes,
                          'stop time

end

begin class COMPUTER.RESOURCE

  every COMPUTER.RESOURCE
    is a RESOURCE and
    has a UTILIZATION method

  define UTILIZATION as a double method

  accumulate AVG.USED as the average of ACQUIRED.UNITS
  accumulate AVG.QLEN as the average,
      MAX.QLEN as the maximum of N.QUEUE

end

processes include JOB
external process is JOB
external process unit is 7

end

```

methods for the COMPUTER class

process method JOB(JOB.PRIORITY, MEMORY.REQUIREMENT, PROCESSING.TIME)

define START.TIME as a real variable

START.TIME = TIME.V

if AVAILABLE.UNITS(MEMORY) >= MEMORY.REQUIREMENT and
(QUEUE(MEMORY) is empty or PRIORITY(F.QUEUE(MEMORY)) < JOB.PRIORITY)
add MEMORY.REQUIREMENT to ACQUIRED.UNITS(MEMORY)

else
call WAIT.FOR(MEMORY)(MEMORY.REQUIREMENT, JOB.PRIORITY)
always

if AVAILABLE.UNITS(CPU) > 0
add 1 to ACQUIRED.UNITS(CPU)
else
call WAIT.FOR(CPU)(1, JOB.PRIORITY)
always

work PROCESSING.TIME minutes

subtract MEMORY.REQUIREMENT from ACQUIRED.UNITS(MEMORY)
subtract 1 from ACQUIRED.UNITS(CPU)

JOB.TIME = (TIME.V - START.TIME) * MINUTES.V

end

process method JOB.GENERATOR

given MEAN.INTERARRIVAL.TIME, MEAN.PROC.TIME, STOP.TIME

until TIME.V >= STOP.TIME
do

schedule a JOB
given RANDI.F(1, 10, 1), RANDI.F(1, TOTAL.UNITS(MEMORY), 2),
MIN.F(EXPONENTIAL.F(MEAN.PROC.TIME, 4), 2 * MEAN.PROC.TIME) now
wait EXPONENTIAL.F(MEAN.INTERARRIVAL.TIME, 3) minutes

loop

end

process method STOP.SIMULATION

skip 6 lines

print 9 lines with TIME.V, UTILIZATION(CPU), UTILIZATION(MEMORY),
AVG.QLEN(MEMORY), MAX.QLEN(MEMORY), AVG.QLEN(CPU), MAX.QLEN(CPU),
NO.PROCESSED, AVG.JOB.TIME thus

A F T E R **.** HOURS

THE CPU UTILIZATION WAS * .** %

THE MEMORY UTILIZATION WAS * .** %

THE AVG QUEUE FOR MEMORY WAS * .** JOBS

THE MAX QUEUE FOR MEMORY WAS * .** JOBS

THE AVG QUEUE FOR A CPU WAS * .** JOBS

THE MAX QUEUE FOR A CPU WAS * .** JOBS

THE TOTAL NUMBER OF JOBS COMPLETED WAS ***

WITH AN AVERAGE PROCESSING TIME OF * .** MINUTES

' ' stop

end

```

method COMPUTER.RESOURCE'UTILIZATION

    return with 100 * AVG.USED / TOTAL.UNITS

end

process JOB 'scheduled externally

    define JOB.PRIORITY, MEMORY.REQUIREMENT as integer variables
    define PROCESSING.TIME as a real variable

    read JOB.PRIORITY, MEMORY.REQUIREMENT, PROCESSING.TIME
    call COMPUTER'JOB(JOB.PRIORITY, MEMORY.REQUIREMENT, PROCESSING.TIME)

end

main

    define MEAN.INTERARRIVAL.TIME, MEAN.PROCESSING.TIME, STOP.TIME
        as real variables

    open unit 7 for input, name is "ex7_x.dat"
    open unit 1 for input, name is "ex7.dat"
    use unit 1 for input

    create COMPUTER'CPU
    create COMPUTER'MEMORY
    read TOTAL.UNITS(COMPUTER'CPU), TOTAL.UNITS(COMPUTER'MEMORY),
        MEAN.INTERARRIVAL.TIME, MEAN.PROCESSING.TIME, STOP.TIME

    print 6 lines with TOTAL.UNITS(COMPUTER'CPU),
        TOTAL.UNITS(COMPUTER'MEMORY), 60 / MEAN.INTERARRIVAL.TIME,
        MEAN.PROCESSING.TIME, STOP.TIME thus
        A C O M P U T E R C E N T E R S T U D Y
        NO. OF CPU'S ** STORAGE AVAILABLE ****
        SMALL JOBS ARRIVE AT THE RATE OF *** / HOUR
        AND HAVE A MEAN PROCESSING TIME OF ***.*** MINUTES
        LARGE JOBS ARE SUPPLIED AS EXTERNAL DATA
        THE SIMULATION PERIOD IS **. ** HOURS

    HOURS.V = 1 'one hour per simulation time unit

    schedule a COMPUTER'JOB.GENERATOR
        given MEAN.INTERARRIVAL.TIME, MEAN.PROCESSING.TIME, STOP.TIME now
    schedule a COMPUTER'STOP.SIMULATION in STOP.TIME hours

    start simulation

    read as / using unit 5 ''to keep text window open

end

```

```

public preamble for the RESOURCE subsystem

begin class RESOURCE

  every RESOURCE
    has a TOTAL.UNITS,
        an ACQUIRED.UNITS,
        an AVAILABLE.UNITS method,
        a WAIT.FOR method,
    and a CLEAN.UP method, and
    owns a REQUEST'QUEUE

    define TOTAL.UNITS      as an integer variable
    define ACQUIRED.UNITS   as an integer variable monitored on the left
    define AVAILABLE.UNITS  as an integer method
    define WAIT.FOR        as a method
        given 2 integer values 'requested units and priority
    before destroying a RESOURCE, call CLEAN.UP

  end

begin class REQUEST

  every REQUEST
    has a UNITS,
        a PRIORITY,
    and a PROCESS.NOTICE, and
    belongs to a QUEUE

    define UNITS, PRIORITY as integer variables
    define PROCESS.NOTICE  as a pointer variable
    define QUEUE as a set ranked by high PRIORITY

  end

end

methods for the RESOURCE class

left method ACQUIRED.UNITS

  define ACQ as an integer variable
  define REQ as a REQUEST reference variable

  enter with ACQ

  while QUEUE is not empty and UNITS(F.QUEUE) <= TOTAL.UNITS - ACQ
  do
    remove first REQ from QUEUE
    add UNITS(REQ) to ACQ
    schedule the PROCESS.NOTICE(REQ) now
    destroy REQ
  loop

  move from ACQ
end

method AVAILABLE.UNITS

  return with TOTAL.UNITS - ACQUIRED.UNITS

end

```

```

method WAIT.FOR(REQ.UNITS, REQ.PRIORITY)

    define REQ as a REQUEST reference variable

    create REQ
    UNITS(REQ) = REQ.UNITS
    PRIORITY(REQ) = REQ.PRIORITY
    PROCESS.NOTICE(REQ) = PROCESS.V
    file REQ in QUEUE
    suspend

end

method CLEAN.UP

    define REQ as a REQUEST reference variable

    while QUEUE is not empty
    do
        remove first REQ from QUEUE
        destroy PROCESS.NOTICE(REQ)
        destroy REQ
    loop

end

```

All example programs from this manual are in the SIMSCRIPT sub-directory `sim3_examples`.