

---

---

# MODSIM III<sup>®</sup>

## The Language for Object-Oriented Programming

---

---

### Reference Manual

---

Title: (CACI)Logo.eps  
Creator: Adobe Illustrator 48(TM) 1.0.3  
CreationDate: (10/8/90) (9:11 AM)

## Products Company

3333 North Torrey Pines Court, La Jolla, California 92037 • (619) 824.5200 • Fax (619) 457-1184  
Watchmoor Park, Riverside Way, Camberley, Surrey GU15 3YL, UK • 1276 671 671 • Fax 1276 670 677  
1600 Wilson Blvd., 13th Floor, Arlington, Virginia 22209 • (703) 875-2900 • Fax (703) 875-2904

---

---

Copyright © 1996 CACI Products Co.  
December 1996

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:**

**In the US and Pacific Rim:**

CACI Products Company  
3333 North Torrey Pines Court  
La Jolla, California 92037  
Phone: (619) 824.5200  
Fax: (619) 457-1184

**In Europe:**

CACI Products Division  
Watchmoor Park  
Riverside Way  
Camberley, Surrey  
GU15 3YL, UK  
Phone: 1276 671 671  
Fax: 1276 670677

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.

SIMGRAPHICS II and MODSIM III are registered trademarks of CACI Products Company.

# Contents

---

<b>FIGURES.....</b>	<b>ix</b>
<b>PREFACE.....</b>	<b>a</b>
WHAT IS MODSIM III?.....	a
MODSIM III DOCUMENTATION.....	a
FREE TRIAL & TRAINING.....	b
<b>1. INTRODUCTION .....</b>	<b>1</b>
1.1 OVERVIEW OF MODSIM III .....	2
1.2 OBJECT-ORIENTED FEATURES.....	4
1.3 DISCRETE-EVENT SIMULATION FEATURES.....	4
1.4 MODULES .....	5
1.5 THE MODSIM DEVELOPMENT ENVIRONMENT .....	5
<b>SECTION I. MODSIM III - SYNTAX AND STRUCTURE.....</b>	<b>7</b>
<b>2. STRUCTURE OF MODSIM III PROGRAMS.....</b>	<b>9</b>
2.1 PROGRAM LAYOUT .....	9
2.2 IDENTIFIERS, RESERVED WORDS AND STANDARD PROCEDURES.....	11
2.3 BLOCK STRUCTURE AND SCOPE OF VARIABLES.....	13
2.4 NESTING OF BLOCKS.....	14
2.5 REDEFINITION OF IDENTIFIERS.....	15
2.6 DELIMITERS .....	15
2.7 SEPARATORS .....	16
2.8 COMMENTS.....	17
<b>3. SIMPLE DATA TYPES AND THEIR OPERATORS.....</b>	<b>19</b>
3.1 WHAT IS A DATA TYPE?.....	20
3.2 INTERNAL REPRESENTATION OF DATA.....	20
3.2.1 Representation of Numerical Data.....	21
3.2.1.1 Type INTEGER.....	21
3.2.1.2 Type REAL .....	21
3.2.2 Representation of Textual Data.....	22
3.2.2.1 Type CHAR .....	22
3.2.2.2 Type STRING .....	23
3.2.3 Representation of TRUE / FALSE or Boolean Data.....	23
3.2.3.1 Type BOOLEAN .....	24
3.3 USER-DEFINED TYPES.....	24
3.3.1 Enumerated Types.....	24
3.3.2 Ordinal Data Types.....	25
3.3.3 Subrange Types .....	25
3.4 EXTERNAL REPRESENTATION OF DATA.....	26
3.4.1 INTEGER Literals .....	26
3.4.2 REAL Literals.....	27
3.4.3 CHARACTER Literals .....	28
3.4.4 STRING Literals.....	28
3.4.5 BOOLEAN Literals.....	29

3.4.6 Enumerations .....	29
3.5 OPERATORS .....	29
3.5.1 Assignment Operator .....	30
3.5.2 Arithmetic Operators .....	30
3.5.3 Relational Operators .....	31
3.5.4 Logical Operators .....	32
3.6 BUILT-IN PROCEDURES AND FUNCTIONS .....	32
<b>4. DECLARATIONS, EXPRESSIONS AND PRECEDENCE .....</b>	<b>35</b>
4.1 DECLARATIONS .....	35
4.1.1 CONSTant Declarations .....	35
4.1.2 TYPE Declarations .....	36
4.1.3 VARIable Declarations .....	37
4.1.4 PROCEDURE Declarations .....	38
4.1.5 PROCEDURE Variable Declarations .....	38
4.2 AUTOMATIC INITIALIZATION OF VARIABLES .....	38
4.3 EXPRESSIONS .....	39
4.4 OPERATOR PRECEDENCE .....	39
4.5 TYPES OF EXPRESSIONS .....	40
4.5.1 Evaluating Boolean expressions .....	40
<b>5. STRUCTURED DATA TYPES .....</b>	<b>43</b>
5.1 USING STRUCTURED DATA TYPES .....	43
5.1.1 Dynamic Versus Fixed Structures .....	46
5.2 MEMORY MANAGEMENT OF DYNAMIC DATA STRUCTURES .....	47
5.2.1 The CLONE Function .....	50
5.2.2 Orphaned Data .....	51
5.2.3 The DISPOSE Procedure .....	51
5.2.4 Hanging References .....	51
5.3 RECORDS .....	52
5.3.1 Using NEW to Allocate RECORDs .....	52
5.3.2 ANYREC, ANYOBJ and NILOBJ .....	55
5.3.3 Operations on RECORDs .....	56
5.4 ARRAYS .....	56
5.4.1 Operations on ARRAYS .....	59
5.4.2 Using the NEW Procedure to Allocate an ARRAY .....	59
5.4.3 Ragged ARRAYS .....	60
5.4.4 The HIGH and LOW Functions .....	61
5.5 OBJECTS .....	62
5.6 DECLARATIONS REVISITED .....	62
5.6.1 Anonymous Types .....	63
5.7 FIXED DATA STRUCTURES .....	63
5.7.1 The FIXED RECORD Type .....	64
5.7.1.1 Declaring FIXED RECORD Types .....	64
5.7.2 The FIXED ARRAY Type .....	64
5.7.2.1 Declaring FIXED ARRAY Types .....	65
5.8 REFERENCING THE ARRAY AND RECORD .....	66
<b>6. STATEMENTS AND TYPE COMPATIBILITY .....</b>	<b>69</b>
6.1 TYPE COMPATIBILITY .....	70
6.1.1 Type Conversion .....	71
6.2 THE ASSIGNMENT STATEMENT .....	73
6.3 PROGRAM FLOW CONTROL .....	73
6.4 THE IF STATEMENT .....	73

6.4.1 Comparing REAL Values in a Boolean Expression .....	74
6.5 THE CASE STATEMENT .....	76
6.6 ITERATIVE STATEMENTS .....	77
6.6.1 The WHILE Statement .....	77
6.6.2 The REPEAT Statement .....	78
6.6.3 The FOR Statement.....	78
6.6.4 The FOREACH Statement .....	79
6.6.5 The EXIT Statement .....	81
6.6.6 The LOOP Statement .....	81
6.6.7 The Other Control Statements .....	81
<b>7. PROCEDURES AND FUNCTIONS.....</b>	<b>83</b>
7.1 FORMAL PARAMETER QUALIFIERS: IN, OUT, INOUT.....	84
7.2 INVOKING PROCEDURES.....	85
7.3 DECLARING PROCEDURES .....	86
7.4 RETURN STATEMENT.....	86
7.5 THE FORWARD QUALIFIER .....	87
7.6 PROCEDURES WITH EMPTY PARAMETER LISTS .....	88
<b>8. MODULES.....</b>	<b>89</b>
8.1 FACTS ABOUT MODULES .....	89
8.2 THE IMPORT STATEMENT.....	90
8.3 MAIN MODULE .....	92
8.4 DEFINITION MODULE .....	92
8.4.1 Cycle Dependencies .....	93
8.5 IMPLEMENTATION MODULE.....	93
8.6 THE MODINIT PROCEDURE .....	94
8.7 FILE NAMING CONVENTIONS FOR MODULES.....	94
8.8 INCLUDING C/C++ CODE IN A MODSIM PROGRAM.....	95
<b>SECTION II. OBJECT-ORIENTED PROGRAMMING.....</b>	<b>99</b>
<b>9. OBJECTS IN MODSIM III.....</b>	<b>101</b>
9.1 OBJECT TYPE VERSUS OBJECT INSTANCE.....	102
9.2 SCOPE OF AN OBJECT'S FIELDS .....	102
9.3 OBJECT TYPE DECLARATION / OBJECT DECLARATION.....	102
9.4 METHOD DECLARATIONS .....	104
9.5 SCOPE OF FIELDS AND VARIABLES IN OBJECTS .....	105
9.6 OBJECT REFERENCE VARIABLES .....	105
9.7 CLASS VARIABLES (FIELDS) AND METHODS .....	107
9.8 OBJECT TYPE CHECKING AND THE ANYOBJ TYPE.....	108
9.9 ALLOCATING AND DEALLOCATING OBJECTS.....	109
9.10 OBJINIT & OBJTERMINATE .....	110
9.11 OBJCLONE .....	110
9.12 PROTO OBJECTS.....	111
<b>10. METHODS AND FIELDS OF OBJECTS .....</b>	<b>115</b>
10.1 INVOKING AN OBJECT'S ASK AND TELL METHODS .....	115
10.2 BUILT-IN REFERENCE CONSTANT SELF .....	118
10.3 REFERENCING AN OBJECT'S FIELDS.....	118
10.3.1 MONITORING OF FIELDS OR VARIABLES .....	120
10.4.1 Example of Static Monitoring.....	121
10.4.2 Defining Monitoring Objects .....	121

10.4.3	Syntax .....	121
10.4.4	Semantics .....	122
10.5	IMPLEMENTATION FEATURES FOR MONITOR METHODS .....	122
10.5.1	Syntax .....	122
10.5.2	Semantics .....	122
10.6	ATTACHING A MONITOR OBJECT TO A VARIABLE OR FIELD .....	123
10.6.1	Syntax for Simple Fields .....	123
10.6.2	Syntax for Monitor Types .....	123
10.6.3	Semantics .....	123
10.6.4	Dynamic Monitors .....	124
<b>11.</b>	<b>INHERITANCE .....</b>	<b>125</b>
11.1	HIERARCHICAL OBJECT TYPES .....	125
11.2	COERCION OF OBJECTS .....	127
11.3	OBJECT INHERITANCE .....	128
11.4	OVERRIDING METHODS .....	129
11.5	EXTENDING INHERITED BEHAVIORS .....	130
11.5.1	Overriding the ObjInit Method .....	131
11.6	MULTIPLE INHERITANCE .....	132
11.6.1	Declaring Multiple Base Types .....	132
11.7	RESOLVING CONFLICTING FIELD NAMES .....	132
11.8	RESOLVING CONFLICTING METHOD NAMES .....	133
11.8.1	Combining Multiple Inherited Methods .....	135
11.8.2	Overriding the ObjInit Method in Multiple Inheritance .....	135
11.9	CONFLICTING FIELD AND METHOD NAMES .....	136
<b>12.</b>	<b>DATA HIDING AND DATA SHARING .....</b>	<b>137</b>
12.1	PRIVATE FIELDS AND METHODS .....	137
	<b>SECTION III. SIMULATION .....</b>	<b>139</b>
<b>13.</b>	<b>PROCESS-BASED DISCRETE-EVENT SIMULATION .....</b>	<b>141</b>
13.1	SIMULATION TIME .....	141
13.2	THE SYSTEM'S PENDING LIST - OBJECTS' ACTIVITY LISTS .....	142
13.3	PROCESS-ORIENTED VS EVENT-ORIENTED SIMULATION .....	142
13.4	TIME ELAPSING METHODS - THE WAIT STATEMENT .....	144
13.4.1	THE WAIT STATEMENT .....	144
13.5	THE ASYNCHRONOUS TELL AND WAITFOR CALLS .....	145
13.6	SYNCHRONIZING ACTIVITIES .....	148
13.6.1	The Terminate Statement .....	149
13.7	ARBITRARY SYNCHRONIZATION WITH TRIGGER OBJs .....	150
13.8	MULTIPLE PROCESS ACTIVITIES .....	151
13.9	ACTIVITY TIE-BREAKING .....	151
13.10	INTERRUPTING ACTIVITIES .....	153
13.10.1	Interrupting Methods and ACTID .....	154
<b>14.</b>	<b>GROUPING OBJECTS .....</b>	<b>157</b>
14.1	USING GROUP OBJECTS .....	157
14.2	THE QUEUE GROUP .....	158
14.3	THE STACK GROUP .....	159
14.4	THE RANKED GROUP .....	159
14.5	STATISTICAL GROUPS .....	160
14.6	ITERATING THROUGH A GROUP .....	161

<b>15. STATISTICAL DISTRIBUTIONS: RANDOMOBJ .....</b>	<b>163</b>
<b>16. RESOURCE OBJECTS .....</b>	<b>167</b>
16.1 ACQUIRING RESOURCES.....	167
16.1.1 Difference Between Requesting Methods.....	168
16.2 CHANGING THE SET OF RESOURCES.....	169
16.3 STATISTICS OF RESOURCES.....	170
<b>SECTION IV. INPUT/OUTPUT .....</b>	<b>171</b>
<b>17. INPUT / OUTPUT .....</b>	<b>173</b>
17.1 INPUT & OUTPUT STATEMENTS.....	173
17.2 STREAM I/O USING STREAMOBJ.....	175
17.3 ASK METHODS OF STREAMOBJ.....	175
17.4 PROCEDURES OF IOMOD.....	176
<b>18. GRAPHICS AND ANIMATION.....</b>	<b>179</b>
<b>APPENDICES.....</b>	<b>181</b>
<b>APPENDIX A. GLOSSARY .....</b>	<b>183</b>
<b>APPENDIX B. RESERVED WORDS .....</b>	<b>187</b>
<b>APPENDIX C. BUILT-IN PROCEDURES .....</b>	<b>205</b>
<b>APPENDIX D. STANDARD LIBRARY MODULES .....</b>	<b>213</b>
D.1 MODULE NAME: DEBUG.....	214
D.2 MODULE NAME: GRPMOD.....	216
D.3 MODULE NAME: IOMOD.....	218
D.4 MODULE NAME: LISTMOD.....	220
D.5 MODULE NAME: MATHMOD.....	221
D.6 MODULE NAME: OSMOD.....	224
D.7 MODULE NAME: RANDMOD.....	234
D.8 MODULE NAME: RESMOD.....	235
D.9 MODULE NAME: SIMMOD.....	236
D.10 MODULE NAME: STATMOD.....	240
D.11 MODULE NAME: UTILMOD.....	242
D.12 MODULE NAME: VERSION.....	245
<b>APPENDIX E. OBJECTS.....</b>	<b>247</b>
<b>INDEX.....</b>	<b>339</b>



# Figures

---

Figure 2-1. Syntax of an Identifier.....	12
Figure 2-2. Syntax of a Program Block .....	13
Figure 2-3. Delimiters .....	16
Figure 3-1. Simple Data Types .....	19
Figure 3-2. Syntax of an INTEGER Literal .....	26
Figure 3-3. Syntax of REAL Literals.....	27
Figure 3-4. Character Literals .....	28
Figure 3-5. Syntax of String Literals.....	29
Figure 3-6. Arithmetic Operators .....	31
Figure 3-7. Relational Operators.....	31
Figure 3-8. Logical Operators .....	32
Figure 4-1. Syntax of a Constant Declaration .....	36
Figure 4-2. Syntax of a Type Declaration.....	37
Figure 4-3. Operator Precedence .....	40
Figure 4-4. Short Circuit Logic.....	41
Figure 5-1. Memory Before Assignments.....	49
Figure 5-2. Memory After Assignments .....	49
Figure 5-3. Memory Before CLONE & Assignment .....	50
Figure 5-4. Memory After CLONE & Assignment .....	51
Figure 5-5. Linked List of RECORDs .....	54
Figure 5-6. Syntax of an Array Type Declaration .....	57
Figure 5-7. An Array.....	58
Figure 5-8. A Ragged Array.....	61
Figure 5-9. FIXED ARRAY Type Declaration .....	65
Figure 6-1. Type Conversion Procedures / Functions .....	72
Figure 6-2. Examples of Type Conversions .....	72
Figure 6-3. Syntax of the IF...END IF Statement. ....	74
Figure 6-4: Syntax of the CASE .. END CASE Statement.....	76
Figure 6-5. Syntax of the WHILE .. END WHILE Statement.....	77
Figure 6-6. Syntax of the REPEAT...UNTIL Statement .....	78
Figure 6-7. Syntax of the FOR ... END FOR Statement.....	79
Figure 6-8. The FOREACH Statement .....	80
Figure 7-1. Syntax of a Procedure Declaration .....	86
Figure 7-2. A Right Triangle .....	87
Figure 7-3. Syntax of the Procedure Block .....	87
Figure 7-4. Empty Parameter Lists .....	88
Figure 8-1. Syntax of an IMPORT Statement .....	91
Figure 8-2. Syntax of a MAIN Module.....	92
Figure 8-3. Syntax of a DEFINITION Module.....	93
Figure 8-4. Syntax of an IMPLEMENTATION Module .....	94
Figure 8-5. File Naming Conventions for Modules .....	95
Figure 8-6. MODSIM Types vs C/C++ Types .....	96
Figure 9-1. Syntax of an Object Type Declaration.....	103
Figure 9-2. Syntax of an Object Declaration .....	104
Figure 9-3. Syntax for Substituting a Replaceable Type.....	111
Figure 9-4. Syntax for 'inherit spec'.....	112
Figure 10-1. Method Invocation .....	115
Figure 10-2. Syntax of the ASK Statement .....	116
Figure 10-3. Syntax of the TELL Statement.....	117
Figure 10-4. Syntax for a Monitor Object Inherited from a Monitor Object .....	121
Figure 10-5. Syntax for Declaring a Monitor Type .....	122

Figure 10-6. Syntax for Simple Fields .....	123
Figure 10-7. Syntax for Monitor Types .....	123
Figure 11-1. Object Type Hierarchy .....	126
Figure 11-2. Multiple-path Inheritance .....	132
Figure 11-3. Common Ancestor .....	134
Figure 13-1. The Pending List.....	142
Figure 13-2. Syntax of the WAIT Statement.....	145
Figure 13-3. Syntax of the TELL call .....	146
Figure 14-1. Built-in Groups.....	157

# Preface

---

## What Is MODSIM III?

MODSIM III is a general-purpose, modular, block-structured high-level programming language which provides direct support for object-oriented programming and discrete-event simulation. It can be used to build large process-based simulation models through modular and object-oriented development techniques.

MODSIM III is supported on a variety of machine architectures and operating systems. MODSIM III programs are highly portable from one machine to another.

## MODSIM III Documentation

These documents pertain to MODSIM III:

- *MODSIM III Reference Manual* - (This document) The language reference. Contains information about the syntax and structure of MODSIM III as a programming language. Also covers object-oriented programming, simulation, and I/O.
- *MODSIM III User's Manual* - Contains information about: release-specific features; use of the compilation manager; use of the MODSIM development environment, and MODSIM compiler options; and debugging MODSIM.
- *MODSIM III Tutorial* - Provides a broad overview of the language features of MODSIM and then concentrates on the object-oriented programming and simulation capabilities in MODSIM.
- *SIMGRAPHICS II User's Manual for MODSIM III* - Contains information about SIMGRAPHICS II, the companion to MODSIM III that provides easy access to presentation graphics and animation.

This manual is organized to give you a quick overview of each of MODSIM III's features, followed by a comprehensive discussion of each.

The first chapter of this *Reference Manual* contains several self-explanatory MODSIM III programs which illustrate the basic structure of the language and its use in simulation. The remainder of the manual is organized into four sections:

- I. **MODSIM III - Syntax & Structure:** Lexical structure, procedures, functions and flow of control. MODSIM III's automated compilation manager and other project management support facilities.

- II. **Object-Oriented Programming:** Objects, their fields and methods. Encapsulation and polymorphism. Inheritance and multiple inheritance.
- III. **Simulation:** Object-oriented, process-based, discrete-event simulation. A review of MODSIM III's facilities for simulation.
- IV. **Input / Output:** MODSIM III's stream oriented, random access and indexed I/O facilities. Formatted I/O. Object oriented I/O features.

## Appendices

- A. **Glossary**
- B. **Reserved Words**
- C. **Built-in Procedures**
- D. **Standard Library Modules**
- E. **Standard Library Objects**

## Index

## Free Trial & Training

MODSIM III is available exclusively from CACI Products Company. MODSIM III can be sent to your organization for a free trial. We provide everything needed for a complete evaluation on your computer: software, documentation, sample models, and immediate support when you need it.

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

**La Jolla, California**  
**Washington, D.C.**  
**London, United Kingdom**

For information on free trials or training, please contact any of our offices.

# 1. Introduction

---

MODSIM III is a modular, object-oriented, strongly typed, block-structured simulation language. This description touches on the several ways in which MODSIM III differs from traditional languages.

- **Modular:** MODSIM III programs may be (but are not required to be) divided into “modules”. Each module is stored in a separate file. The advantages of this approach are that these modules may be compiled separately, saving time when only one of them is edited, and that a single module may serve multiple programs. This is because modules can import constructs and definitions from each other. Readers familiar with Modula-2 and Ada will recognize this approach. The modular concept formalizes the notion of libraries of reusable code.

MODSIM III, itself, makes use of this feature. Many commonly-used features of the language, such as Input/Output and simulation, are described in “library modules” and are imported for use by a program.

MODSIM III modules may be compiled separately. This means that if you alter one module in a program, only that module needs to be recompiled. This is a powerful time-saving feature which greatly speeds program development and evolution. Separate compilation is distinct from independent compilation in that dependencies between modules are checked. As a result, any module which requires re-compilation as a result of edits to another module can be identified and also scheduled for compilation. The C/C++ language allows independent compilation of files, but offers no assistance in analyzing the effects of an editing change to one module. Even in Modula-2, it is the user's responsibility to assess the consequences of a change to one module.

- **Object-oriented.** An "object" is an encapsulation of a data record which describes the state of the object and procedures called methods which describe its behaviors. Objects are more concrete than most programming constructs. They interact through a clearly defined protocol and the fields of an object instance are private. A new object type can inherit the attributes of an existing object type and elaborate on the fields and methods of its ancestor type. Finally, objects are capable of polymorphism. A group of objects which share common ancestry can also share a method, yet each implements it differently. Thus, if we take a collection of objects which share Vehicle Object as their ancestor and ask each to refuel, the Car Object might take on unleaded gas, the Truck Object diesel fuel and the Mule Object would eat hay.
- **Strongly typed:** Every expression, assignment statement and parameter is type checked at compile time for consistency. This eliminates errors which can go un-

discovered until runtime in untyped languages. The concept of types also allows users to define their own types and to then declare variables of those types.

- **Block-structured:** Pascal, Modula-2 and Ada are examples of block-structured languages. A block is made up of declarations and executable statements. It may contain smaller blocks. The important feature of block-structured languages is that the scope or visibility of variables is restricted to the block in which they are declared and any subsidiary blocks. This control of scope of variables is fundamental to contemporary software engineering practices.
- **Simulation:** Simulation capabilities are provided both in library modules and directly through the language. These modules provide direct support for all capabilities needed to program discrete-event simulation models. All MODSIM III objects have the capability of using Process methods. A "Process" method is a method which can elapse simulation time. This is the meaning used throughout the manual. A process might WAIT in simulation time and interact at specific simulation times with other processes.

Each of these issues is discussed in greater depth in this *Reference Manual*. The topics of object-oriented programming and simulation are additionally covered in detail in the *MODSIM III Tutorial*.

## 1.1 Overview of MODSIM III

MODSIM is the result of evolutionary language development. It combines the best features which have emerged from contemporary programming language design and software engineering research and development. It serves as a complete development environment for large software projects.

Before proceeding into a detailed description of MODSIM's syntax and structure, it will be useful to provide a short sample of MODSIM code. The general structure and syntax should look familiar to anyone who has had contact with Algol, Pascal, Modula-2 or Ada. Indeed, anyone who has used a contemporary procedural language should have no problem understanding what this program does and how it works.

```

MAIN MODULE Sample1;

VAR
  sum, number : REAL;
  count       : INTEGER;

BEGIN
  OUTPUT("This program computes the average of a sequence
of");
  OUTPUT("positive numbers. Enter a sequence of numbers...");
  OUTPUT("Terminate the sequence with a negative number:");
  INPUT(number);

```

```

WHILE number >= 0.0
  INC(count); { increment the count }
  sum := sum + number;
  INPUT(number);
END WHILE;
IF count > 0
  OUTPUT(count, " numbers were entered");
  OUTPUT("Average is ", sum / FLOAT(count));
ELSE
  OUTPUT("Nothing was entered.");
END IF;
END MODULE.

```

We see from this code:

- MODSIM programs can consist of just one main module.
- All variables used in a MODSIM program or module must be declared by type.
- There are **INPUT** and **OUTPUT** statements to support simple, free-form I/O.
- Sequences of statements are delimited by the control and choice statements such as the **IF / END IF** instead of a **BEGIN / END** (Algol, Pascal) or **{ / }** (C).
- Control statements are symmetric... **IF / END IF, WHILE / END WHILE**.
- In mathematical expressions, as in Pascal and Ada, all type conversion is specified explicitly by the programmer.
- MODSIM has basic built-in types such as **REAL** and **INTEGER**.

Not apparent from this example, MODSIM also supports built-in **BOOLEAN**, **CHAR** and **STRING** types. In addition to the built-in scalar and **STRING** types, MODSIM supports the structured types **ARRAY** and **RECORD**, **OBJECT** types, subrange types, enumerated types and user defined types.

MODSIM III, like Pascal and Ada, is a strongly typed language. This means that expressions, assignments statements and parameters passed to procedures and methods are checked for type consistency. Inconsistent usage of variables is discovered and flagged at compile time. This leads to more reliable code and speedier development since errors are caught sooner. For example, if a procedure is expecting an **INTEGER** as an incoming argument and it is passed a **STRING**, the compiler will flag this as an error. If this were not caught at compile time, then, when the program was run, the **STRING** would be interpreted as if it were an **INTEGER** and the program would behave incorrectly. Errors such as this are often difficult to track down. Indeed, they may not show up in testing.

MODSIM III is a general-purpose, procedural programming language which can be used to write traditional style computer programs. But there is obviously more to it than that. Its distinction is as an object-oriented language and as a discrete-event **simulation** language. Finally, it is modular and provides support for large-scale software development.

## 1.2 Object-oriented Features

Objects are dynamically allocated data structures coupled with routines, called methods. The fields in the object's data structure define its state at any instant in time while its methods describe the actions which the object can perform. The values of the fields of an object are modified only by its own methods.

The utility of an object is that it is analogous to an object in the real world. It has a set of attributes, its fields, and a set of behaviors, its methods. There is a well-defined interface to each object type. The objects methods, or behaviors, are invoked by sending a message to the object. We can define new object types based on existing object types. Finally, we can give disparate objects a behavior, or method, with the same name. Each object, when asked to perform the behavior which goes with that name, can perform a unique behavior. This powerful concept, which is known as polymorphism, allows “generic” calls.

## 1.3 Discrete-event Simulation Features

All modern simulation languages support some construct for keeping track of simulation time and scheduling events relative to that simulation time. Simulation time is the clock which a simulation language uses to keep track of events and the ordering of these events.

In MODSIM III, simulation is supported by a library module which contains a number of objects and support procedures. All objects are allowed to perform actions which elapse simulation time. A method of one object might include a statement to WAIT until some future time before proceeding to the next statement, or it might send a message to another object so that the message arrives at that object at a specific simulation time.

The simulation paradigm supported by MODSIM is that of the **process**. The process is capable of carrying on multiple, concurrent **activities** each of which can elapse simulation time. The activities can operate autonomously or they can synchronize their operation. Any or all activities of a process can be interrupted, if necessary.

The process approach elaborates on the traditional technique of discrete-event simulation in a very important way. It allows a related group of activities to be coded in one routine. When it is necessary to elapse simulation time, the routine suspends execution until the stated amount of simulation time has elapsed and then the routine resumes execution. The traditional approach requires a separate routine for each event which can occur. This process-based view of simulation is similar to that supported by SIMSCRIPT II.5.

## 1.4 Modules

MODSIM III programs can be divided into library modules, each of which supports some particular functionality. Any module can import data constructs or procedures from other modules. A library module can be shared by many programs. New, more elaborate objects can be built from objects imported from library modules.

Any module can be compiled separately. This means that it is not necessary to re-compile a whole program after changes have been made to a module. Only affected modules need be re-compiled. This capability is known as separate compilation.

## 1.5 The MODSIM Development Environment

The MODSIM III development environment consists of a suite of tools which include a compilation manager, interactive debugger, graphics drawing package (SIMDRAW) and a comprehensive help system.

The MODSIM debugger is a fully functional interactive source level debugger which includes the ability to set break points, step through MODSIM code and browse MODSIM data structures. In addition, the debugger is aware of the simulation features of the MODSIM language.

SIMDRAW is an interactive graphics package which is used to create graphics files for use with SIMGRAPHICS II. SIMDRAW can be used to design the Graphical User Interface (GUI) to your MODSIM program, including specification of menus, dialog boxes, palettes, charts, graphs and images.

MODSIM III source and object code management is supported by a compilation manager. The compilation manager provides a variety of services to assist in the management of large and small projects. At the simplest level it manages the automatic compilation of programs. Given the name of a MAIN program module, the compilation manager will assess the status of each module which comprises the program. It will then compile each module which requires compilation because it has been edited or because it is dependent on another module which must be re-compiled. It links the resultant object modules and produces an executable.

The MODSIM development environment is fully documented in the *MODSIM III User's Manual*.



## **Section I. MODSIM III - Syntax and Structure**



## 2. Structure of MODSIM III Programs

---

In this chapter we will describe the general layout of a MODSIM III program. For simplicity, we will treat a MODSIM program as a single module contained in one file. Most large programs will consist of a number of library modules in separate files. However the discussion of library modules in MODSIM will be deferred until Chapter 7 to allow more elementary language concepts to be covered first.

By program structure we mean the layout of the component parts of a program such as type and variable declarations, procedure declarations and actual program code. Also implied in this topic is the concept of scope, locality and visibility of variables and the related concept of blocks.

Lexical components such as identifiers, literal constants and operators are the most basic parts of a program. The term which describes all of these components is **token**. A program consists of a sequence of tokens. As the compiler examines the tokens it finds that some are reserved words such as **IF** and **ELSE**, others are operators such as **\*** and **+**, some are literal constants such as **5** or **67.32**, some are delimiters such as **BEGIN** and **END** and, finally, some are separators such as the comma, space or semicolon.

Tokens are used to build statements. Statements are used to construct larger lexical components such as declaration blocks or procedure bodies. These larger lexical components are then used to build modules. Finally, a program is built from one or more modules.

In discussing program structure we will necessarily use terms which may not be completely familiar to readers who have no previous exposure to languages of this type such as Algol, Pascal, or Ada. If you fall into this category, it might be worth skipping ahead to Chapter 3 at this point to scan the information about types, variables, constants and literals.

Finally, MODSIM is a strongly typed language. This means that all variables which are used in a program must be declared by type and that variables passed into procedures are checked for type consistency. This reduces errors in user code and ensures that they are caught at compile time instead of showing up later as run-time errors which can be very hard to find.

### 2.1 Program Layout

Although large MODSIM programs are usually organized into a number of separate modules, the simplest MODSIM program can be completely self-contained in one **MAIN MODULE**.

Constant, type, variable and procedure declarations come at the beginning of the **MAIN MODULE** followed by the actual program code. The following sample of MODSIM code illustrates how a program is structured. MODSIM is a case sensitive language. Note

also that reserved words and built-in procedures in the language are all capitalized so they stand out from the user's identifiers.

```

MAIN MODULE Sample2;
{ This is a comment } (* So is this *)
{ Comments can be nested and can continue on
  as many lines as needed. { This is a nested
    comment. } }

{ declaration sections start here }
CONST { constant declarations }
  NumberOfTrucks = 45;

TYPE { user-defined type declarations }
  DayOfWeekType = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
  WorkDayType   = [Mon .. Fri];

VAR { variable declarations }
  sum, number : REAL;
  count       : INTEGER;
  WorkDay     : WorkDayType;

PROCEDURE foo(IN x : REAL); { procedure declaration }
BEGIN
  {.. procedure code here }
END PROCEDURE; { foo }

BEGIN { program "sample2" execution starts here
      i.e the "main" routine }
  ...
  foo(47.83);
  ...
END MODULE. { program "sample2" ends here }

```

There are sections for constant, type, variable and procedure declarations. These are all optional and appear only if the program requires them. In fact a simple program could have no declarations.

```

MAIN MODULE hello;
BEGIN
  OUTPUT("Hello, world!");
END MODULE.

```

Every module has a name. A MODSIM program takes its name from the **MAIN MODULE**. This one is called **hello**. The executable produced by the compiler will be called **hello**. Module names are case sensitive as are the names of executables.

There can be any number of constant, type, variable and procedure declaration sections. The declaration sections can appear in any order. The only restriction to the ordering, and it is an important one, is that declarations build one upon the other. In other words,

if a user-defined type is used to declare a variable, the type's declaration must precede its use in a variable declaration.

In general, MODSIM, like other languages of this type expects each thing it sees to have been previously defined. There are several areas in which this requirement has been relaxed as a convenience to the programmer, but the principle is important. If the compiler can not recognize something it sees, it draws attention to the potential error. This helps ensure that code is correct and error free by identifying errors at compile time.

Once all of the declarations have been made, a **BEGIN** statement marks the end of the declarations and the beginning of the program's executable code. In actual practice it is desirable to have all declarations grouped together to enhance program readability. Usually the declaration sections are organized in the following order:

```

CONST
TYPE
VAR
PROCEDURE

```

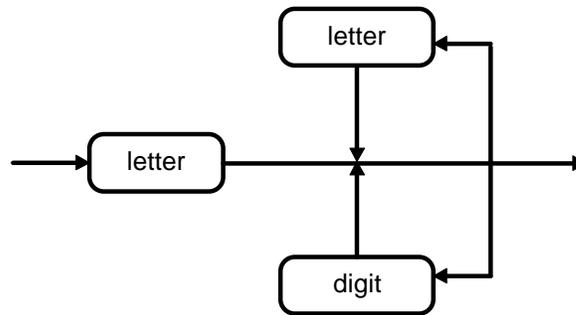
The reason for this ordering becomes obvious quickly. We use the constant definitions in declaring types, and then use the type definitions in declaring variables and finally use all of these pieces in coding the procedures. However, the language does not require this ordering provided everything is declared before it is referenced.

## 2.2 Identifiers, Reserved Words and Standard Procedures

Every statement in the language is comprised of a number of tokens. These include identifiers, literals, operators, delimiters and separators. Identifiers are names the programmer supplies to refer to such user-defined constructs as variables, types, constants, procedures, and modules.

Discussion of literals, operators, delimiters and separators will be deferred, but because they are so basic, identifiers are introduced here.

An identifier must begin with an alphabetic character optionally followed by any number of alphabetic or numeric characters. MODSIM is a case-sensitive language, so upper and lower case letters are distinct.



**Figure 2-1. Syntax of an Identifier**

Following are examples of identifiers:

`numberOfTrucks, A32, a32, x, count, Count`

**Note:** The identifiers `count` and `Count` are different because MODSIM is case sensitive.

Here are a few **illegal** identifiers:

`A_32, 3beanSalad, $JCL, Improvement%, zot.com`

Identifiers are used to name the following elements of MODSIM programs:

**Modules, Procedures, Constants, Types, Variables, Enumerations, Records, Record Fields, Objects, Object Fields, Object Methods**

The definitions of these language elements will be covered later in this manual but the common thread is that identifiers are used to name each of these constructs.

There is a simple rule about the use of identifiers:

**Identifiers must be unique within a scope.**

If the programmer has used an identifier called `numberOfTrucks` to name a procedure, the same identifier cannot be used in the same scope for another purpose, for instance, to name a variable. However, when the concept of scope is discussed shortly, we will see how identifiers can be reused for a different purpose within a particular scope.

Reserved words are identifiers such as `BEGIN`, `IF`, `THEN`, etc. which the MODSIM language itself has already used to define its syntax. These reserved words, which are listed in Appendix B, cannot be used as identifiers, as variables, or as any other user defined name by the programmer for obvious reasons. Although the compiler might be able to keep reserved words distinct from user's identifiers because of their context, it would lead to code which is difficult to read and maintain.

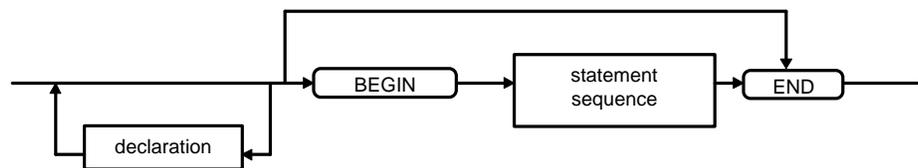
MODSIM also has defined a number of standard procedures such as `MAXOF`, `DISPOSE`, `OUTPUT`, etc. whose identifiers also can not be used by the programmer. These are listed in Appendix C. The identifiers used for the reserved words and standard procedures in

MODSIM are all in upper-case letters. By convention, MODSIM programmers should avoid composing identifiers in all upper-case. This helps the reader of the code to distinguish user-defined items from reserved words.

Finally, there is no limit on the length of an identifier in MODSIM, but the computers on which MODSIM runs may have linkers which truncate long names. See the discussion at the end of Chapter 8 for more information on this subject.

### 2.3 Block Structure and Scope of Variables

The block is the main structural unit of a MODSIM III program. A block is made up of constant, type, variable, and procedure declarations followed by executable code.



**Figure 2-2. Syntax of a Program Block**

A simple MODSIM program can consist simply of a block inserted inside a heading for a **MAIN MODULE**. We can see this clearly in **sample2**:

```

MAIN MODULE sample2;
  block
END MODULE.
  
```

The block is used in several other ways as well. A **PROCEDURE**, which is equivalent to a routine, sub-routine, sub-program or function in other languages, consists of a procedure heading and a block. Later, when the concept of modules is covered in more detail, we will see that modules are also built from blocks.

There are three important characteristics about blocks:

- The identifiers which name constants, types, variables and procedures in a block are known or “visible” only within that block. We say that their scope is limited to that block.
- Memory for variables declared in a block is allocated automatically upon entry to a block and deallocated automatically when leaving a block. Since the block which defines a module is never entered or left, the variables defined in the module remain in place throughout the execution of the program.

When the concept of strong typing is combined with the concept of scope several important benefits are realized:

- Program reliability is enhanced since many more errors in coding can be now detected by the compiler.
- Memory management is improved since memory is allocated for variables only when needed.
- Access to types, variables and procedures is more tightly controlled since they cannot be accessed from outside of their scope.

## 2.4 Nesting of Blocks

Since in MODSIM III one block can be placed within another, we have explicit control over the scope or visibility of identifiers. As an example, we can examine a program skeleton. The blocks in this example are numbered, but there is no corresponding ability or need in the language to give a name to a block.

```

1  [-----] MAIN MODULE Sample3;
      VAR
          x, y: REAL;

      PROCEDURE foo;
      2  [-----] VAR
          a, b: REAL;
          x : INTEGER;
      BEGIN
          x := 5;
          OUTPUT("Second time x=", x);
          ...
      END PROCEDURE;

      BEGIN { main program code starts here }
          ...
          x := 12.34;
          OUTPUT("First time x=", x);
          foo;
          OUTPUT("Third time x=", x);
          ...
      END MODULE.

```

In `Sample3` the `REAL` type variables `x` and `y` and procedure `foo` are visible throughout the entire program, in this case block 1. We say that their scope is **global**. The `REAL` type variables `a` and `b` are visible only within procedure `foo`, i.e. block 2.

## 2.5 Redefinition of Identifiers

The code in `sample3` illustrates another important point about the scope of identifiers and, consequently, the entities to which they provide a name. Note that the identifier for the **REAL** type global variable `x`, which is defined in block 1, is reused to name an **INTEGER** type within block 2. The original variable called `x` is visible everywhere in the program except within block 2 where the new variable called `x` has been defined. Note that the local definition of a new variable called `x` in block 2 does not affect the global variable called `x`. Within block 2 the local definition applies. Outside of block 2, the original definition applies. If we run `sample3`, it outputs the following:

```
First time x = 12.34
Second time x = 5
Third time x = 12.34
```

The point to note is that the local definition of `x` in block 2 temporarily overrides the original definition. Although the global variable called `x` is not visible within block 2, the value stored in that global variable is secure and will once again be available when the program exits from block 2.

It is important to note that the uniqueness and reuse of identifiers is applicable across all identifiers, regardless of the kind of construct they are identifying. Thus, if a globally visible procedure called `foo` exists, it is possible to use `foo` to name an **INTEGER** variable within a block. Within that block, the procedure would not be visible since its name had been usurped to name a variable.

Within a block any attempt to reuse an identifier already defined in that block will be flagged as an error.

## 2.6 Delimiters

A **delimiter** is a programming element which marks the beginning or end of some part or component of a program. Following is a complete list of symbols which serve as delimiters:

Delimiter	Meaning
<b>BEGIN</b>	marks start of a block
<b>END . . .</b>	Marks end of a construct
( )	start and end of parameter lists and list of enumerated constants
[ ]	array index brackets
[ ]	subrange brackets
{ }	comments
(* *)	comments
.	marks end of a module
" "	string delimiter
' '	character delimiter

**Figure 2-3. Delimiters**

## 2.7 Separators

A **separator** is a token which separates two other tokens. For instance:

```

IF SomeStatus <carriage return>
    OUTPUT("Status was TRUE.");<carriage return>
    OUTPUT();<carriage return>
END IF;
```

In the code above, spaces, tabs, and a carriage return were used to separate the elements of an IF statement. A space was used to separate the reserved word “**IF**” from the identifier “**SomeStatus**”, so that it would not be mistaken for “**IFSomeStatus**”. Finally, a semicolon at the end of the statement separates it from the following statement.

Any number of the following separators may be inserted between tokens anywhere in a program to disambiguate the meaning, or to improve readability:

- A **space** character
- A **carriage return** or **new line** character
- A **tab** character
- A **comment** (discussed below).

Statements must be separated by semicolons. Any number of semicolons may be placed before or after any statement, and have no effect, but at least one must be placed between any two statements to separate them.

## 2.8 Comments

A **comment** is an arbitrary sequence of characters which serves to document or comment on the code, but is ignored by the compiler. Comments are delimited by either the curly bracket or brace symbols “{“ and “}”, or by “(“ and “)”. MODSIM allows nested comments. That is, the following:

```
{ { This is a comment } within a comment! }  
or  
{ (* This is a comment *) within a comment! }
```

will all be treated as a comment. Comments may extend over any number of lines. Comments may be nested to any depth. The symbols used to delimit any one level of comment must match. In other words, the delimiters used on either end of a comment cannot be mixed. Nested comments are particularly useful when it is desired to comment out a section of code which might already contain comments.



### 3. Simple Data Types and Their Operators

---

In this chapter we discuss the simple data types. These are characterized by one common trait. They are used to hold a single unit of data. This data might be a number, a character, a text string, a Boolean flag or one value chosen from an enumeration of values. In Chapter 5 we will discuss the more complex structured data types which are used to hold multiple units or aggregates of data.

MODSIM III supports the following built-in simple data types:

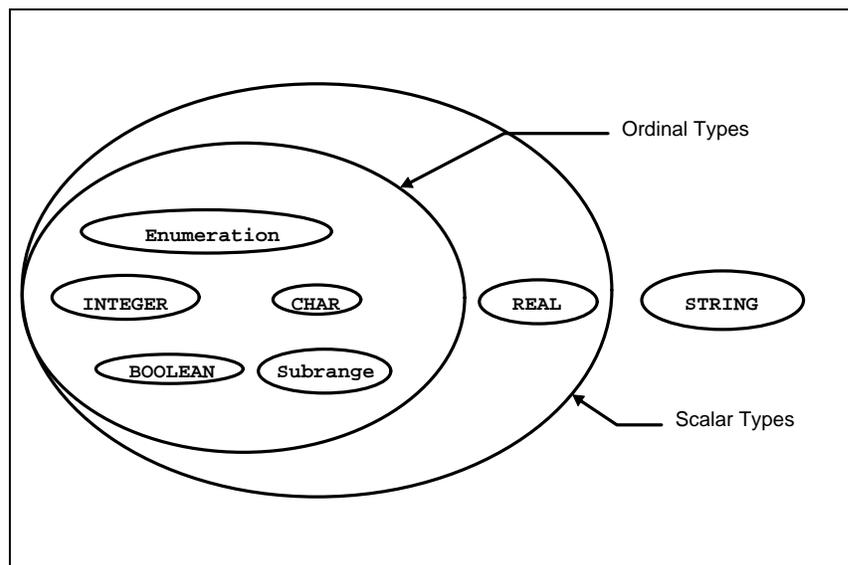
**INTEGER, REAL, BOOLEAN, CHAR, STRING**

There are also two user-defined simple data types:

**enumeration, subrange**

Using these simple data types, it is possible to declare structured data types, arrays, records and objects, which are based on these built-in types.

A subset of the simple data types can be referred to as scalar types because they can be used to scale or measure things. A further subset of the scalar types are known as ordinal types because they have a particular ordering and a known sequence. The figure below shows the relationships.



**Figure 3-1. Simple Data Types**

To store data in a program, the programmer must declare a storage area called a **variable**. This is done by supplying an identifier which names the variable and by specifying the data type. All units of data which can be manipulated in a program are either these variables or literal constants such as “7” or “93.57”.

This chapter starts by discussing what a data type is and how it is represented in the computer's memory. Later we describe how each data type is represented externally; for instance when it is written or printed out.

For each simple data type the language defines a set of operators which can be used to manipulate variables and constants of that data type. These operators are discussed in this chapter. The chapter finishes by giving a brief description of the built-in procedures which allow more elaborate manipulation of data.

### 3.1 What is a Data Type?

The term **type** describes the nature of data and how it will be represented internally in the computer's random-access memory. Data which is stored and manipulated in digital computers typically can be placed into one of three categories:

- Numbers
- Textual information
- Flags or Switches which are **TRUE** or **FALSE**

MODSIM defines a number of built-in data types which can be used to represent each of these categories of data. First we will discuss how these data types are influenced by the way in which a computer internally represents each category of data. Then we will show how each category of data is expressed in MODSIM.

### 3.2 Internal Representation of Data

Hardware and memory limitations in earlier computer systems strongly influenced early language design in the representation of data. The tendency was to provide several alternatively sized representations of each category of data so the programmer could minimize the amount of memory used to store numerical data. This concern for minimizing memory use often led to the use of storage representations which lacked the accuracy to support precise mathematical computation.

In contemporary digital computers, data are typically represented internally using the binary number system. Unfortunately, the binary number system cannot exactly represent real or fractional numbers from the base 10 number system. When a real number is translated into the binary system, only an approximate value can be represented. How-

ever, the greater the number of binary digits, or bits, used to represent the number, the greater the accuracy of representation.

When a smaller number of binary digits is being used in the computer to represent a number, the approximation error will be greater. Large accumulative errors can creep into repetitive calculations. These are known as **approximation errors** or **round off errors**.

Earlier languages typically represented real numbers in 32 bits and provided “double precision” 64 bit representations where accuracy was more important. Similarly there were a number of different ways to represent whole or integer numbers. MODSIM had the advantage of being designed at a time when computer hardware was more mature. The designers decided to specify a larger number of binary digits to represent both real and integer numbers and to do away with alternative ways of expressing the same data type. This means that MODSIM provides accurate numerical representation while simplifying program design since there is only one way to represent each category of data.

### 3.2.1 Representation of Numerical Data

MODSIM III provides two built-in data types which can be used to represent numerical information:

#### **INTEGER** and **REAL**

The **INTEGER** type is used to represent whole numbers. The **REAL** type is used to represent fractional or floating point numbers.

#### 3.2.1.1 Type **INTEGER**

The **INTEGER** type provides an **exact** representation of each whole number. All implementations of MODSIM use at least 32 bits to represent each integer. This means that base 10 numbers in the range -2,147,483,648 to 2,147,483,647 can be exactly represented. This capability is equivalent to a type often called “long integer” in many languages.

#### 3.2.1.2 Type **REAL**

The **REAL** type is used to represent fractional or floating point numbers. MODSIM uses at least 64 bits to represent each real number. Although the handling of real numbers can be hardware specific, this typically means that real numbers in the range -1.7E308 to +1.7E308 can be represented. This is equivalent to a type often called “double precision floating point” in many languages. The capability afforded by this 64-bit representation becomes apparent when compared to 32-bit representations which can typically represent numbers in the range +3.4E38 to -3.4E38. Not only is the range of values which can be represented larger, but the accuracy with which they can be represented is also greater.

There are two built-in functions called **MAX** and **MIN** which determine the largest and smallest possible values for each type on a given machine. On typical machines which implement type **INTEGER** in 32 bits, **MAX(INTEGER)** would return a value of 2,147,483,647. Likewise **MAX(REAL)** would yield 1.7E308.

### 3.2.2 Representation of Textual Data

MODSIM provides two built-in data types which can be used to represent textual information:

#### **CHAR** and **STRING**

##### 3.2.2.1 Type **CHAR**

The **CHAR** type is used to represent a single character. Each character is stored in one byte, which is 8 bits. This means that the **CHAR** type can represent 256 possible character values. The ASCII (American Standard Code for Information Interchange) character set is used in all implementations of MODSIM. This means that the characters from 0 to 127 are as defined by the ASCII standard. The characters from 128 through 255, however were not included in the original ASCII standard, so their interpretation and appearance will vary from computer to computer and printer to printer. The ASCII character set is sometimes referred to as the ANSI (American National Standards Institute) character set. For some time ANSI has considered a draft standard for the upper 128 characters, but has not taken any action.

Although the ASCII character set is in very wide use throughout the world, the international standard is the "ISO 646" (International Standards Organization) character set. Unfortunately it is not possible to easily use this character set with MODSIM. It is a subset of the ASCII character set and lacks the following characters which are used in the language:

[     ]     ^     {     }

Although its native character set is ASCII, MODSIM can be used with katakana or kanji I/O devices and with devices which output the Chinese character set. This is because it supports eight bits in the **CHAR** type in all situations where the hardware and operating system allow this type of support.

Pre-defined MODSIM functions associated with the type **CHAR** include **CAP**, **CHR**, **DEC**, **INC**, **ORD**, **MAX**, **MIN**, and **VAL**. A brief description of these functions follows at the end of this chapter. A full description of these functions is provided in Appendix C.

### 3.2.2.2 Type **STRING**

The **STRING** type is used to represent any sequence of characters. The **STRING** type is fully dynamic. This means that the programmer does not need to specify the size, in number of characters, of the text string to be stored in a **STRING** type. This type is a powerful feature of the language. Most high-level languages do not support dynamic strings. They typically require the programmer to manipulate arrays of characters to achieve the effect of strings in a program.

Among languages which do **not** support dynamic strings are: FORTRAN, Algol, Pascal, C and Ada.

Some languages which **do** support dynamic strings are: MODSIM III, BASIC, SIMSCRIPT II.5 and PL/I.

There are several important characteristics of strings in MODSIM. The characters which make up strings are numbered from 1, not 0. In the string “orange”, the 'o' is at position 1 in the string. The 'g' is at position 5. The length of the string is 6. An empty string is known as a null string. It has a length of 0.

The **CHAR** type is a conformant type to the **STRING** type. This means that a **CHAR** can be used anywhere a **STRING** is expected. A **CHAR** is treated in these situations as a **STRING** of length one.

A conformant type is one which is completely type compatible with another. For instance, the **CHAR** type is conformant to the **STRING** type because any single character is logically equivalent to a string of length one. The opposite is not true however. The **STRING** type is not conformant to the **CHAR** type because a string may, and usually does, consist of more than one character.

Built-in MODSIM functions associated with the type **STRING** include **STRLEN**, **SUBSTR**, **POSITION**, **INSERT**, **REPLACE**, **UPPER**, **LOWER**, **INTTOSTR**, **STRTOINT**, **REALTOSTR**, **STRTOREAL**, **STRTOCHAR**, **CHARTOSTR** and **SCHAR**. A brief description of these functions follows at the end of this chapter. A full description of these functions is provided in Appendix C.

### 3.2.3 Representation of **TRUE / FALSE** or Boolean Data

MODSIM provides a built-in data type which can be used to represent **TRUE / FALSE** or Boolean information:

**BOOLEAN**

### 3.2.3.1 Type **BOOLEAN**

The **BOOLEAN** type is used as a switch or flag to represent a **TRUE** or **FALSE** state. Although this information can be represented in one bit, the information is actually stored in one byte on most implementations. A whole byte is used because that is the smallest unit of data which can be efficiently manipulated in most machines. To pack the data into bits and manipulate the bits would result in poor performance. Since memory limitations are becoming less of an issue, the trade-off between performance and efficient use of memory has been resolved in favor of better performance.

## 3.3 User-defined Types

There are a number of user-defined types. We will cover the two simple types here and defer discussion of the more complex ones until arrays and records have been covered.

### 3.3.1 Enumerated Types

The enumerated type is a completely user-defined type. The programmer explicitly lists all of the possible values for the enumerated type in a specified order. The values are described using standard identifiers. Up to 256 values can be specified in an enumerated type. In other words, an enumeration consists of an ordered collection of values expressed as valid MODSIM identifiers. The **TYPE** declaration below illustrates the use of enumerated types:

```
TYPE
  dayType          = (Sun, Mon, Tue, Wed, Thurs, Fri, Sat);
  compassType     = (North, South, East, West);
  directionType   = (Up, Down);
```

An enumerated type may contain any unique, valid identifier. Note that identifiers may not belong to more than one type since they would then be ambiguous. The ordinal value of any element is available from the built-in function **ORD**, which will return a value from 0 to **n-1** where **n** is the number of elements defined for that enumerated type. For instance, **ORD(Tue)** would evaluate to 2.

Other functions which can operate on enumerated types are **MAX**, **MIN**, **DEC**, **INC**, and **VAL**. These are documented in Appendix C.

Having defined an enumerated type we can then declare a variable of that type. The variable holds one possible value at a time. If we had a variable called **whichWay** of type **directionType**, it could hold either the value **Up** or the value **Down** at any one time.

Relations between values of enumerated types can be checked in Boolean expressions:

```
Tue < Wed is TRUE
Tue > Wed is FALSE
```

Values from two different enumerated types cannot be compared because that would be meaningless. For instance the following expression:

```
Tue > North
```

would be flagged by the compiler as an error.

Unlike other languages which have enumerated types, MODSIM has provisions for output. For instance:

```
IF Tday = Thurs
    OUTPUT(Tday);
...
```

Now that we have introduced the enumerated type, it is worth observing that the **BOOLEAN** type can be considered to be a built-in enumerated type with the following definition:

```
TYPE
BOOLEAN = (FALSE, TRUE);
```

### 3.3.2 Ordinal Data Types

The following subset of scalar data types are known as ordinal types:

```
INTEGER, CHAR, BOOLEAN, Enumeration, Subrange
```

These data types are characterized by a common trait. Each takes on only discrete values and each has a known ordering. In other words, given the integer value *seven* we know that the next possible value is *eight* and the previous value is *six*. This certainty in ordering is not possible with real numbers. Given the real number **98.632491** we cannot say what the next or previous value is. Similarly, given the string "**Hello**" we cannot say what the next value is.

Why the formal definition of ordinal types? Because they are used in several contexts in the language. They are used as indices in arrays, choices in **CASE** statements and to define subrange types.

Before leaving the subject of ordinal types it is worth noting that the expected ordering for type **CHAR** is that given in the ASCII character set. The order for type **BOOLEAN** is as follows: **FALSE, TRUE**.

### 3.3.3 Subrange Types

A subrange type is simply a subset of an ordinal data type. For instance:

```

TYPE
  scoreType = [1..10];
  gradeType = ['A'..'F'];

```

Given an enumerated type definition, we can define a subrange of that type since it is also an ordinal type:

```

TYPE
  dayType      = (Sun, Mon, Tue, Wed, Thurs, Fri, Sat);
  weekdayType = [Mon .. Fri];

```

The advantage of the subrange type is that many logic errors in a program can be caught at run-time. Assigning a value which is out of bounds to a variable of a subrange type causes a run-time error. If we had a variable of type `gradeType` and tried to assign a grade of 'M', this would result in an error.

### 3.4 External Representation of Data

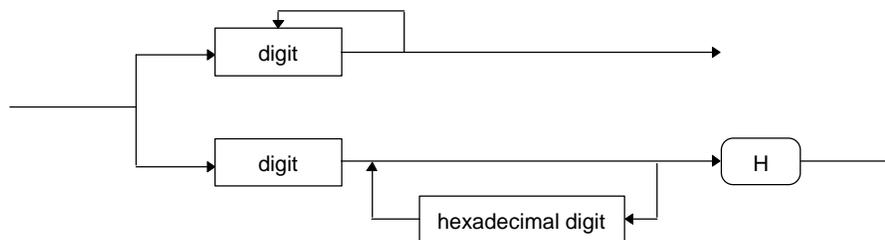
For each of the simple data types just described, MODSIM specifies formats which are used to express values of that type. These are known as **literals**. Literals of each data type are used in three contexts:

- In program code to express constant values
- For input to a program from a terminal or text data file
- For output from a program.

The formats for literals have been designed so that the type of data being described is apparent from examination of the literal.

#### 3.4.1 INTEGER Literals

There are two ways to express integer numbers. They may be expressed as decimal (base 10) or hexadecimal (base 16) literals. In all cases, they are internally represented in the computer as binary numbers. The syntax of an integer literal is as follows:



**Figure 3-2. Syntax of an INTEGER Literal**

The numeric digit described above may be (0-9) for decimal or (0-9, A-F) for hexadecimal. A sign prefix (+ or -) may optionally be added. Hexadecimal literals must begin with a digit (0-9) and end with the character H. To express a hexadecimal number which begins with (A-F), a leading "0" must be supplied. Note that the radix indicator H for hexadecimal and the digits A-F used in hexadecimal literals must be in upper-case.

Thus the decimal number two hundred twenty eight can be expressed in the following ways:

```
decimal      228
hexadecimal 0E4H
```

Both of these literals will be stored internally as the binary number 011100100.

### 3.4.2 REAL Literals

There are two ways to express real numbers. They may be written in the familiar decimal notation or using exponential notation which is sometimes called scientific or engineering notation.

Decimal notation consists of an optional sign, 1 or more decimal digits, a decimal point and 0 or more decimal digits.

Exponential notation consists of an optional sign, 1 or more decimal digits, a decimal point, 0 or more decimal digits, E (for Exponent), an optional sign and a decimal integer.

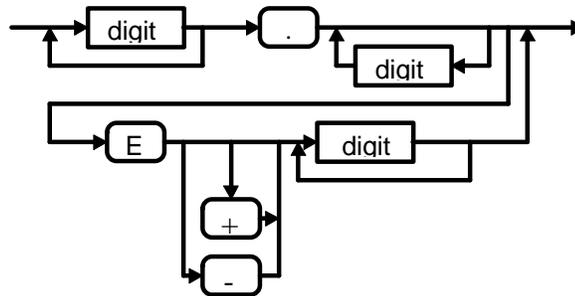


Figure 3-3. Syntax of REAL Literals

Examples of REAL literals are:

```
23.45    1.2    0.3    34.56E12    -13.4E-7    0.0
```

### 3.4.3 CHARACTER Literals

There are two ways to express character literals.

Any printable character (alphanumeric or symbol) can be expressed by enclosing it in apostrophes (single quotes).

```
'A'  'a'  '?'  '$'  '+'
```

Control characters which are not printable can be expressed using a format similar to that for decimal integers. The decimal (i.e. base 10) value of the character is written followed by the letter C. The C must be in upper-case.

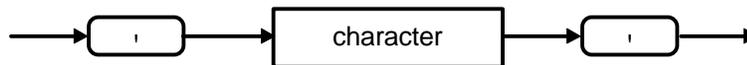
```
13C ⇒ carriage return    7C ⇒ bell    27C ⇒ escape
```

To refer to the apostrophe character itself in a **CHAR** literal, the enclosed apostrophe must be repeated. The following character literal:

```
''''
```

will evaluate as:

```
'
```



**Figure 3-4. Character Literals**

### 3.4.4 STRING Literals

String literals in MODSIM consist of any sequence of printable characters on one line enclosed in quotation marks.

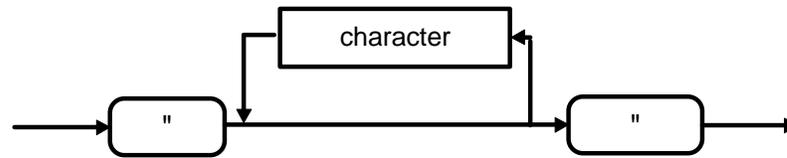
```
"The rain in Spain falls mainly on the plain."
```

To use the quotation character inside a string literal, place two quotation marks wherever a single one is needed in the string. For instance:

```
"He said ""Thank you"""
```

will evaluate as:

```
He said "Thank you"
```



**Figure 3-5. Syntax of String Literals**

### 3.4.5 BOOLEAN Literals

There is one format for Boolean literals. The possible values are **TRUE** and **FALSE**. These identifiers are reserved words, and must be in upper-case. Note that these are literals, not strings, so it is not correct to do the following:

```
IF Status = "TRUE"
```

The correct use is:

```
IF Status = TRUE
```

### 3.4.6 Enumerations

An enumeration consists of an ordered collection of values expressed as valid MODSIM identifiers. Therefore a literal for an enumerated type is simply a MODSIM identifier.

## 3.5 Operators

An operator is a symbol or reserved word which specifies an action or operation to be performed. “+”, “-”, and “DIV” are examples of operators. A complete list of reserved words is given in Appendix B.

The operations which can be performed on variables and constants of each data type differ with the data type. For example, it is obvious that the “+” operator can be used to add two integer constants together:

```
2 + 2
```

but it would not be appropriate to use the same operator with two Boolean constants:

```
TRUE + FALSE
```

This has no meaning.

This section lists the operations which may be performed on variables and constants of each built-in data type. Later, in the sections which cover expressions and assignment, some of this information will be described in greater detail.

In addition to these fundamental operators which allow for the basic manipulation of data, MODSIM supports a rich collection of built-in procedures to perform more specialized and/or elaborate operations on data. Examples of these procedures include the **SUBSTR** functions which return a substring of the input string and the **ROUND** function which rounds real numbers to the nearest whole number. These built-in procedures are described in detail in Appendix C.

Finally, MODSIM provides a standard set of library modules from which more specialized support procedures and objects can be imported for use in a program.

### 3.5.1 Assignment Operator

The assignment operator is used to assign the value of a constant, variable or expression to a variable for storage. The assignment operator is the colon followed immediately by the equal sign:

**:=**

Many languages use the equal sign alone for assignment, but since the = operator is used for logical comparison the := operator more clearly denotes the intended action. As a note of historical interest the ← symbol was in early drafts of the ASCII character set. It was intended to be used as the assignment operator. It may help to enforce the notion of assignment to think of the := symbol as an equivalent to the ← symbol. Thus,

**x := 13.5**      is equivalent to      **x ← 13.5**

The assignment operator is valid for all types, both built-in and user-defined.

### 3.5.2 Arithmetic Operators

The arithmetic operators are used to manipulate variables and constants of the **INTEGER** and **REAL** types. Figure 6 shows a list of the arithmetic operators and the types to which they can be applied.

Note that some operations apply only to one type or the other. This is because MODSIM is a strongly typed language and some operations are meaningful only with that type.

The modulus operator returns the remainder of an integer division. Thus, the expression **33 MOD 7** would evaluate to **5**. Likewise the expression **33 DIV 7** would evaluate to a **4**. The expression **5.0 / 2.0** would evaluate to **2.5**.

Most of the operators are “binary” operators, i.e. they take two operands, one on either side. The unary plus and minus operators are used to change the sign of a variable. These take only one operand which is to the immediate right. Thus, if **x** evaluates to **5.3**, then **-x** evaluates to **-5.3**.

Operator	Meaning	Applicable Types
+	addition	<b>INTEGER, REAL</b>
-	subtraction	<b>INTEGER, REAL</b>
*	multiplication	<b>INTEGER, REAL</b>
/	real division	<b>REAL</b>
<b>DIV</b>	integer division	<b>INTEGER</b>
<b>MOD</b>	modulus	<b>INTEGER</b>
+	unary plus	<b>INTEGER, REAL</b>
-	unary minus	<b>INTEGER, REAL</b>

**Figure 3-6. Arithmetic Operators**

The two operands of each binary operator must be type compatible and that type must be appropriate for the operator. Thus, the following combinations of operands and operators are legal:

```
2.0 + 2.0
2 + 2
5.0 / 7.0
8 DIV 2
```

while the following are not legal:

```
2.0 + 2      ⇒ mixed REAL and INTEGER operands
5.0 DIV 7.0 ⇒ operand types incompatible with operator
5 / 7       ⇒ operand types incompatible with operator
```

### 3.5.3 Relational Operators

The relational operators are used to perform comparisons of values. They are all binary operators which take an operand on either side. Like arithmetic operators, the operands used with relational operators must both be of the same type. The result of the expression is of type **BOOLEAN**.

Operator	Meaning	Applicable Types
=	equal	All types
<>	not equal	All types
<	less than	All scalar types and <b>STRING</b>
<=	less than or equal	All scalar types and <b>STRING</b>
>	greater than	All scalar types and <b>STRING</b>
>=	greater than or equal	All scalar types and <b>STRING</b>

**Figure 3-7. Relational Operators**

### 3.5.4 Logical Operators

There are three logical operators which can be applied to Boolean expressions:

**NOT, AND, OR**

**NOT** is a unary operator which takes one operand on its right. **AND** and **OR** are binary operators which take two operands, one on either side. The following table summarizes the effect of each logical operator:

<b>NOT TRUE</b>	<b>⇒</b>	<b>FALSE</b>
<b>NOT FALSE</b>	<b>⇒</b>	<b>TRUE</b>
<b>TRUE AND TRUE</b>	<b>⇒</b>	<b>TRUE</b>
<b>FALSE AND TRUE</b>	<b>⇒</b>	<b>FALSE</b>
<b>FALSE AND FALSE</b>	<b>⇒</b>	<b>FALSE</b>
<b>TRUE OR TRUE</b>	<b>⇒</b>	<b>TRUE</b>
<b>FALSE OR TRUE</b>	<b>⇒</b>	<b>TRUE</b>
<b>FALSE OR FALSE</b>	<b>⇒</b>	<b>FALSE</b>

**Figure 3-8. Logical Operators**

MODSIM uses "short circuit" evaluation of Boolean expressions. This means that only as much of an expression is evaluated as is needed to determine the value of the expression. This topic will be covered in more detail later.

There is another operator which we will mention here for completeness. We will defer discussion of its use until expressions and structured types have been covered.

The “.” is used to construct qualified identifiers which refer to individual fields of a record.

### 3.6 Built-in Procedures and Functions

MODSIM provides a number of predefined or built-in procedures and functions. Below is a partial list of these procedures. Appendix C provides a complete listing in greater detail of each procedure and function.

<b>Function</b>	<b>Use</b>
<b>ABS (Num)</b>	Absolute value of <b>INTEGER</b> or <b>REAL</b> number.
<b>CAP (Chr)</b>	Returns the upper-case equivalent of the character <b>Chr</b> .

<b>CHARTOSTR</b> (ArrayOfChar)	Returns the <b>STRING</b> representation of an <b>ARRAY OF CHAR</b> .
<b>CHR</b> (Int)	Returns the character with the given <b>INTEGER</b> ordinal value.
<b>CLONE</b> (Ref)	Returns a copy of the <b>ARRAY</b> , <b>RECORD</b> or <b>OBJECT</b> instance referred to by <b>Ref</b> .
<b>FLOAT</b> (Int)	Converts <b>INTEGER</b> to <b>REAL</b> .
<b>HIGH</b> (Array[])	Returns the high bound of an array element.
<b>INTTOSTR</b> (Int)	Returns the <b>STRING</b> representation of <b>Int</b> .
<b>LOW</b> (Array[])	Returns the low bound of an array element.
<b>LOWER</b> (Str)	Returns a string in which all characters in <b>str</b> have been changed to lower case.
<b>MAX</b> (ScalarType)	Returns the highest allowed value of a scalar type.
<b>MAXOF</b> (ScalarTypList)	Returns the highest value from the list.
<b>MIN</b> (ScalarType)	Returns the lowest allowed value of a scalar type.
<b>MINOF</b> (ScalarTypList)	Returns the lowest value from the list.
<b>ODD</b> (Num)	Returns <b>TRUE</b> if odd, <b>FALSE</b> if even.
<b>ORD</b> (Ordinal)	Returns the particular ordinal value of an ordinal type.
<b>POSITION</b> (Str1, Str2)	Returns the position of <b>Str2</b> in <b>Str1</b> .
<b>REALTOSTR</b> (RlNum)	Returns the <b>STRING</b> representation of <b>RlNum</b> .
<b>SCHAR</b> (Str, pos)	Returns the character at position <b>pos</b> in <b>Str</b> .
<b>STRLEN</b> (Str)	Returns the length of string <b>Str</b> .
<b>STRTOINT</b> (Str)	Returns the <b>INTEGER</b> representation of <b>Str</b> .
<b>STRTOREAL</b> (Str)	Returns the <b>REAL</b> representation of <b>Str</b> .
<b>SUBSTR</b> (pos1, pos2, Str)	Returns substring of <b>Str</b> from <b>pos1</b> to <b>pos2</b> .
<b>TRUNC</b> (RealNum)	Truncates a <b>REAL</b> value to <b>INTEGER</b> .
<b>UPPER</b> (Str)	Returns a string in which all characters in <b>str</b> have been changed to upper-case.
<b>VAL</b> (OrdType, OrdNum)	Returns the <b>OrdType</b> value which has ordinal value <b>OrdNum</b> .

<b>Procedure</b>	<b>Use</b>
<b>DEC(Ord [,n])</b>	Decrements an ordinal variable by <b>n</b> . <b>n</b> defaults to 1 if omitted.
<b>DISPOSE(Ref)</b>	Deallocates the instance of the <b>ARRAY</b> , <b>RECORD</b> or <b>OBJECT</b> referenced by <b>Ref</b> .
<b>HALT</b>	Terminates a program.
<b>INC(Ord [,n])</b>	Increments an ordinal variable by <b>n</b> . <b>n</b> defaults to 1 if omitted.
<b>INPUT(Var1,...)</b>	Reads values from the standard input.
<b>INSERT(Str1, pos, Str2)</b>	Inserts <b>Str2</b> at position <b>pos</b> in <b>Str1</b> .
<b>NEW(Ref)</b>	Allocates an instance of <b>Ref</b> , and returns a reference to it. Can be used with <b>ARRAYS</b> , <b>RECORDS</b> and <b>OBJECTS</b> .
<b>OUTPUT(Var1,...)</b>	Writes values to the standard output.
<b>REPLACE(Str1, pos1, pos2, Str2)</b>	Replaces the part of <b>Str1</b> from <b>pos1</b> to <b>pos2</b> with <b>Str2</b> .
<b>STRTOCHAR(Str, ArryOfChar)</b>	Converts <b>Str</b> to an <b>ARRAY OF CHAR</b> .

## 4. Declarations, Expressions and Precedence

---

The previous chapter discussed built-in data types and how to express values of each data type. This chapter discusses how to declare constants, declare user-defined types which are based on the built-in types, how to declare variables of any type and, finally, how to use variables and constants in expressions. Before describing data structures and showing how they are declared we first need to discuss what a declaration is and why it is needed.

### 4.1 Declarations

Declarations are statements placed ahead of executable code in a program or block which describe the nature of the data which is to be manipulated. The language uses these declarations to allocate and organize the memory which will be used to store the data and to check for consistent use of that data. For example, if we have declared that a variable named **x** will be used to store **REAL** numbers, then **x := "Hello"** will be caught as an error when the program is compiled.

Another type of declaration is a procedure declaration which describes a **procedure** or routine which will be called later by the program's code.

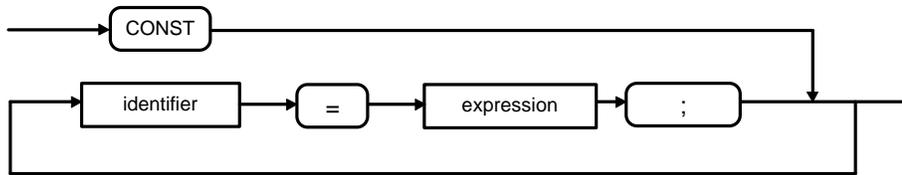
To review the information in Chapter 2, there are four categories of declarations: constant, type, variable and procedure. They can actually appear in any order, but, since the declarations tend to build one upon the other, they are usually found in the order just described. It is also worth noting that the declarations need not be grouped in only one declaration section. For instance, there could be several **VAR** declarations sections.

#### 4.1.1 CONSTANT Declarations

A constant declaration defines an expression which is substituted at compile-time for the specified identifier everywhere it occurs. Constants are useful for several reasons:

- Program code becomes more readable when we refer to something by name rather than by value – i.e **NumEmployees** vs. **57**. Instead of sprinkling program code with meaningless literal constants, the programmer uses a constant with good mnemonic value.
- If a value used throughout the program needs to be changed, it needs to be changed only once in the constant declaration section if a constant has been used throughout the code instead of once for each use of a literal.
- A constant expression is evaluated only once at compile time, not every time the constant is used in a program.

A constant is declared in a constant declaration block as shown in the syntax diagram below.



**Figure 4-1. Syntax of a Constant Declaration**

It consists of the reserved word **CONST** followed by pairs of identifiers and expressions connected with the equal sign. A constant declaration block ends when another declaration block such as a variable declaration block starts or when a **BEGIN** statement is reached.

Once a constant has been declared, its value cannot be altered. Since a constant may be defined by an expression (such as  $2 + 3$ ) one declaration can build on another. For instance:

```

CONST
  pi      = 3.141592654;
  ArraySize = 25;
  circ    = 2.0 * pi;
  twoPi   = circ;
  hello   = "Hello";
  
```

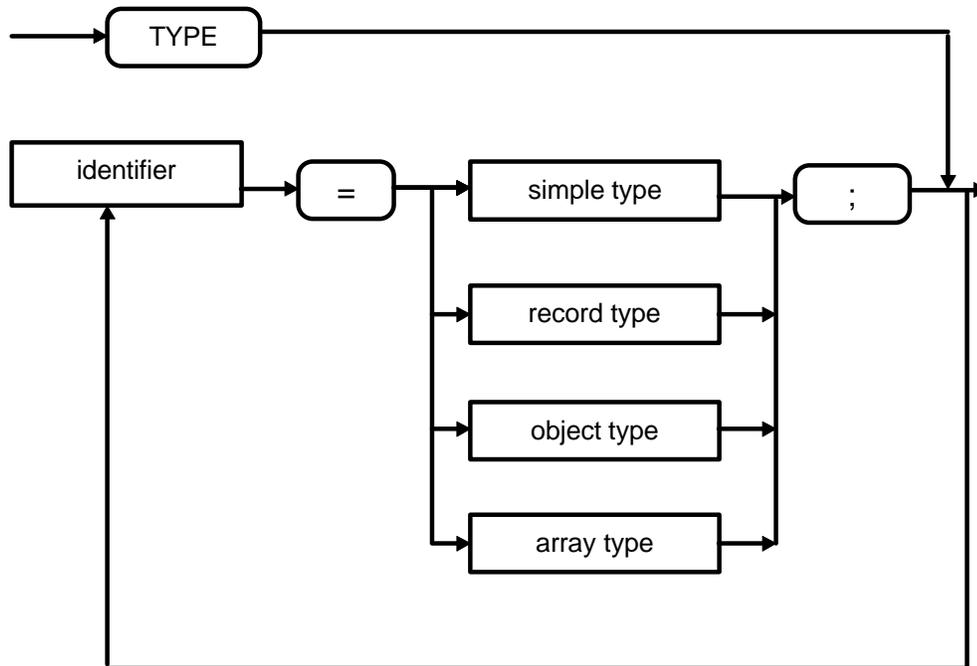
Only a constant or a literal may be used in the expression which defines a constant.

#### 4.1.2 TYPE Declarations

The **TYPE** declaration is used to create new data types which build upon the existing built-in types or other user-defined types. The only user-defined data type we have introduced so far is the enumeration. So we will use the enumeration to show how type declarations are expressed. Later, when the structured data types **ARRAY**, **RECORD** and **OBJECT** have been introduced, we will revisit the subject and elaborate on **TYPE** declarations.

```

TYPE
  carType = (Chrysler, Ford, Porsche, Saab);
  colorType = (red, yellow, green);
  
```



**Figure 4-2. Syntax of a Type Declaration**

Note that the identifiers used to name the user-defined enumeration types both end with the word **type**. This is simply a convention which helps make code more readable. The word **type**, when appended to the word **color** gives the identifier some mnemonic content. The identifier is being used to name a type definition which holds information about colors. This convention helps to distinguish type identifiers from variable identifiers.

### 4.1.3 VARIABLE Declarations

Variable declarations are the statements which reserve memory space to hold a program's data and which assign names to these data storage areas. The syntax is straightforward. One or more identifiers is listed followed by a colon and the data type. Note that some of the variable declarations below draw on the type definitions we declared above.

```

VAR
  n, j, k      : INTEGER;
  x, y         : REAL;
  filename     : STRING;
  car          : carType;
  trafficLight : colorType;
  
```

For each identifier listed on the left of a colon, memory space appropriate for the data type specified on the right of the colon is reserved.

#### 4.1.4 PROCEDURE Declarations

Finally we have procedure declarations. These are the definitions for user-defined procedures. Constant, type and variable declarations describe data. Procedure declarations specify the internal data and executable code which will be used to execute the procedure. User-defined procedures will be covered in detail in Chapter 7.

#### 4.1.5 PROCEDURE Variable Declarations

You can declare a type or variable as a procedure type. Variables of this type may be assigned a procedure that returns the specified type. To invoke the procedure using such a variable, the variable is preceded by the **CALL** keyword and followed by an argument list.

**Note:** No type checking will be performed on the argument list as to either the number of types or the number of arguments. For example:

```

VAR
  p:PROCEDURE BOOLEAN;

PROCEDURE IsInRange(IN x : INTEGER) : BOOLEAN;
BEGIN
  ... { Implementation code }
END PROCEDURE;

BEGIN {MAIN}
  p:= IsInRange;
  IF CALL P(49)

  ...

END MODULE.
```

## 4.2 Automatic Initialization of Variables

MODSIM provides for the automatic initialization of all variables. Each variable is automatically initialized to a specific value at the time it is declared. **INTEGER** and **REAL** variables are initialized to zero. Variables of ordinal types such as **CHAR**, **BOOLEAN** and enumerations are initialized to the value of that type which has ordinal value zero. For **CHAR** this is the ASCII NUL character, for the **BOOLEAN** type this is **FALSE** and for enumerated types it is the first value in the list. Subrange variables are initialized to the lowest value in the subrange. Finally, variables of type **STRING** are initialized to the null string. A null string is a string of length zero.

### 4.3 Expressions

An expression is a collection of variables, constants or sub-expressions connected by operators. Once all of the operations have been performed one value results. Here is a simple expression:

$$2 + 2$$

Once the addition operation on the two constants has been performed we are left with the result 4.

Expressions can yield numerical, textual or Boolean results. Since an expression is always evaluated to one resulting value, an expression always yields results of one type. For the moment we will concentrate on numerical expressions as we explain the evaluation rules.

Here is a more complicated expression which includes the previous expression as one of its components:

$$5 + ( 3 * ( 2 + 2 ) )$$

Note that we used parentheses to clarify our intention for evaluation of this expression. In this case we evaluate the subexpression  $2 + 2$  yielding a result of 4 and then evaluate the remaining expression, and so on until we have one resulting answer.

$$5 + ( 3 * ( 2 + 2 ) ) \Rightarrow 5 + ( 3 * 4 )$$

$$5 + ( 3 * 4 ) \Rightarrow 5 + 12$$

$$5 + 12 \Rightarrow 17$$

### 4.4 Operator Precedence

The above operations were performed in a certain order. There are precedence rules for the operators. The rule for evaluating expressions is that the innermost parenthesized portions are evaluated first. Within any one level of parenthesization the operations are performed in a specific order. The following table gives the order of precedence for the operators. The highest precedence, those which are performed first, are at the top. Within any group of operators at the same level the precedence is from left to right. The operator precedence in MODSIM is described below:

highest	( )	function calls	.	[ ]	NOT
↑↑	*	/	DIV	MOD	AND
↓↓	+	-	OR		
lowest	=	<>	<	<=	> >=

Figure 4-3. Operator Precedence

## 4.5 Types of Expressions

The expressions shown above are mathematical expressions. There are two types of these: those which yield an **INTEGER** result and those which yield a **REAL** result. These have a direct analog in traditional mathematical expressions.

Another type of expression is one which yields a **STRING** type. For example:

```
"To be or " + "not to be." ⇒ "To be or not to be."
```

Expressions can yield **CHAR** types. For example, if we use the built-in **INC** function to increment a character, we get the next character:

```
INC('A') ⇒ 'B'
```

One of the more useful expression types is the Boolean expression which, of course, yields a **BOOLEAN** result. Boolean expressions are used wherever a **TRUE** or **FALSE** answer is expected. The most common place to find them is in an **IF** statement.

```
IF x <= 0
  OUTPUT("x was less than or equal to zero");
ELSE
  OUTPUT("x was greater than zero");
END IF
```

The **NOT**, **AND** and **OR** operators can be used to build more complex expressions:

```
( x <= 0 ) AND ( y > 13 ) OR ( day = Thurs )
```

### 4.5.1 Evaluating Boolean Expressions

MODSIM optimizes evaluation of Boolean expressions by using **short-circuit evaluation**. A complex **BOOLEAN** expression is evaluated only as far as is necessary to determine its value. The following truth tables illustrate this:

<b>x</b>	<b>AND</b>	<b>y</b>	result
<b>FALSE</b>		not evaluated	<b>FALSE</b>
<b>TRUE</b>		<b>FALSE</b>	<b>FALSE</b>
<b>TRUE</b>		<b>TRUE</b>	<b>TRUE</b>
<b>x</b>	<b>OR</b>	<b>y</b>	result
<b>FALSE</b>		<b>FALSE</b>	<b>FALSE</b>
<b>FALSE</b>		<b>TRUE</b>	<b>TRUE</b>
<b>TRUE</b>		not evaluated	<b>TRUE</b>
	<b>NOT</b>	<b>x</b>	result
		<b>FALSE</b>	<b>TRUE</b>
		<b>TRUE</b>	<b>FALSE</b>

**Figure 4-4. Short Circuit Logic**

This feature is important to the programmer for two reasons:

- As an optimization it avoids evaluating portions of a potentially complicated expression which will not contribute to the answer.
- It allows the programmer to check a variable to see if it falls within some value range before proceeding to use it in an expression. This is particularly useful in preventing divide by zero errors.

Consider :

```
IF ( x <= 37 ) AND ( SomeFunction(k) / SQRT(x) < 21.0 )
...

```

In some languages, both expressions will be evaluated even if **x** is greater than 37. Short-circuit logic will prevent this by not evaluating the second operand if the first is false, thus short-circuiting the statement. In other words, in an **AND** expression the answer will be false if either operand is false, so there is no need to continue once it is clear that the first operand is false.

Next consider:

```
IF ( x <> 0.0 ) AND ( Y / X < 20.0 ) ...

```

In this case, if the variable **x** contains a value of zero, the first operand of the **AND** evaluates to false and the rest of the expression is ignored. In this case ignoring the rest of the expression is desirable. Otherwise, when the expression **Y / X** is evaluated, we would be dividing by zero and this could cause a run time error. Without short-circuit evaluation of Boolean expressions we would have to construct a nested **IF** statement which first checked to see if **x** contained the value zero before checking the next **IF** statement to see if **Y / X < 20.0**.



## 5. Structured Data Types

---

There are times when the built-in simple data types described earlier are not adequate for some programming tasks. Often it is desirable to build and use more complex data structures. The structured data type fills this need. It is a user-defined type which is composed of more than one data element.

There are three structured data types:

- ARRAY**      An ordered set of data elements referenced using an index or indices. All elements in an array are of the same type.
- RECORD**      A user-defined data structure composed of some number of built-in types or other data structures. Each element is a field which has a name and can be referenced individually.
- OBJECT**      Objects in MODSIM are dynamically allocated data structures coupled with routines, called methods. Objects have fields, as records do. The object's fields define its state at any instant in time while its methods describe the actions which the object can perform. Objects will be discussed in Section II of this manual.

We will examine how these structured data types are used, and will then discuss alternative ways of managing their memory allocation before covering each one in detail.

### 5.1 Using Structured Data Types

To provide a framework for the discussion on the use of structured data types, we will contrive a simple problem and examine a number of ways in which it can be solved.

**Problem:**      *We want to store and retrieve information about the names and ages of a family's children and to be able to do this conveniently.*

The family we use in this example has four children:

Who	Age
Joe	28
Sue	25
Tim	18
Ana	16

Using the simple data types we have discussed so far, our choices are few. Here is the code we could write:

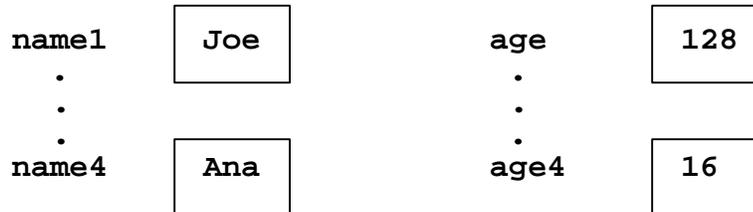
```

VAR
  name1, name2, name3, name4 : STRING;
  age1, age2, age3, age4    : INTEGER;
BEGIN

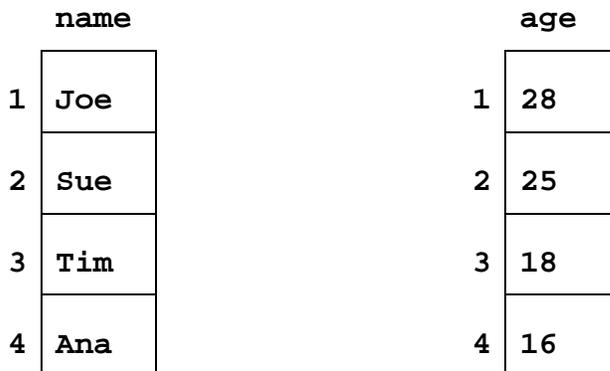
  name1 := "Joe";  age1 := 28;
  ...
  name4 := "Ana";  age4 := 16;

```

This is how it would appear in memory:



But this is not very useful. In order to discover Ana's age we would have to know that it was stored in the variable `age4`. We would like to be able to say, "*Give me the age of the fourth child in the family*". This is where the array can be useful. When there is some natural ordering we can use as an index, the array becomes very useful:



Here we have constructed two arrays. One holds the names and the other the ages of the children. We need two arrays, because an array can hold only one data type. Thus, we must store the names in an **ARRAY OF STRING** and the ages in an **ARRAY OF INTEGER**. Now, we can look up the "*age of the fourth child in the family*" by checking the fourth entry in the `age` array. This is how this problem would be solved in a language like FORTRAN.

More contemporary languages, however, would use a record to hold the related name and age information. First, we would define a record with two fields: **name** and **age**. Then we would make an array of these fields. Here is the record:

<b>name</b>	<b>age</b>
-------------	------------

And here is an array of these records filled with the data about our family:

	<b>name</b>	<b>age</b>
<b>1</b>	<b>Joe</b>	<b>28</b>
<b>2</b>	<b>Sue</b>	<b>25</b>
<b>3</b>	<b>Tim</b>	<b>18</b>
<b>4</b>	<b>Ana</b>	<b>16</b>

This is a more useful structure because we can find the record at index location 4 in the array and then query about any of the information in that record. We do not have to look in a separate array for each item of information about the sibling.

So far so good. But there is a nagging problem. How big should we make the array so that it can handle any size family? Languages typically require arrays to be declared with information about the size of the array. This information is fixed at the time the program is written and cannot be changed or specified while the program is running.

What about a family with 13 children? Or how would we handle the family where a widow and widower, each with 13 children, married. Now we have to allow for 26 entries. This seems quite wasteful when the average family might only have 2 or 3 children.

MODSIM provides a solution to this problem. Its arrays are **dynamic**. They are allocated explicitly by the program while it is running. The required size of the array can be computed by the program and specified at the time it is allocated.

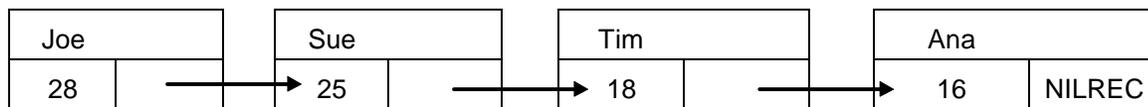
But we still have a problem to solve. We have been indexing into the array of records using the inherent ordering in a family's children: "*What is the age of the fourth child in the family?*". The combined family of 26 children has two orderings. The fourth child in the father's original family and the fourth child in the mother's original family. What now? In fact this is a fairly common situation. We are often presented with data which has no natural ordinal index.

The solution involves the use of records. We simply store the information for each child in a record and then link the records together. We can link them together by making one record point to the next. We can do this because a reference to a record is a legal field in a record. The advantage is that we can string together an arbitrary number of these records without having to state in advance how many there will be.

Even with dynamic arrays we eventually reach a moment of truth when we have to finally decide how big to make the array. A dynamic array cannot be grown in size once it is allocated. With records, each one is allocated individually. This means that we can link together any number of records into a linked list. Here is a record to which we have added a field pointing to the next record in the list:

name	
age	next

Now we can allocate an instance of this new record type, fill it with data and link it to the next one. We end up with a structure which looks like this:



The **NILREC** in the **next** field of the last record indicates that there is no record following that one.

Refer to Chapter 14 for a discussion of linked lists in MODSIM.

### 5.1.1 Dynamic Versus Fixed Structures

The point of the discussion above was to demonstrate the advantage of the dynamic data structure when compared to fixed data structures. When records and arrays use fixed allocation like the **INTEGER** and **REAL** types, then all information concerning the number and size of these structures must be stated at the time the program is written. By contrast the size of dynamically allocated arrays can be decided while the program is running. With dynamically allocated records, new instances can be allocated as needed and linked to a list of records. Objects in MODSIM also behave in the same way.

Because the dynamic structured data type is normally the more useful, MODSIM supports the **ARRAY**, **RECORD** and **OBJECT** types as dynamically allocated types. For the more limited occasions when fixed allocation may be more appropriate, MODSIM provides the **FIXED ARRAY** and the **FIXED RECORD**.

In the remainder of this chapter, we will first discuss how dynamic data structures are allocated and deallocated, and then we will cover the **RECORD** and **ARRAY** types. Finally, we will discuss the **FIXED ARRAY** and the **FIXED RECORD**.

## 5.2 Memory Management of Dynamic Data Structures

The built-in and user defined data types we have discussed so far, such as the **INTEGER**, **REAL**, **STRING**, **CHAR**, **BOOLEAN** and enumerations, share several important attributes which characterize the way they are used:

- A variable of the desired type must be declared. Conceptually, this variable will contain the data.
- The memory which is used to store variables of each type is automatically allocated when the program enters the block in which the variable has been declared.
- The memory is automatically deallocated when the block is exited.
- For each piece of memory which holds data, there is one and only one identifier which refers to it.
- When we assign one variable to another, the value of the data in one is copied into the other. Two copies of the data then exist.

We can describe these as “fixed” data types. The storage requirements are fixed at the time the program is written and cannot be changed while the program is running. A programmer can perform a large variety of tasks with these data types. There is one limitation, however. All data storage requirements must be known and stated in advance using variable declarations.

MODSIM supports three “dynamic” data types: **ARRAY**, **RECORD** and **OBJECT**. These types bring with them a number of capabilities not available with the fixed data types discussed so far. In particular, they provide a dynamic memory management capability. This capability is particularly important in the case of structured data types because they can occupy significantly larger amounts of memory than the simple data types which only contain one element of data.

There are several ways in which these dynamic data types differ from the fixed data types:

- Memory is explicitly allocated and deallocated by the programmer using the built-in procedures **NEW** and **DISPOSE**.
- There can be more than one identifier which refers to a particular dynamic data structure.

- A dynamic data structure can exist with no identifier referring to it.

This last point is very important. It means that the programmer can allocate an instance of an **OBJECT**, **RECORD** or **ARRAY** and place it in a MODSIM group or build a linked list with it. It is not necessary to declare a variable for each instance of a data structure which might be created by a program while it runs.

The variables which refer to dynamic data structures in MODSIM are called **reference variables**. In one sense they act like pointers do in many languages. A dynamic data structure can have any number of reference variables referring to it, or it can have none.

MODSIM's dynamic data types, their reference variables and the way they are used, differ from pointers in one important respect. It is not necessary to employ a special syntax to reference this data in MODSIM. References to these structured data types are made in the same way as for simple data types.

Despite the significant added utility of these dynamically allocated structured data types, there are only two ways in which their use differs from the fixed data types:

- The programmer controls the allocation and deallocation of instances of this type.
- The effect of an assignment of reference variables differs from assignment of fixed data types where an actual copy of the data is made.

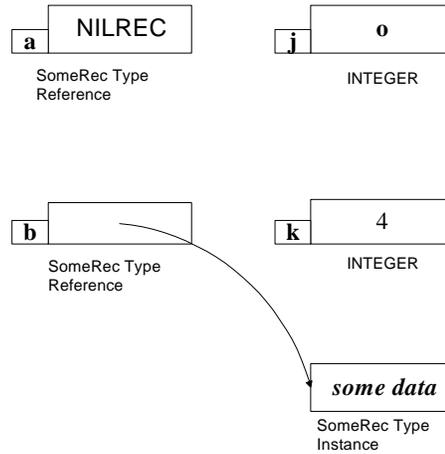
Since structured data types can contain significant amounts of data, making a copy of that data during an assignment is inappropriate. Instead, the reference variable being assigned to it is simply made to refer to the same instance of structured data. The following code fragment illustrates the difference:

```

TYPE
  SomeRecType = a record type declaration ;
VAR
  j, k : INTEGER;
  a, b : SomeRecType;
  ...
  k := 4;
  NEW(b); { allocate an instance of b }
  b := some record data { fill record b with data }
  ...
  j := k; { a copy of the value in k is stored in j }
  a := b; { a and b now refer to the same record
           instance }

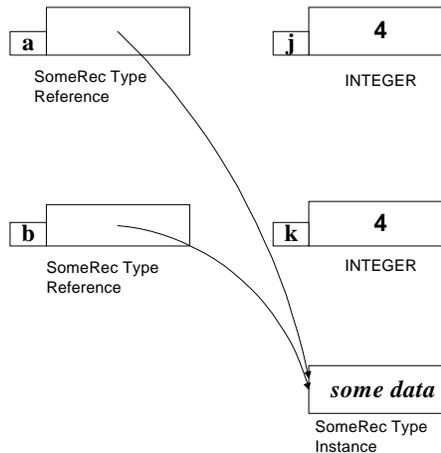
```

Here is a diagram which illustrates how the data is organized in memory before the last two assignments are made:



**Figure 5-1. Memory Before Assignments**

After the assignments are made this is how the data is organized:



**Figure 5-2. Memory After Assignments**

The important point to remember is that variables of type **ARRAY**, **RECORD** and **OBJECT** are reference variables. They contain no data. Instead, they simply refer to an instance of a structured data type which does contain the data. While a variable for a simple data type uniquely names a particular storage area, a reference variable for an **ARRAY**, **RECORD** or **OBJECT** type is a way of referring to unnamed data structures of that type.

### 5.2.1 The CLONE Function

There are times when the programmer does, in fact, want a copy of a dynamic data type. MODSIM provides a built-in function called **CLONE** for this purpose. It works with the three dynamic data types: **ARRAY**, **RECORD** and **OBJECT**.

The **CLONE** function takes a reference variable as an argument. It then allocates space for a new instance of the same type and copies the values of every element of the original into the new instance. Finally, it returns a reference to the new copy. **CLONE** does not always copy all fields. Refer to paragraph 9.11.

The user could accomplish the same effect by allocating a new instance and explicitly copying element by element, but the **CLONE** function provides a short-cut way of expressing this functionality and accomplishes the copy more efficiently.

To illustrate the **CLONE** function we will revisit the previous example and substitute a call to the **CLONE** function instead of using an assignment statement.

```

TYPE
  SomeRecType = a record type declaration;
VAR
  j, k : INTEGER;
  a, b : SomeRecType;
...
  k := 4;
  NEW(b); { allocate an instance of b }
  b := some data { fill record b with data }
...
  j := k; { a copy of the value in k is stored in j }
  a := CLONE(b); { a now refers to a copy of b }

```

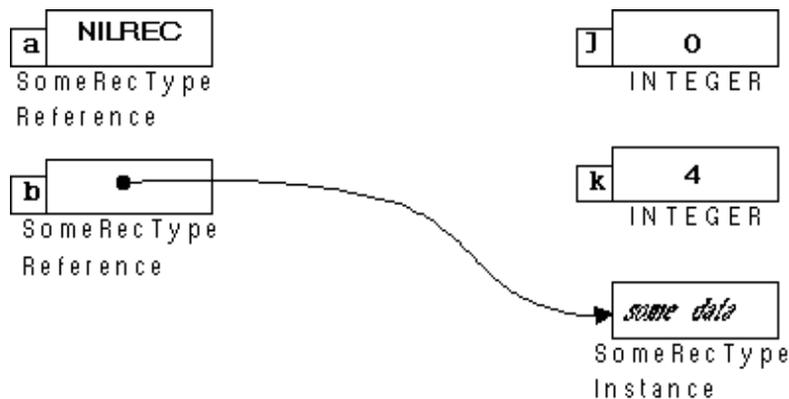
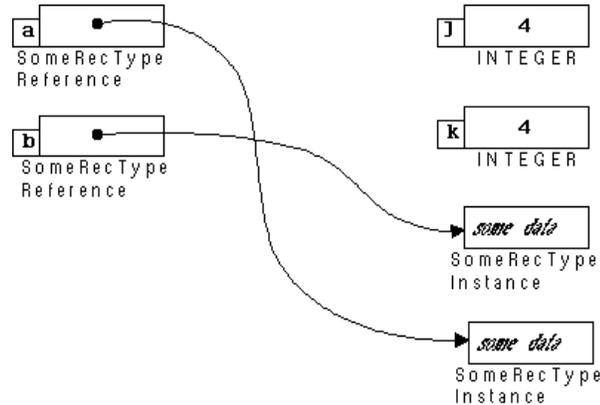


Figure 5-3. Memory Before CLONE and Assignment

After the **CLONE** and assignment are done, the data is organized as follows:



**Figure 5-4. Memory After CLONE and Assignment**

### 5.2.2 Orphaned Data

Since a dynamic data structure can be referred to by zero, one or many reference variables, it is possible for an instance to become “orphaned” or fall into limbo. If an instance of data has no reference variable referring to it and is not being kept in some linked list or group, then it is orphaned. It is no longer possible to reference that data, yet it still exists and is using memory.

In the code above, if the reference variable **a** had been referring to some instance of data, and it was the only reference, then any assignment to **a** would cause the original instance of data to be orphaned.

If this occurs in a loop or some other construct which is executed repeatedly, the program will eventually run out of memory. This is called a memory leak.

### 5.2.3 The DISPOSE Procedure

The key point to remember is that the memory used by the lost data is no longer available during the remainder of the time that the program is running. If the data is no longer needed, then the instance of data should be returned to the system using the built-in procedure **DISPOSE**. The **DISPOSE** procedure takes an **ARRAY**, **RECORD** or **OBJECT** instance and destroys it. The memory previously used for the instance of data then becomes available for allocation with **NEW** and the reference variable is re-initialized.

### 5.2.4 Hanging References

Just as there can be orphaned instances of data, there can be reference variables which refer to an instance of data which has been deallocated with **DISPOSE**. This is not a problem as long as the programmer is aware of this fact and does not try to reference part

of the disposed data. In fact these hanging references occur whenever more than one reference variable refers to the same instance of data. For instance, if we have two **RECORD** reference variables, **recA** and **recB**, each referring to the same instance of data and we make the following call:

```
DISPOSE(recA);
```

Then **recA** will contain **NILREC** which indicates that it is not referring to any record, but **recB** will still refer to the now defunct data. If the programmer tries to reference any data referred to by **recB** the results are undefined.

### 5.3 Records

A **record** is a user-defined aggregate data structure composed of some number of built-in types or other data structures.

Each element of a record is called a **field**. In a record, the fields are typically of several different types. One field may be an integer, the next an array of real and the next a string. Instead of using indices to access individual elements, they are referenced by their field name. Individual fields are referenced by appending a period and the name of the field to the name of the record variable. The following **RECORD** type declaration and code illustrate how this is done:

```
TYPE
  positionType = (first, second, third, pitcher,
                 shortstop, outfielder, catcher);
  playerType = RECORD
    Name      : STRING;
    BatAvg    : REAL;
    Team      : STRING;
    Position  : positionType;
  END RECORD;

VAR
  Player1: playerType;
  ...
  Player1.Team      := "Padres";
  Player1.BatAvg    := 0.225;
  Player1.Position  := pitcher;
  Player1.Name      := "Smith"
  ...
```

#### 5.3.1 Using NEW to Allocate RECORDS

The following example uses the **playerType** record type declaration from the earlier example. It shows how a number of records can be allocated and then be organized into a list. There are only two **RECORD** reference variables involved in this code. However, the list could be grown to any length. This is an example of the reference variable's utility.

Note in the example that the `playerType` which is being defined is used as one of its own fields. This is a relaxation of the rule that any identifier used must first be defined. The rule is relaxed in the same way for the object type which will be discussed in the next section.

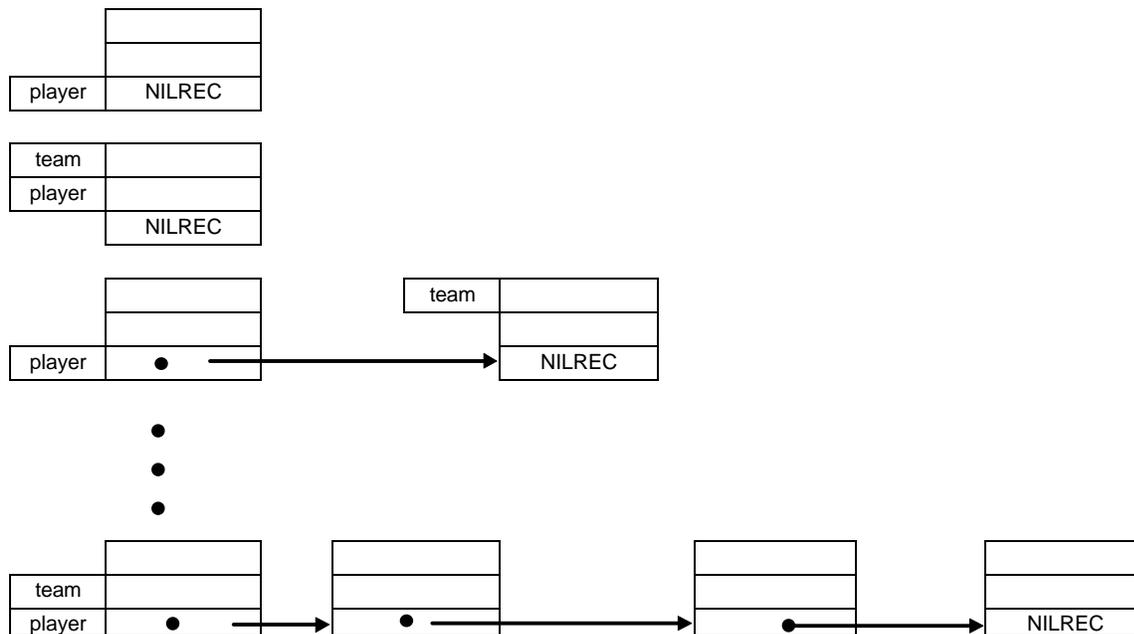
```

TYPE
  playerType = RECORD
    name      : STRING;
    batAvg    : REAL;
    team      : STRING;
    position  : positionType;
    nextPlayer : playerType;
  END RECORD;

VAR
  team,
  player : playerType;
...
NEW(player);           { Allocate memory for a
                        playerType record and make
                        refer to it }
player
team := player;       { now both player & team
                        refer to the same record }
NEW(player);          { Allocate another playerType
                        record }
player.nextPlayer := team; { new record refers to first }
team := player;
NEW(player);          { Allocate another playerType
                        record }
player.nextPlayer := team; { new record refers to second }
team := player;
NEW(player);          { Allocate another playerType
                        record }
player.nextPlayer := team; { new record refers to third }
team := player;

```

The figure below shows the sequence of steps that built the player records:



**Figure 5-5. Linked List of RECORDs**

The code creates four player records and links them together. This is called a linked list. We could make this list arbitrarily long by simply adding new player records and presumably filling them with some useful information as well. We would still need only the two reference variables.

The following sequence of code is worth examining to illustrate again the difference between the behavior of fixed and dynamic data types :

```
NEW(player);
team := player;
```

In the first line a memory area appropriate to a **playerType**, i.e. a player record, is allocated and the variable called **player** is made to refer to it. In the second line the variable called **team** is made to refer to the same memory area. If we contrast this to the way variables for fixed data types work, we see a significant difference:

```
1 - VAR
2 - a, b : INTEGER;
3 - ...
4 - a := 4;
5 - b := a;
```

Here, in the fourth line, the value 4 is stored in a variable called **a**. In the fifth line, the value which is stored in the variable called **a** is copied to a variable called **b**. There are now two different memory locations storing the value 4.

To summarize the difference between variables for fixed and dynamic data types:

- A memory storage area for a fixed data type can only be referred to by one variable. No two variables can refer to the same storage area.
- A memory storage area for a dynamic data type can be referred to by one, many or no reference variables.

Reference variables for records are automatically initialized to the value **NILREC**. This means that they are referring to nothing. The programmer can also explicitly assign the built-in constant **NILREC** to any **RECORD** typed reference variable to indicate that it is currently not referring to a record.

### 5.3.2 ANYREC, ANYOBJ and NILOBJ

The predefined type **ANYREC** is simply a generic type for any record type. It is used in cases where the programmer wants to refer to something without worrying about the type. The value stored in any record type reference variable can be assigned to a variable of type **ANYREC** and vice versa.

There are restrictions on the use of type **ANYREC**. It cannot be used in a call to the built-in procedure **NEW**. This is because there is no information associated with the type **ANYREC** which would tell the system how much and what type of memory to allocate. On the other hand, a record of type **ANYREC** can be passed to the **DISPOSE** procedure.

The key to its usefulness is that it is compatible with all **RECORD** types. This is desirable when writing general purpose procedures or methods which must deal with all record types. Although we haven't studied procedures yet, this example illustrates the point:

```
PROCEDURE SwapRecords(INOUT record1, record2: ANYREC);
VAR
  temp : ANYREC;
BEGIN
  temp      := record1;
  record1  := record2;
  record2  := temp;
END PROCEDURE;
```

The code above will swap two records of any type. MODSIM does not check the type of the records being passed, so only one routine need be written for all possibilities.

Since the record reference variable of type **ANYREC** circumvents MODSIM's strong type checking, it is a two-edged sword. It should be used only where it is really necessary to

defeat type checking as in the above example. Consider what would happen, for instance, if two different record types were passed in to the `SwapRecords` procedure. One would be a player record we have already described and the other would be a personnel record:

```
personType = RECORD
  name    : STRING;
  salary  : REAL;
  age     : INTEGER;
END RECORD;
```

Once they were swapped and we attempted to ask for the age field of a record which was actually a player type, a serious and probably mysterious run-time error would occur. We might simply get a strange answer, or the computer might crash.

The moral of the story is that safety features are there for a reason and should be circumvented reluctantly and with the advance knowledge that trouble might ensue.

We mentioned the built-in constant `NILREC` earlier. `NILREC` is a built-in constant of type `ANYREC`.

There is one other “generic” type which also has a pre-defined constant. `ANYOBJ` is a generic type used to circumvent type checking of objects. The constant `NILOBJ` is of type `ANYOBJ`. It is used to indicate that an object reference variable is referring to no object. These are mentioned here for completeness, but both will be covered in more detail in the section on objects.

### 5.3.3 Operations on RECORDS

Individual fields of records are referenced using the dot notation. A field of a record can be used in the same way as a variable of the same type can be used.

There are a number of operations which can be performed on the record as a whole. In addition to the assignment statement which was discussed above, a record can be used in Boolean expressions with the `=` and `<>` operators.

## 5.4 Arrays

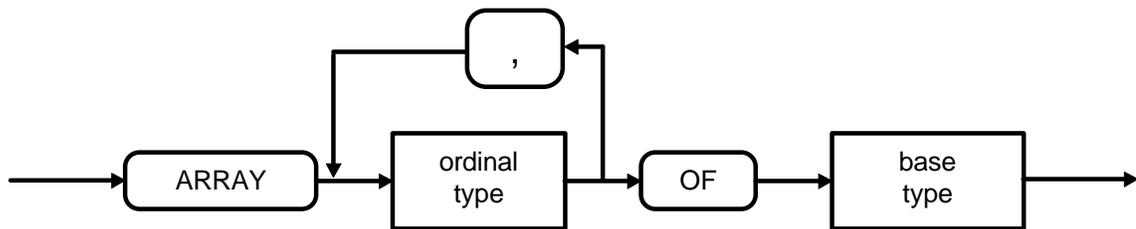
An **array** is an ordered set of data elements referenced using an index or indices. The elements which compose an array can be of any type, but all elements in an array are of the same type. This is known as the base type of the array.

An array can have any number of dimensions. Each dimension is referenced by its own index. The index is defined with a subrange which specifies its lower and upper bounds. Each index may be of any ordinal type.

The power of an array is that a program can compute an index into the array to efficiently reference specific elements in a collection of data.

Like a record, an array structure is dynamically allocated using the **NEW** procedure. Arrays in MODSIM are not only allocated dynamically, but they are also sized dynamically. This is done by passing the **NEW** procedure information about the desired upper and lower bounds of each index using a subrange expression.

This is the syntax for an array type declaration:



**Figure 5-6. Syntax of an Array Type Declaration**

The code below declares two arrays, allocates and sizes them and then assigns values to some of their elements:

```

TYPE
  dayType      = (Sun, Mon, Tue, Wed, Thurs, Fri, Sat);
VAR
  valList      : ARRAY INTEGER OF REAL;
  dailyCount   : ARRAY dayType OF INTEGER;
  n : INTEGER;
  ...
  n := some calculation
  NEW(valList, 1..n)          { allocate space for valList }
  valList[3] := 4.14159;

  NEW(dailyCount, Mon..Fri)  { allocate space for dailyCount }
  } dailyCount[Thurs] := 97;
  ...

```

Since the information about the size of an array is specified dynamically by the program at run-time, this means that declarations for array types are simpler in MODSIM. For instance, in Pascal the variable declaration of `valList` would look like this:

```
valList: ARRAY [1..10] OF REAL;
```

In other words, since it is not possible to dynamically size an array in Pascal, we must specify the bounds of each dimension of the array at the time it is declared. In MODSIM the programmer need only specify the following information to declare an array type:

the type of the index for each dimension and the base type of the array. The information about the bounds for each dimension is provided at run-time, when the array is allocated with the **NEW** procedure.

So the MODSIM equivalent to the above statement would be:

```
valList : ARRAY INTEGER OF REAL;
...
NEW(valList, 1..10);
```

But the bounds of the array are specified using constants, so we have gained little utility when compared to Pascal. If we use variables to specify the upper and lower bounds of the array:

```
NEW(valList, a..b);
```

the utility of dynamically sized and allocated arrays becomes apparent.

When variables or expressions are used to specify the upper and lower bounds, the value of the first must be less than or equal to the second. This is consistent with the concept of subranges.

Multi-dimension arrays are demonstrated below:

```
TYPE
  square      = (blank, X, O); { enumerated type }
  tictacType = ARRAY INTEGER, INTEGER OF square;
              or
  tictacType = ARRAY INTEGER OF ARRAY INTEGER OF square;

VAR
  tictacBoard : tictacType;
...
NEW(tictacBoard, 1..3, 1..3)
tictacBoard[1, 3] := X;
tictacBoard[2][2] := O; { alternative syntax for referenc-
ing                       an element }
...

```

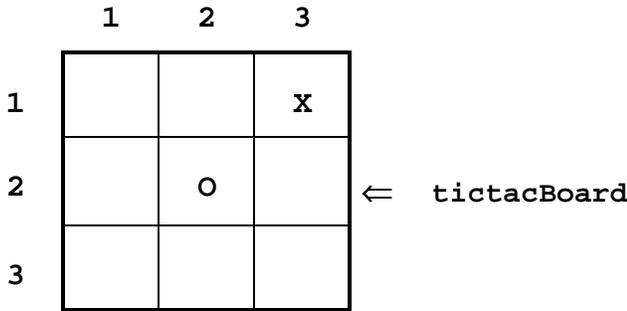


Figure 5-7. An Array

Note that we can either declare a two-dimensional array or we can declare an array of arrays. In either case, the ways in which a single element can be referenced are the same. There is no specified limit to the number of dimensions which can be declared.

5.4.1 Operations on ARRAYS

As demonstrated above, individual array elements are referenced by using the array name followed by the indices in brackets:

```
schedule[month, day] := booked;
```

An element of an array referenced in this way can be used in the same way that a variable of that base type can be used.

There are a number of operations which can be performed on arrays as a whole. One array reference can be assigned to another. Thus, if **schedule2** is an array of the same type as **schedule** above, we can do the following:

```
schedule2 := schedule;
```

After this assignment, both **schedule** and **schedule2** refer to the same array. If we desire a copy of the array, we can do the following:

```
schedule2 := CLONE(schedule);
```

After the **CLONE** is performed, there are two distinct copies of the array each referred to respectively by **schedule** and **schedule2**.

In addition to assignment, an array can be used in Boolean expressions with the **=** and **<>** operators. For two arrays to be considered equal, they must meet the following criteria:

- The number of dimensions must be the same.

- The index type and bounds of each dimension must be identical.
- The contents of each element of the two arrays must be equal.

### 5.4.2 Using the NEW Procedure to Allocate an ARRAY

The **NEW** procedure is used not just to allocate an array, but to determine its size. For each dimension of an array, the **NEW** procedure takes an extra parameter which gives the lower and upper bounds of that dimension. If the elements of an array are a dynamic type (such as a **RECORD** or **OBJECT** type), each entry in the array must be **NEW**ed separately. You do not have to new each entry for fixed allocation types (such as **REAL** or **INTEGER**).

**ARRAY** type reference variables are automatically initialized to **NILARRAY**. **NILARRAY** is a built-in constant which is compatible with all array types.

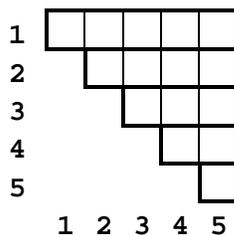
### 5.4.3 Ragged ARRAYS

The **NEW** procedure can be used to allocate an array in piecemeal fashion to build a ragged array. An example of this would be the allocation of a triangular array:

```

...
VAR
  b : ARRAY INTEGER, INTEGER OF REAL;
...
NEW(b, 1..5);
FOR k := 1 TO 5
  NEW(b[k], 1..k);
END FOR;

```



The semantics of the **NEW** procedure when invoked for arrays is that indices can be allocated piecemeal only as we go from left to right in the declaration. Given the array **b** which we described above, we cannot do the following:

```
NEW(b, , 1..10); { illegal allocation! }
```

When an array is allocated with **NEW**, each element of that array is initialized just as individual variables of that base type would be.

The built-in procedure **DISPOSE** is used to deallocate arrays. After **DISPOSE** has been used on an array variable, the value of the reference variable which was passed in is **NILARRAY**.

### 5.4.4 The HIGH and LOW Functions

The two functions **HIGH** and **LOW** can be used to discover the high and low bounds of any dimension of an array:

```
HIGH(<array var>);    ---    LOW(<array var>);
```

```
xArr : ARRAY INTEGER, INTEGER OF REAL;
```

Given the above declaration, **HIGH(xArr)** would return the high bound of the first dimension of array **xArr**. **HIGH(xArr[2])** would return the high bound of the array at position **xArr[2]**. **HIGH** and **LOW** are type compatible with any scalar type variable.

To be less abstract, we can allocate a ragged array as follows:

```
NEW(xArr, 1..3);
NEW(xArr[1], 1..5);
NEW(xArr[2], 2..3);
NEW(xArr[3], 1..4);
```

This would yield the following array:

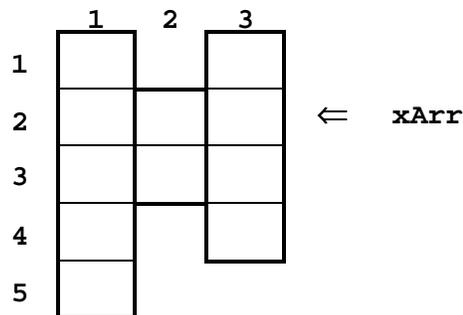


Figure 5-8. A Ragged Array

Having done this we will get the following results from calls to **HIGH** and **LOW**:

<b>LOW(xArr)</b>	<b>⇒</b>	<b>1</b>
<b>HIGH(xArr)</b>	<b>⇒</b>	<b>3</b>
<b>LOW(xArr[1])</b>	<b>⇒</b>	<b>1</b>
<b>HIGH(xArr[1])</b>	<b>⇒</b>	<b>5</b>
<b>LOW(xArr[2])</b>	<b>⇒</b>	<b>2</b>
<b>HIGH(xArr[2])</b>	<b>⇒</b>	<b>3</b>
<b>LOW(xArr[3])</b>	<b>⇒</b>	<b>1</b>
<b>HIGH(xArr[3])</b>	<b>⇒</b>	<b>4</b>

## 5.5 Objects

Object types are mentioned here for completeness. They have fields just as records do. But they are a special type and the entire second section of this manual is devoted to them. Object types can be manipulated in the same way as other dynamic data types. We can have arrays of objects or use object types as fields of records.

## 5.6 Declarations Revisited

Now that structured types have been described, the potential uses of the **TYPE** and **VAR** declaration have been considerably expanded. To tie loose ends together, we will combine several previous declarations together and build more complex structures.

```

CONST
  startYear = 1940;
  thisYear  = 1990;
TYPE
  positionType = (first, second, third, pitcher,
                  shortstop, outfielder, catcher);
  playerType   = RECORD
    name       : STRING;
    batAvg     : REAL;
    team       : STRING;
    position   : positionType;
  END RECORD;
  teamType    = RECORD
    manager    : STRING;
    players    : ARRAY INTEGER OF playerType;
    win, loss  : INTEGER;
  END RECORD;
  leagueType  : ARRAY INTEGER OF teamType;
VAR
  statistics  : ARRAY INTEGER OF leagueType;
  x          : ARRAY INTEGER OF INTEGER;
...

```

```

NEW(statistics, startYear..thisYear);
{ etc, etc.... }

```

### 5.6.1 Anonymous Types

Note that we have declared arrays in two different ways above. In some cases we have explicitly declared an array type and then declared a reference variable of that type. In other cases, such as the variables **x** and **statistics** above, we have simply declared a variable and described the specifics of the array. The declarations for the arrays called **x** and **statistics** use what is known as an **anonymous type** declaration. The array type is bound to the variable declaration and it has no name. Thus, it is an anonymous type.

Arrays and enumerations can be declared this way but records and objects cannot.

Programmers often ask which is the “preferred” way to declare an array—using an explicit type declaration and then declaring a variable of that type, or using the anonymous declaration.

On the one hand, the anonymous type declaration is a shortcut which yields a perfectly usable array variable and saves the extra step of declaring an array type. Anonymous arrays cannot be used as parameters.

On the other hand, declaring an explicit type is a more general approach. Since the type has a name it can be used to specify parameters to procedures. Also assignments of array reference variables can only be accomplished when the two variables involved are of the same exact type. When anonymous types are involved, this condition cannot be satisfied.

There is no simple answer. It depends on how the variable will be used.

## 5.7 Fixed Data Structures

At the beginning of this chapter we made the case for using dynamic data structures. While they are the preferred choice for most programming needs, there will still be occasions when the programmer may prefer to use the fixed data structures. These are the **FIXED RECORD** and **FIXED ARRAY**. Variables of these types are allocated and deallocated automatically by the system in the same way as **INTEGER** and **REAL** typed variables are.

The variables associated with these two types are the same as variables for the other fixed data types; e.g. **INTEGER**, **REAL**, etc. This means that there is one and only one identifier associated with each fixed data structure. It also means that there is no way to string **FIXED RECORDS** together into a linked list since there are no reference variables associated with this type.

The fixed data structure has another feature which distinguishes it from the dynamic data structures. Every element of a **FIXED ARRAY** or **FIXED RECORD** is of a fixed size. This means that its base type or its fields must be of a predetermined size. The fields or base type cannot be a reference variable which refers to some other structure.

The intent is that the information which these two data types carry is completely contained within a known memory area of known size. There are no strings of indeterminate length and no references to other data structures. This means that the **FIXED RECORD** can be used as a template for random access file I/O since each record has a known size and all information pertinent to that **FIXED RECORD** is contained in the record itself. This is an important feature since random access files are characterized by a known, fixed record size.

### 5.7.1 The **FIXED RECORD** Type

The **FIXED RECORD** type can be declared with fields of the following type: **INTEGER**, **REAL**, **CHAR**, **BOOLEAN**, enumeration, subranges, **FIXED ARRAY**, **FIXED RECORD**. This means that it cannot have fields of type: **STRING**, **RECORD**, **ARRAY** or **OBJECT**.

A fixed record can be used in an assignment. A copy of the record on the right side of the assignment statement is made and copied into the record variable of the same type on the left side of the assignment.

A fixed record can be passed as an **IN** parameter to a procedure. A copy is made and passed in. In other words, it behaves like any other fixed data type.

#### 5.7.1.1 Declaring **FIXED RECORD** Types

The syntax used to declare a **FIXED RECORD** type is identical to that used for a **RECORD** type except that the **STRING**, **RECORD**, **ARRAY** and **OBJECT** types cannot be used as fields. Other **FIXED RECORD** types may be used as fields, but the **FIXED RECORD** type cannot use itself as a field, unlike the **RECORD**, since this would be like placing two mirrors facing each other. As with the **RECORD** type, anonymous **FIXED RECORD** declarations are not allowed. This means that a **FIXED RECORD** must be declared as an explicit type and not simply as part of a variable declaration.

### 5.7.2 The **FIXED ARRAY** Type

Like the **FIXED RECORD**, the **FIXED ARRAY** type can be declared with the following as its base type: **INTEGER**, **REAL**, **CHAR**, **BOOLEAN**, enumeration, subranges, **FIXED ARRAY**, **FIXED RECORD**. This means that it cannot use the following as its base type: **STRING**, **RECORD**, **ARRAY** or **OBJECT**.

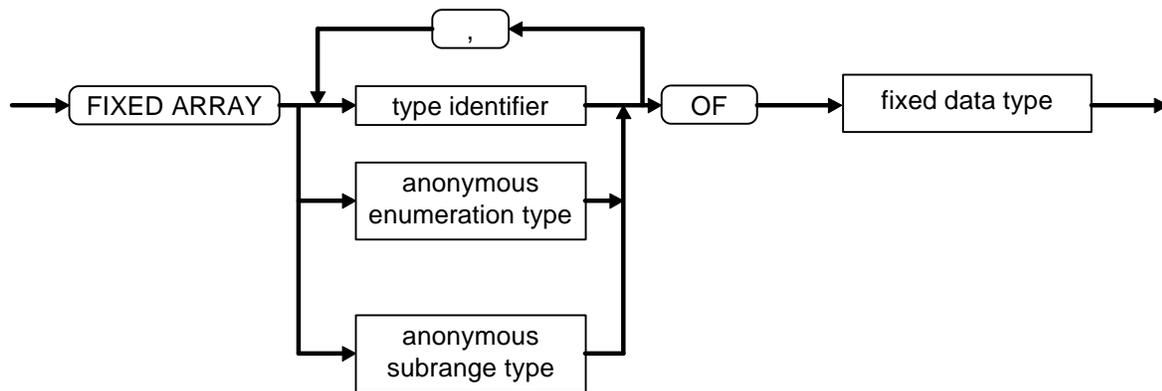
A fixed array **cannot** be used in an assignment statement. If this were allowed, it would mean that the entire contents of a fixed array would have to be copied. Elements of a fixed array can obviously be used in assignment statements. So, if it is necessary to

make a copy of one fixed array to another, this can be accomplished in the traditional way using a loop which iterates through the array copying one element at a time.

When a **FIXED ARRAY** is used as an **IN** parameter, it is treated as if it had been passed with the **INOUT** qualifier. A copy is not passed to the invoked routine. The invoked routine simply refers to the original array. In addition, a **FIXED ARRAY** cannot be passed as an **OUT** parameter.

### 5.7.2.1 Declaring FIXED ARRAY Types

Since the **FIXED ARRAY** type must specify the size of each of its dimensions in the declaration statement, it has a slightly more complex syntax than the **ARRAY** type which need only specify the type of each dimension's index. Anonymous type declarations of the **FIXED ARRAY** are allowed.



**Figure 5-9. FIXED ARRAY Type Declaration**

For each dimension of the fixed array the following information which fixes the size of that dimension must be provided:

- An anonymous subrange
- An anonymous enumeration type
- A type identifier of either an enumeration or a subrange.

It is not possible to declare a ragged fixed array since each element of each dimension must have the same size. Here are some examples of fixed array declarations:

```

CONST
  last = 47;
TYPE
  square      = (blank, X, O);
  tictacType = FIXED ARRAY [1..3], [1..3] OF square;

  dayType     = (Sun, Mon, Tue, Wed, Thurs, Fri, Sat);
  dailyCount  = FIXED ARRAY dayType OF INTEGER;

VAR
  valList : FIXED ARRAY [1..10] OF REAL;
  multi   : FIXED ARRAY [-4..15], [20..last] OF REAL;

```

## 5.8 Referencing the ARRAY and RECORD

Since arrays and records can themselves be composed of structured types, it is worth reviewing how we can refer to a particular element of an array or a particular field of a record in these more complex cases. The following code fragment gives several type declarations and follows them by examples showing how elements of the structures can be referenced. Note that the **ARRAY** and **FIXED ARRAY** types are referenced identically. The **RECORD** and **FIXED RECORD** types are also referenced identically.

```

TYPE
  positionType = ( dayManager, niteManager, clerk1, clerk2 );

  xArrType = ARRAY INTEGER OF REAL;
  personRecType = RECORD
    lastName : STRING;
    age      : REAL;
    payByMonth : xArrType;
  END RECORD;
  jobRecType = RECORD
    persRec      : personRecType;
    jobDescnum,
    monthsInPos : INTEGER
  END RECORD;

  storeType = ARRAY positionType OF jobRecType
...
VAR
  person : personRecType;
  position : jobRecType;
  store : storeType
...
  person.lastName := "Smith";

```

```
    person.payByMonth[5] := 867.75;
...
    position.persRec.lastName := "Jones";
    position.persRec.payByMonth[12] := 965.45;
...
    store[niteManager].persRec.payByMonth[11] := 1256.75;
    store[clerk1] := position;
```





## 6. Statements and Type Compatibility

---

**Statements** are the executable code which perform actions in a program. Statements are used in the block of a program, procedure, or method. The ease or difficulty of coding programs in a language is strongly influenced by the way in which a language handles the concept of a sequence of related statements which perform some action. It is worth digressing here to examine how a number of languages have handled this notion. We will look at the **IF** statement to see how alternative sequences of statements are handled.

In each language we show two **IF** statements. The first has only one statement per choice, the other has a statement sequence for each choice. In all cases, if the Boolean expression is true, Statement1 and Statement2 are executed followed by Statement5. If the Boolean expression is false, Statement3 and Statement4 are executed followed by Statement5.

MODSIM:

```
IF x < 0
    Statement1;
ELSE
    Statement3;
END IF;
Statement5;
```

```
IF y > 0
    Statement1;
    Statement2;
ELSE
    Statement3;
    Statement4;
END IF;
Statement5;
```

*Ada is similar*

Pascal:

```
if x < 0 then
    Statement1
else
    Statement3;
Statement5;
```

```
if y > 0 then
    begin
        Statement1;
        Statement2;
    end
else
    begin
        Statement3;
        Statement4;
    end;
Statement5;
```

*Algol is similar*

```

C:
    if (x < 0)
        Statement1;
    else
        Statement3;
    Statement5;

    if (y > 0)
    {
        Statement1;
        Statement2;
    }
    else
    {
        Statement3;
        Statement4;
    };
    Statement5;

```

The important thing to note is that MODSIM III, like Ada, delimits statement sequences using the control structure itself. Pascal and C use the **begin** “ **end** and { “ }, respectively to delimit these sequences. The significance is that control statements like the **IF** statement have a different appearance in these languages depending on the number of statements included in a choice. In MODSIM the appearance of control structures is consistent in all cases.

## 6.1 Type Compatibility

Before examining statements in detail we will discuss type compatibility in more detail. This manual has emphasized that MODSIM is a strongly typed language. Assignment statements, expressions and parameters passed to procedures and methods are checked for consistency. Here are the specific type compatibility rules which govern the use of variables and constants. Wherever the notation Type1 or Type2 is used we mean a variable or literal constant.

1. Type1 and Type2 are the same type, e.g. they are both of type **INTEGER**.
2. Type1 and Type2 are explicitly defined to be equal in a **TYPE** statement, e.g. **TYPE Type1 = Type2;**
3. Type1 is a subrange of Type2, e.g. Type1 is a subrange [4..23] and Type2 is of type **INTEGER** or vice versa. *Note that **INTEGER** is **not** considered to be a subrange of **REAL**.*
4. Type1 and Type2 are both subranges of the same base type.
5. The **CHAR** type is a conformant type to the **STRING** type. This means that a **CHAR** type may be used anywhere that a **STRING** type is expected. The reverse is not

true. A **STRING** type cannot be used where a **CHAR** type is expected, even if it is of length 1.

6. An object type value may be assigned to an object variable of an underlying type. This subject will be covered in detail in Section II of this manual.
7. Any record type value may be assigned to a variable of type **ANYREC** or vice versa.
8. Any object type value may be assigned to a variable of type **ANYOBJ** or vice versa.
9. Any array type value may be assigned to a variable of type **ANYARRAY**, or vice versa.

### 6.1.1 Type Conversion

By **type conversion**, we mean the conversion of a value from one type to another while maintaining the conceptual meaning. For instance, conversion of **INTEGER** to **REAL** or a **STRING** to an **INTEGER**. In some cases it is possible for a conversion to result in essentially the same resulting value. An example would be the conversion of the integer **14** to a real. The real value would be **14.0**. If, however, we convert the real value **37.557** to an integer we would get **37** if we truncated, or **38** if we rounded.

The type conversion procedures are useful in a number of contexts. In some cases expressions are made up of several different types, so it is necessary to convert some elements to the expected type for the whole expression.

The table below briefly reviews each of the type conversion procedures which MODSIM provides for type conversion. Appendix C contains a detailed description of these procedures and functions.

Procedure / Function	Use
RealNum := FLOAT(IntNum)	INTEGER ⇒ REAL
IntNum := TRUNC(RealNum)	REAL ⇒ INTEGER
IntNum := ROUND(RealNum)	REAL ⇒ INTEGER
Char := CHR(IntNum)	[0..255] ⇒ CHAR
IntNum := ORD(OrdinalVal)	Ordinal ⇒ INTEGER
Ordinal := VAL(OrdinalType, IntNum)	INTEGER ⇒ Ordinal
IntNum := STRTOINT(Text)	STRING ⇒ INTEGER
Text := INTTOSTR(IntNum)	INTEGER ⇒ STRING
RealNum := STRTOREAL(Text)	STRING ⇒ REAL
Text := REALTOSTR(RealNum)	REAL ⇒ STRING
STRTOCHAR(Text, ArrayOfChar)	STRING ⇒ ArrayOfChar
Text := CHARTOSTR(ArrayOfChar)	ArrayOfChar ⇒ STRING

**Figure 6-1. Type Conversion Procedures / Functions**

**Note:** MODSIM does not support the concept of type casting in which the data representation of one type is treated as if it were another type. This concept is extremely machine specific and leads to non-portable code. Instead, the explicit conversion procedures above ensure portability.

Below are some examples of the use of these type conversion procedures and functions:

```

TYPE
  condition = (excellent, good, fair, poor);
VAR
  yesOrNo   : BOOLEAN;
  equipStat : condition;
  letter    : CHAR;
  intNum    : INTEGER;
  realNum   : REAL;
  text      : STRING;
  ...
intNum     := ORD(FALSE);           ⇒ 0
intNum     := TRUNC(4.999);         ⇒ 4
realNum    := FLOAT(intNum);        ⇒ 4.0
equipStat  := VAL(condition, 2);    ⇒ fair
letter     := CHR(65);              ⇒ 'A'
letter     := VAL(CHAR, 66);        ⇒ 'B'
yesOrNo    := VAL(BOOLEAN, 1);     ⇒ TRUE
intNum     := ORD('D');            ⇒ 68
text       := "34.56";              ⇒ "34.56"
realNum    := STRTOREAL(text);      ⇒ 34.56

```

**Figure 6-2. Examples of Type Conversions**

In addition to those library modules discussed above and in earlier paragraphs, the library modules contain many useful functions and procedures which may be explicitly imported for use.

## 6.2 The Assignment Statement

In each of the preceding paragraphs the assignment statement was used in examples. In most cases, this statement simply assigns to the variable on the left, the value obtained by evaluating the expression on the right. The expression on the right of the assignment statement must be type compatible with the variable on the left.

```
variable := expression
```

## 6.3 Program Flow Control

The normal sequence of execution in a program is simple. Each statement is executed in turn. All procedural programming languages provide constructs which can alter this normal flow of program control. MODSIM provides a rich variety of these constructs each of which is described below. MODSIM does not require or provide a GOTO statement.

## 6.4 The IF Statement

The **IF** statement is perhaps the simplest and most often used of control structures. We have already seen examples of its use.

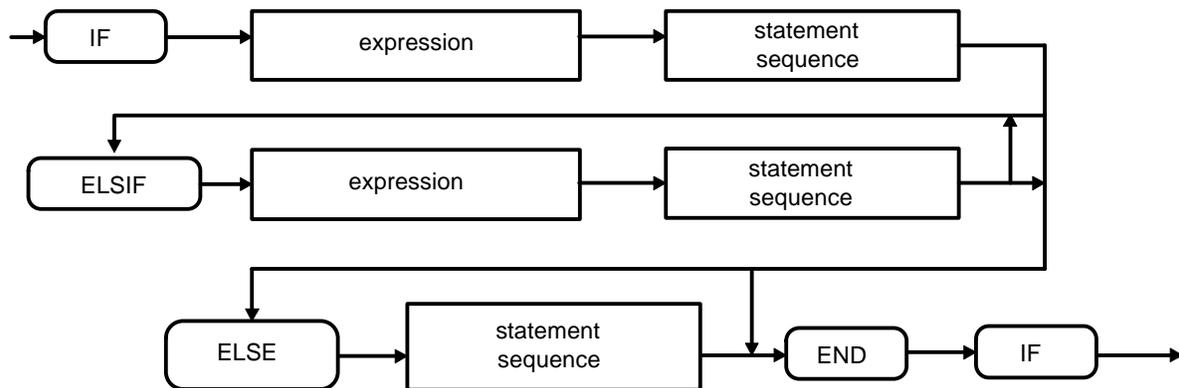
```
IF x < 0
  y := 13;
END IF;

IF y >= 30
  x := x + 5;
ELSE
  x := x - 5;
END IF;
```

The following example illustrates a multiple choice **IF** statement and should be self-explanatory:

```
IF month <= 3
  quarter := "1st Quarter";
ELSIF month <= 6
  quarter := "2nd Quarter";
ELSIF month <= 9
  quarter := "3rd Quarter";
ELSE
  quarter := "4th Quarter";
END IF;
```

Of course, at any place where one assignment statement has been used above, we could substitute a sequence of statements. Here is the formal definition of an IF statement:



**Figure 6-3. Syntax of the IF...END IF Statement**

If the Boolean expression following the **IF** evaluates to **TRUE**, the statements up to the next **ELSIF**, **ELSE**, or **END IF** are executed, after which program control jumps to the statement following the **END IF**. If the Boolean expression is **FALSE**, each **ELSIF**, if any, is evaluated until one is found which evaluates to **TRUE**. If none of the clauses evaluates to **TRUE**, the statements following the **ELSE** are executed.

The **ELSIF** structures and the terminating **ELSE** structure are optional. If there is no **ELSE** in the statement, it is possible for none of the clauses to be **TRUE** and the statement does nothing.

#### 6.4.1 Comparing REAL Values in a Boolean Expression

We mentioned earlier that **REAL** values cannot be represented exactly in digital computers because the binary number system used for internal representation can only approximate the floating-point decimal values being manipulated. Because of this, programmers should always avoid making comparisons with exact **REAL** values.

This topic of numerical representation and approximation errors in computer programs is not one to be taken lightly! The whole discipline of numerical analysis has grown up around the problem. Suffice to say that any programmer wishing to indulge in serious mathematical calculation on a digital computer should consult a good text on the subject.

The small program below illustrates some of the pitfalls associated with **REAL** numbers. The program is followed by its output. The program starts the variable **a** at zero and then adds **0.2** to it a million times:

```

MAIN MODULE sample4;
VAR
  a: REAL;
  k: INTEGER;
BEGIN

```

```

OUTPUT("MODSIM Program 'sample4'");
a := 0.0; k := 0;

WHILE a < 200000.0    a := a + 0.2;
  INC(k);
  IF a = 1.0
    OUTPUT("a should = 1.0 - actually is = ", a);
  END IF;
  IF a = 1.4
    OUTPUT("a should = 1.4 - actually is = ", a);
  END IF;
  IF a = 1.6
    OUTPUT("a should = 1.6 - actually is = ", a);
  END IF;
  IF a = 1.8
    OUTPUT("a should = 1.8 - actually is = ", a);
  END IF;
  IF ( a > 999.99999 ) AND ( a < 1000.00001 )
    OUTPUT("a should = 1000.0 - actually is = ", a);
  END IF;
END WHILE;

OUTPUT("Out of loop - k= ", k);
OUTPUT("Final value of a minus 200000 = ",a - 200000.0);
END MODULE.

MODSIM Program 'sample4'
a should = 1.0 - actually is = 1.000000
a should = 1.4 - actually is = 1.400000
a should = 1000.0 - actually is = 1000.000000
Out of loop - k= 1000000
Final value of a minus 200000 = 0.000003

```

We would expect `a` to have a value of `200,000.0` when we finish, but it actually has a value of `200,000.000003`. In many cases this is an acceptable amount of cumulative error after doing a million calculations.

But there is another, much more serious problem here. Note that the two `IF` statements which are looking for values of `1.6` and `1.8` fail. This means that by the time we have added `0.2` to the variable `a` only eight times, we have already accumulated enough error that the Boolean expression `a = 1.6` evaluates to `FALSE`! On the other hand, the Boolean expression `( a > 999.99999 ) AND ( a < 1000.00001 )` evaluates to `TRUE`. The moral of the story is obvious. When writing a Boolean expression which wants to match a real value, always include an epsilon to allow for error in representation of real numbers.

Another item of interest is the fact that the system output shows a value of `1000.000000` for the variable `a` when we know that the actual value was not exactly equal to `1000.0`. When debugging a program with `OUTPUT` statements, you should be aware that the `OUTPUT` statements will show only a certain amount of precision and they will often round the figure in the last decimal place.

In fact, if we used the more precise `StreamObj WriteReal` method to print out a more exact representation of `a - 200000` at the end of the program, the value printed would have been `0.0000026657650`.

## 6.5 The CASE Statement

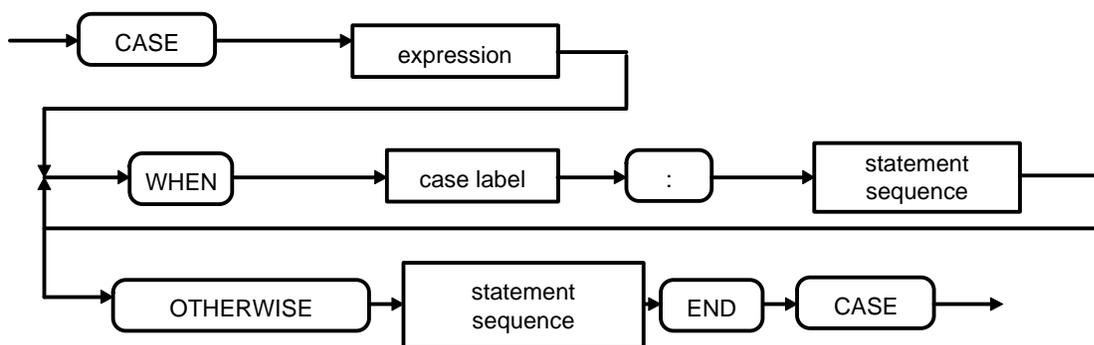
The **CASE** statement provides a convenient method for branching on various values or ranges of values of a single expression. We can rewrite the example used with the **IF** statement as:

```

CASE month
  WHEN 1..3:
    quarter := "1st Quarter";
  WHEN 4, 5, 6:           ← alternative syntax
    quarter := "2nd Quarter";
  WHEN 7..9:
    quarter := "3rd Quarter";
  OTHERWISE
    quarter := "4th Quarter";
END CASE;

```

The syntax is:



**Figure 6-4. Syntax of the CASE .. END CASE Statement**

The type of the expression in a **CASE** statement can be any ordinal type, or **STRING**. The expression cannot be of type **REAL**. The **case label** cannot be a variable. The **OTHERWISE** clause is optional. If the **OTHERWISE** clause is not included in a **CASE** statement and none of the stated choices are selected, then a run-time error will be issued. **REAL** is not allowed in **CASE** statements because comparisons of exact **REAL** values are not reliable in binary computers where **REAL** numbers are represented by approximations. The case labels in a **WHEN** clause need not be contiguous. Here are some more examples:

```

CASE month
  WHEN 1, 3, 5, 7..8, 10, 12:
    days := 31;

```

```

WHEN 4, 6, 9, 11:
  days := 30;
OTHERWISE
  IF leapYear
    days := 29;
  ELSE
    days := 28;
  END IF;
END CASE;

```

## 6.6 Iterative Statements

MODSIM III provides a rich variety of loop statements: **WHILE**, **FOR**, **REPEAT**, and **LOOP**. In MODSIM the **EXIT** statement may be used in any of these loop constructs to leave the loop and continue execution at the first statement which follows the loop construct.

### 6.6.1 The WHILE Statement

The **WHILE** statement is a loop which is repeated 0 or more times. As long as the Boolean expression at the head of a **WHILE** construct remains true, the enclosed statements are executed. For example:

```

n := 2;
WHILE n < 5
  OUTPUT("n = ", n);
  INC(n); ← { same as n := n + 1 }
END WHILE;

```

This loop would output the following:

```

n = 2
n = 3
n = 4

```

The enclosed statements will be repeated until **n** equals 5. They would not have been executed at all if **n** was greater than or equal to 5 to begin with. In other words, if the Boolean expression evaluates to false on entry to the **WHILE** statement, the enclosed statements are never executed. The syntax of the **WHILE** statement is:

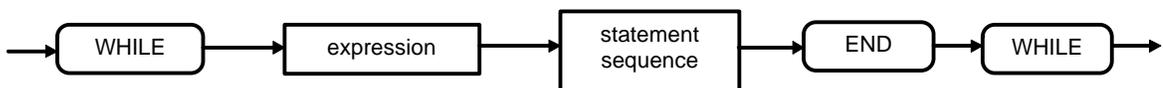


Figure 6-5. Syntax of the WHILE .. END WHILE Statement

## 6.6.2 The REPEAT Statement

The **REPEAT** statement is a loop which is repeated one or more times. The Boolean expression is located at the end of the statement and is not evaluated until the body of the loop has been executed at least once. An example is:

```
REPEAT
  OUTPUT("This statement will print at least once.");
  INC(A); ← { same as A:= A + 1 }
UNTIL A > 5;
```

The above statement will repeat until **A** is greater than 5. If **A** is greater than 5 before the loop begins (or greater than 4, in this case), then the statement sequence will be executed only once. The syntax of the **REPEAT...UNTIL** statement is:

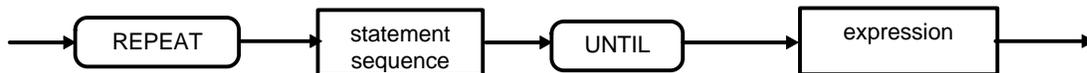


Figure 6-6. Syntax of the REPEAT...UNTIL Statement

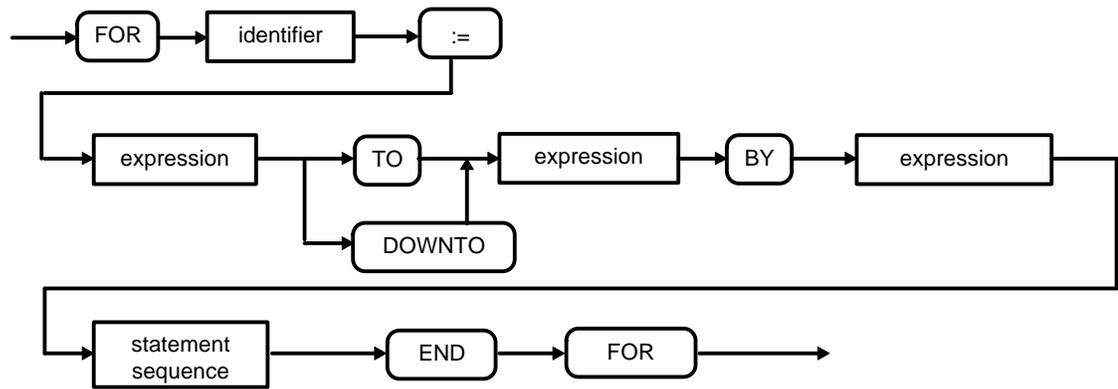
## 6.6.3 The FOR Statement

The **FOR** statement is a loop statement which increments (or decrements) a variable by some integral value until it has iterated through a specified range, each time repeating the enclosed statements. An example is:

```
FOR n := 1 TO 5
  OUTPUT("The next number is:", n);
END FOR;
```

The loop variable may be of any ordinal type. The loop may step by increments different than one by adding the optional **BY** statement. It may also step backward by replacing **TO** with **DOWNTO** (leaving the increment positive). The stepping value may be any expression compatible with type **INTEGER**. For example:

```
FOR Letter:= 'z' DOWNTO 'a' BY 2
  OUTPUT("stepping down the alphabet by two. ", Letter);
END FOR;
```



**Figure 6-7. Syntax of the `FOR ... END FOR` Statement**

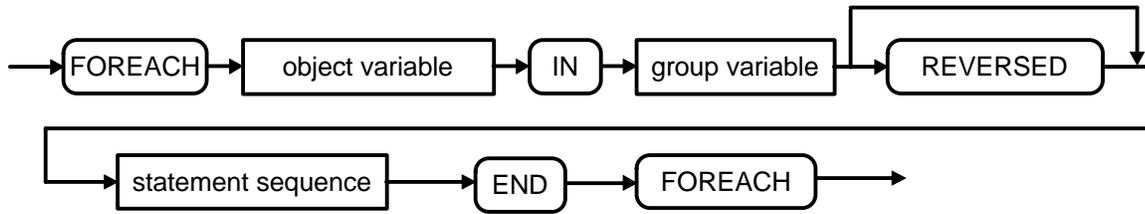
It is important to know the semantics of the **FOR** statement. The expressions which give the starting point, the end point and the increment are evaluated only once, at entry to the **FOR** loop. Changing any of the variables involved in specifying the start, end or increment of the loop variable will have no effect on the loop while the loop is executing. Changing the value of the loop variable itself will not alter the execution of the loop, although it will affect the value of the loop variable for that iteration.

The bottom line is that all bookkeeping in **FOR** loops is done external to the loop. If it is necessary to leave the loop before it has finished iterating, the programmer can always use the **EXIT** statement under control of an **IF** statement. If it is necessary to alter the increment value or the end point of a loop while it is executing, the programmer must use one of the other loop constructs.

#### 6.6.4 The **FOREACH** Statement

The **FOREACH** statement provides an easy mechanism for iterating over the contents of a group object (defined in the MODSIM library module **GrpMod**) or any object derived from a group object. The **FOREACH** statement will iterate over the members of the group even if a group contains the same object more than once.

The form of the statement is as shown in figure 6-8.



**Figure 6-8. The FOREACH Statement**

In figure 6-8 **object variable** is a variable declared to be an object type compatible with the object types stored in the **group variable** type. The **group variable** may be a variable declared to be a type derived from any of the group objects defined in 'GrpMod' including **QueueObj**, **StackObj**, **RankedObj**, **BTreeObj** and their respective statistical definitions.

The **object variable** will contain the first member of the group upon entrance to the loop. If the group is empty, the statements within the **FOREACH** block will not be executed at all. Upon each iteration of the loop the **object instance variable** will be updated to contain the next member of the group, even if the current **object variable** value has been removed from the group. If the optional keyword **REVERSED** is included in the statement, the members of the group will be iterated from the last member to the first.

In addition to objects defined from 'GrpMod', it is possible to use the **FOREACH** statement to iterate over user defined groups. To do this, the user group object must define the methods: **First**, **Next** (for forward iteration) and/or **Last**, **Prev** (for reverse iteration), e.g.:

```

MyGroupObj = OBJECT
  ASK METHOD First : Aobj;
  ASK METHOD Next (IN obj : Aobj) : Aobj;
  ASK METHOD Prev (IN obj : Aobj) : Aobj;
  ASK METHOD Last : Aobj;
END OBJECT;
. . .
PROCEDURE iterate (IN grp: MyGroupObj);
VAR
  a : Aobj;
BEGIN
  FOREACH a IN grp
    {Use 'a' }
  END FOREACH;
END PROCEDURE;

```

The **FOREACH** statement can also be used to iterate over the **RECORD** groups defined in 'ListMod'.

### 6.6.5 The EXIT Statement

The **EXIT** statement immediately transfers control to the first statement after a loop construct. The **EXIT** statement can be used with any loop construct.

### 6.6.6 The LOOP Statement

The **LOOP** statement simply loops forever. The only way to stop this loop is to use the **EXIT** statement. This loop construct is quite versatile since the **EXIT** statement(s), and any corresponding Boolean expression, may be located at the beginning, end, or anywhere within the body of the loop. For example:

```

LOOP
  OUTPUT("bread");
  INC(IntVar);           { same as IntVar:= IntVar + 1 }
  IF IntVar > 5
    EXIT;
  END IF;
  OUTPUT("cheese and turkey");
END LOOP;

```

The example above will always create a valid sandwich (bread at both ends) as long as **IntVar** is initially less than 5.

### 6.6.7 The Other Control Statements

There are three other control statements which will be covered later. For completeness we will summarize their use here.

The **WAIT** statement is used to elapse simulation time. Its syntax is similar to the **IF** statement. It will be covered in Section III of this manual. The **RETURN** statement is used to end the execution of a procedure or function before reaching the end of the routine. In the case of functions, it is also used to specify the value which will be returned. It will be covered in the next chapter

Finally, the **TERMINATE** statement is used in simulations to end the execution of a chain of method calls. It will be covered in Section III of this manual.



## 7. Procedures and Functions

---

Procedures are named blocks of code which may be invoked from other parts of a program. Every procedure must have a **BEGIN**, even if it is empty. Procedures have a parameter list which is used to communicate information to and from the procedure. Certain types of procedures may act like expressions since they yield a value when executed. Terms used to describe this type of functionality in other languages are:

ROUTINE, SUBROUTINE, SUBPROGRAM, FUNCTION

There are two kinds of procedures in MODSIM. Proper procedures are those which do not yield a value when executed. Function procedures yield a value when executed and can be used like a variable in expressions. They are defined in much the same manner, using the reserved word **PROCEDURE**. When a function procedure is defined, its return type is specified. The return type of a function procedure can be any type except for the **FIXED ARRAY** type. MODSIM supports recursive procedure calls.

Throughout the text we will use the terms, procedure and function, in the following ways:

- Procedure:** Refers to either a proper procedure or a function procedure. If the distinction is important, the correct one will be specified.
- Function:** Refers to a function procedure.

Procedures have optional parameter lists which are used to communicate data between the invoker and the procedure. The number, type and order of parameters in the procedure declaration and the procedure call of user-defined procedures must match exactly.

Several built-in procedures relax these rules. For instance, the **MAXOF** procedure takes any number of arguments, either **INTEGER** or **REAL**, and returns the value of the largest argument. The **OUTPUT** and **INPUT** procedures take any number and several types of arguments in any order.

Within a procedure, the parameters which have been communicated through the parameter list are treated like variables. When declaring a procedure, the programmer specifies, for each parameter in the parameter list, the following three pieces of information:

- An identifier which will name the parameter
- The type of parameter
- The direction in which information will flow: **IN**, **OUT** or **INOUT**

Whenever a procedure is invoked, the parameters are type-checked for consistency with the declaration. There are two kinds of parameters:

**Formal parameters:** These are the parameters which are detailed in the declaration of the procedure.

**Actual parameters:** These are the parameters which are actually passed in to a procedure when it is invoked.

### 7.1 Formal Parameter Qualifiers: **IN**, **OUT**, **INOUT**

MODSIM, like Ada, requires a distinction between input and output parameters. The formal parameter qualifier affects how the variables are treated and assists in documenting program code.

Each parameter must be declared with one of the three possible qualifiers:

**IN:** The value may only be passed **in** to the procedure from the caller (pass by value). When the **IN** qualifier is specified, a copy is made of the value and the copy is passed in to the procedure. This means that the actual parameter and the formal parameter are two separate copies. If the formal parameter is changed inside the procedure this will have no effect on the actual parameter. The **IN** qualifier may be used with all types. In all cases except for the **FIXED ARRAY**, a copy of the data stored in the variable is made and passed in. The **FIXED ARRAY** is treated as if it had the qualifier **INOUT**. See the note below about subtleties of behavior when dynamic types are passed with the **IN** qualifier.

**INOUT:** The value may be passed in either direction (pass by reference). This means that no copy is made. The formal parameter is simply an alias for the actual parameter. If the formal parameter is modified inside the procedure, this change will affect the actual parameter.

**OUT:** The **OUT** qualifier operates identically to the **INOUT** qualifier with one extra property. The variables passed with the **OUT** qualifier are re-initialized as they are passed in. This enforces the notion that information only flows in one direction.

Note that when a reference variable for a dynamic data type is passed by value using the **IN** qualifier, a copy of the actual parameter, and not the data structure, is made and passed as a formal parameter to the routine being called. The formal parameter refers to the same structured data instance as the actual parameter did. So the behavior in this case is the same as if the **INOUT** qualifier had been used, as long as no assignments are made to the actual parameter. This means that any changes made to the structured data instance will be reflected outside of the procedure call.

Constants and literals cannot be used as **OUT** or **INOUT** parameters for the same reason that they cannot be used on the left side of an assignment statement. The following example illustrates the point and gives us a preview of a procedure declaration:

```
PROCEDURE increment(INOUT n : INTEGER);
BEGIN
  n := n + 1;
END PROCEDURE;
```

If we were to call the procedure with a literal, e.g. `increment(3)`, this has the same effect as trying to do the following assignment: `3 := 3 + 1`.

Procedures are defined in MODSIM much as they are in Pascal, and Ada. They are specified as declarations before the body of the main program. User-defined procedures and functions are invoked in exactly the same way as the built-in procedures. A user-defined procedure with the same name as a built-in procedure will replace the built-in procedure.

Methods, as mentioned in the introduction, are the procedures which an object can execute as part of its behavior. We will cover methods in detail in Section II of this manual. However, it is worth noting that the way in which methods are defined is nearly identical to the way in which procedures are defined.

There are two aspects of procedures to be covered in this chapter. How to call or invoke procedures and how to declare them. Since invoking procedures is straightforward, and there are already many built-in procedures to use, this topic will be covered first.

## 7.2 Invoking Procedures

We have already used several built-in procedures in examples of code. One of these, the **OUTPUT** procedure, is a built-in procedure which prints a list of variables and constants. To invoke the procedure we simply use its name and supply parameters on which it can operate:

```
OUTPUT("Hello there");
```

**Note:** We do not have to explicitly use the term “CALL” to invoke a procedure.

Functions are invoked by placing them in the same context as an expression. In other words, anywhere an expression is allowed a function may be used or included as a term of the expression.

```
n := ROUND(35.5556)
```

or

```
OUTPUT(ROUND(35.5556))
```

### 7.3 Declaring Procedures

The following trivial program shows how a procedure is declared and then used.

```

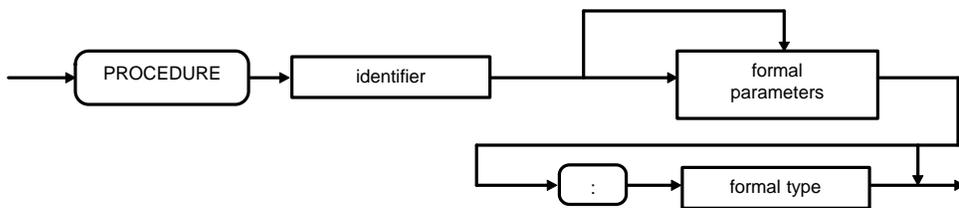
MAIN MODULE Sample5;
VAR
  textLine : STRING;

PROCEDURE PrintIt(IN Str: STRING);
BEGIN
  OUTPUT(Str);
END PROCEDURE;

BEGIN
  textLine := "This is a VERY simple program.";
  PrintIt(textLine); { Call the procedure defined above }
END MODULE.

```

A **PROCEDURE** heading identifies the beginning of a procedure block. The syntax of a procedure heading is:



**Figure 7-1. Syntax of a Procedure Declaration**

When the optional **function result type** is specified in the procedure declaration, it indicates that this is a function procedure, and a **RETURN** statement must be specified in the body of the procedure to exit and return the specified function result.

### 7.4 RETURN Statement

The **RETURN** statement has two purposes:

- It can be used to exit from a procedure and return to the invoker before the end of the procedure is reached.
- It must be used in a function procedure to communicate the return value to the invoker.

To illustrate the points we have made so far, here are two procedures which use the Pythagorean theorem to compute the length of a hypotenuse given the length of two sides of a right triangle. **Hyp1** is implemented as a proper procedure and returns the answer through its parameter list. **Hyp2** is implemented as a function procedure which returns the answer as a value.

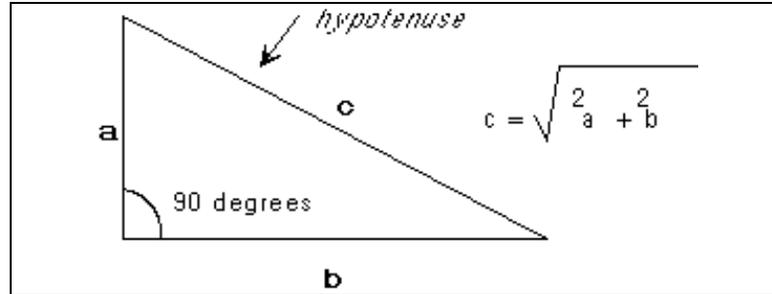


Figure 7-2. A Right Triangle

```

PROCEDURE Hyp1(IN a, b : REAL; OUT c : REAL);
BEGIN
  c := SQRT( a*a + b*b );
END PROCEDURE;

PROCEDURE Hyp2(IN a, b : REAL) : REAL;
BEGIN
  RETURN SQRT( a*a + b*b );
END PROCEDURE;

```

To use these we could do the following:

```
Hyp1(3.0, 4.0, answer);    or    answer := Hyp2(3.0, 4.0);
```

So far we have shown only the simplest of procedure declarations. In actual use, a procedure may declare its own local variables. The formal definition of a procedure declaration states that we use a procedure heading followed by a subblock. What is this subblock? It is almost identical to the syntax for the block which makes up the main program. The main difference is that it may not contain the declarations of other procedures, objects or types. This means that we can declare any constants and variables needed by the procedure, and then any sequence of executable statements.

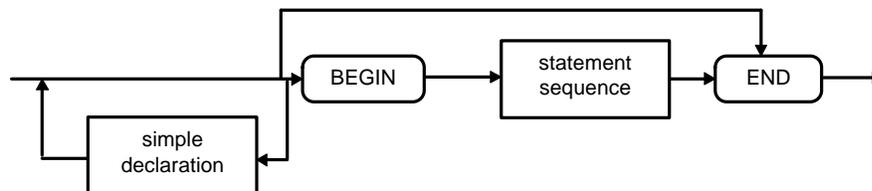


Figure 7-3. Syntax of the Procedure Block

## 7.5 The FORWARD Qualifier

There may be times when a procedure needs to be used before it has been defined. When it is desirable to do so, simply put the procedure heading in the declaration section followed by the reserved word **FORWARD** ahead of the first place where it will be used.

Later, in the same module, provide the full declaration in the usual way. If we had wanted to provide a forward declaration of `Hyp1`, this is how it would look:

```
PROCEDURE Hyp1(IN a, b : REAL; OUT c : REAL); FORWARD;
```

## 7.6 Procedures With Empty Parameter Lists

A procedure with no parameters can be declared and invoked with or without using an empty parameter list. We can illustrate this by showing how each of the procedures whose headings are listed below would be called:

<pre>PROCEDURE Proc1;       or PROCEDURE Proc1();  PROCEDURE Proc2() : INTEGER;       or PROCEDURE Proc2 : INTEGER;</pre>	<p>← procedure headings</p>
<pre>Proc1;       or Proc1(); n := Proc2();       or n := Proc2;</pre>	<p>← procedure invocations</p>

**Figure 7-4. Empty Parameter Lists**

## 8. Modules

---

Up to this point we have explained MODSIM's language features from the viewpoint of a traditional language such as Pascal, in which all of the components of a program are found in one file which is compiled as a unit.

One of MODSIM's strong points is its modular structure which allows programs to be constructed from library modules. Any part of a program can import types, variables, constants and procedure definitions from these library modules as needed. Each module can be compiled separately to facilitate program maintenance and reduce development time.

### 8.1 Facts About Modules

Since so many features of modules are interdependent, it is difficult to cover the topic in a strictly sequential fashion. To make things easier, we will list a number of brief facts about modules and then discuss them at greater length.

- Every MODSIM program must contain a **MAIN** module.
- As the name implies, there can only be one **MAIN** module in a program.
- Every **MAIN** module must have a **BEGIN**, even if it is empty.
- **MAIN** modules and **IMPLEMENTATION** modules may have **ModInit** procedures. **ModInit** procedures may be used to initialize modules before the first statement of a program is executed.
- Each module is named using a standard identifier.
- A program may consist of any number of modules. Each module is stored in a separate file.
- There are three types of MODSIM modules: **MAIN**, **DEFINITION**, and **IMPLEMENTATION**.
- Any module can be compiled separately.
- A library consists of two modules: **DEFINITION**, and **IMPLEMENTATION**. Each is named with the same identifier.
- Any constant, type, variable or procedure declared in a **DEFINITION** module may be imported by other modules.
- Any constant, type, variable or procedure declared in a **DEFINITION** module is implicitly visible in the accompanying **IMPLEMENTATION** module.
- There can be no executable code in a **DEFINITION** module.
- If a procedure or object method is defined in a **DEFINITION** module it must be coded in the accompanying **IMPLEMENTATION** module.

- Nothing in an **IMPLEMENTATION** module is visible anywhere else, including within that library's **DEFINITION** module.
- Nothing can be imported from an **IMPLEMENTATION** module.
- Nothing can be imported from a **MAIN** module.
- Anything imported into a **DEFINITION** module is implicitly visible in that library's **IMPLEMENTATION** module.
- **IMPORT** statements must be the first statements in any module. They may be preceded only by comments.

## 8.2 The **IMPORT** Statement

We discuss the **IMPORT** statement first because it is possible to use MODSIM without taking advantage of user-defined libraries; yet MODSIM, itself, provides a number of built-in libraries from which the user may wish to **IMPORT**.

The **IMPORT** statement can be used in any kind of module to selectively import any constant, type, variable or procedure definitions from a **DEFINITION** module. Here is an example of an import from MODSIM's built-in Utility and Math modules:

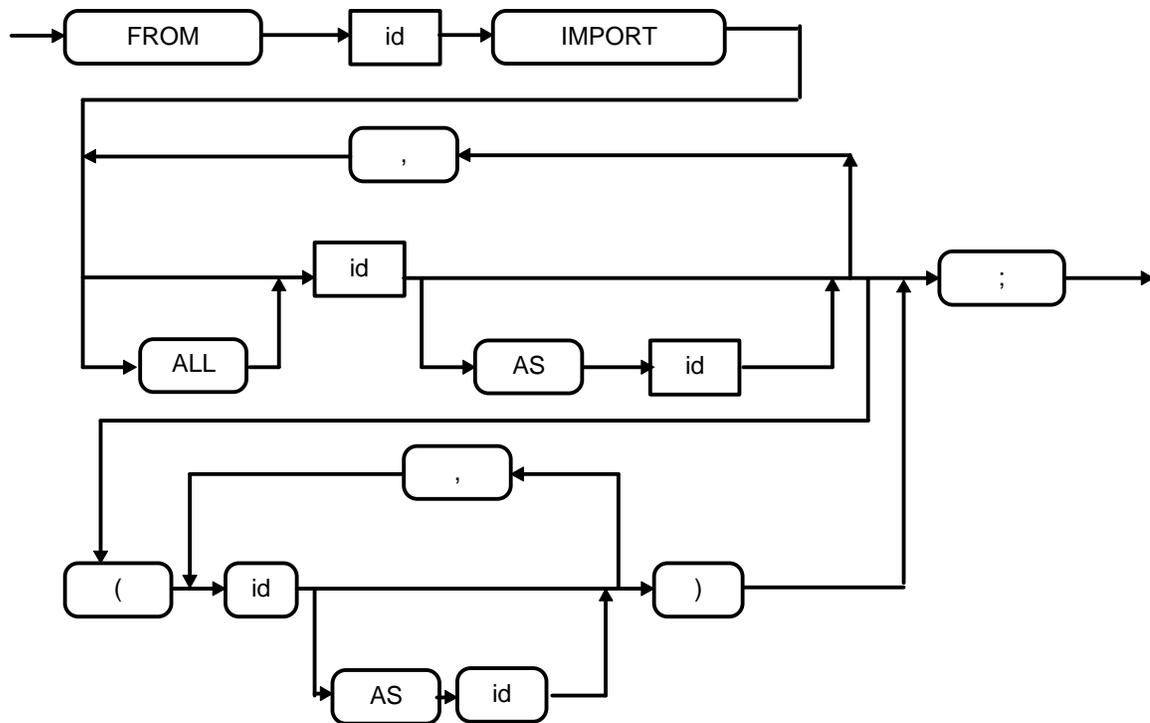
```

FROM UtilMod IMPORT, GetComputerType;
FROM MathMod IMPORT SIN, COS, pi;
           or
FROM MathMod IMPORT SIN AS sine, COS AS cosine, pi;

```

In the **IMPORT** statements above, **pi** is a constant and the remainder of the imported constructs are procedures. Note that we simply use the identifier of the construct whose definition is to be imported. If the identifier which names the imported construct would conflict with an identifier already in use, the imported construct can be renamed using the **AS** clause to rename it. As the example above shows, this renaming feature can be used arbitrarily for aesthetic reasons as well. The renaming of the variable applies only within the module which has imported the variable.

ules



**Figure 8-1. Syntax of an IMPORT Statement**

Note in the syntax diagram above that MODSIM allows considerable flexibility in the importing of enumerated types. Here is an enumeration which is defined in **IOMod**, MODSIM's built-in input-output module:

```

TYPE
  FileUseType = ( Input, Output, InOut, Append, Update );
  
```

To illustrate ways in which enumerations can be handled in an **IMPORT** statement we will show four different ways to handle imports from that definition:

```

1 - FROM IOMod IMPORT ALL FileUseType;
2 - FROM IOMod IMPORT FileUseType( Input, Output );
3 - FROM IOMod IMPORT FileUseType( Input AS in, Output);
4 - FROM IOMod IMPORT ALL FileUseType( Input AS in );
  
```

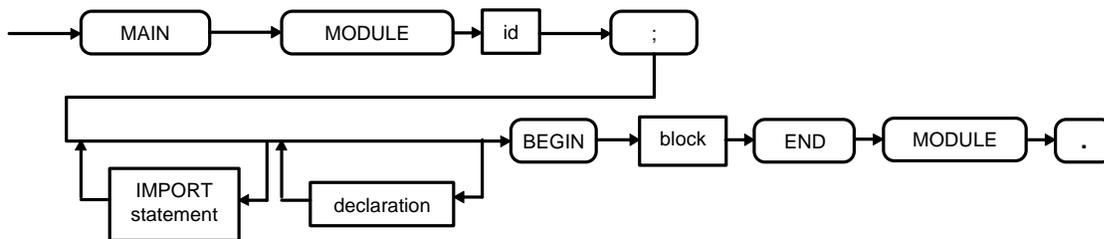
1. Imports the type definition for the enumeration and all of its individual enumeration values.
2. Imports the type definition for the enumeration and two specific enumeration values.
3. Imports the type definition for the enumeration and two specific enumeration values. It renames one of the enumeration values which is imported.

4. Imports the type definition for the enumeration and all of its individual enumeration values. It renames one of the enumeration values which is imported.

An attempt to import one of the enumeration values without its parent type will be treated as an error.

### 8.3 MAIN Module

The **MAIN** module contains the main routine of the program. It is the only required module.



**Figure 8-2. Syntax of a MAIN Module**

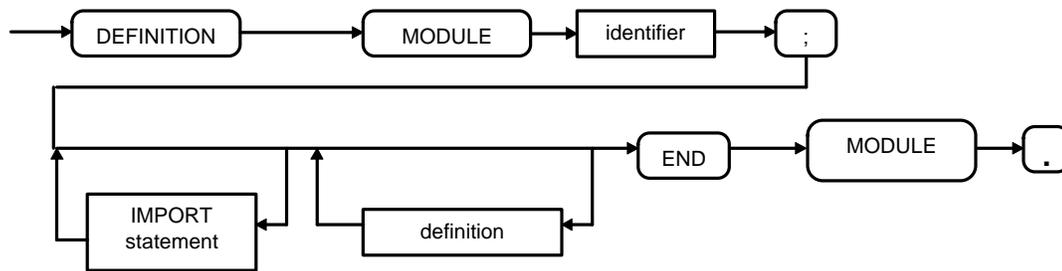
The **MAIN** module consists of a module heading followed by optional **IMPORT** statements and a block.

### 8.4 DEFINITION Module

A **DEFINITION** module contains a set of definitions for export to some other module or modules. These definitions must be explicitly imported by the other modules which need them. If a procedure or object method is defined in a **DEFINITION** module, then a corresponding **IMPLEMENTATION** module must be provided and an implementation of any procedures must be provided. An **IMPLEMENTATION** module is needed *even if it is empty* except for the header and ending statements. The **IMPLEMENTATION** module contains the executable code for procedures and methods defined in the **DEFINITION** module.

Any construct defined in a **DEFINITION** module will automatically be visible in the corresponding **IMPLEMENTATION** module. In other words, definitions are implicitly imported to the corresponding **IMPLEMENTATION** module.

ules



**Figure 8-3. Syntax of a DEFINITION Module**

When a procedure is defined in a **DEFINITION** module, only the procedure heading is listed. The actual procedure declaration takes place in the corresponding **IMPLEMENTATION** module. As an example, if we were to place the procedure **Hyp1** in a **DEFINITION** module called **Trig** so it would be available for export, we would do the following:

```

DEFINITION MODULE Trig;

CONST
  goldenSection = 3.0 / 5.0;

PROCEDURE Hyp1(IN a, b : REAL; OUT c : REAL);

END MODULE.
  
```

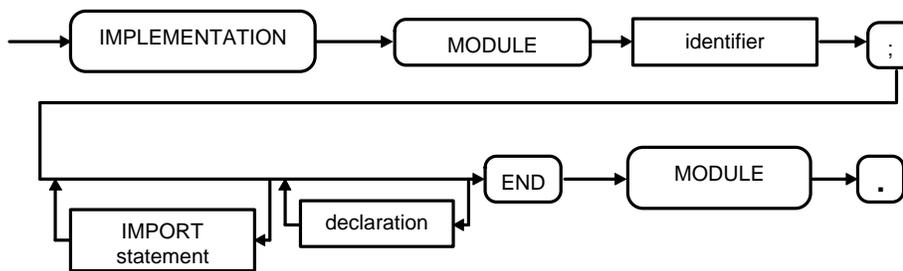
Note that we also defined a constant which will be available for export. The layout of a definition module is nearly the same as the layout of a block except that procedures are not fully defined. Only their heading is listed. We can include constant, type, variable and procedure declarations.

### 8.4.1 Cycle Dependencies

MODSIM III does not require users to identify cyclic relationships between definitions in various modules. Users can simply import types, constants and variables as needed without regard to such dependencies.

### 8.5 IMPLEMENTATION Module

An **IMPLEMENTATION** module contains the actual code for the objects and procedures whose interfaces are specified in the **DEFINITION** module. In other words it contains the full declarations for any procedures or object methods whose headings are listed in the **DEFINITION** module. The **IMPLEMENTATION** module must have the same name as its corresponding **DEFINITION** module. Any constants, types or variables declared in the **DEFINITION** module are automatically visible in the **IMPLEMENTATION** module.



**Figure 8-4. Syntax of an IMPLEMENTATION Module**

An **IMPLEMENTATION** module may also contain declarations of constants, types, variables or procedures which will be used strictly in that module to implement its capabilities.

An **IMPLEMENTATION** module may import any constructs it needs from other **DEFINITION** modules. If a construct is imported into a **DEFINITION** module it will automatically be visible in the corresponding **IMPLEMENTATION** module. In other words, items defined in a **DEFINITION** module are implicitly visible in the accompanying **IMPLEMENTATION** module, and items imported into the **DEFINITION** module are implicitly visible in the **IMPLEMENTATION** module.

## 8.6 The ModInit Procedure

**MAIN** modules and **IMPLEMENTATION** modules are allowed to have a **ModInit** procedure. **ModInit** procedures are procedures guaranteed to be executed before the program's first statement is executed. They are a useful tool for initializing modules.

To define a **ModInit** procedure, one need only to include a definition for a procedure named **ModInit** somewhere in the module. The **ModInit** procedure must have zero arguments.

No guarantee is given as to the order in which the **ModInit** procedures of a set of included modules will be executed.

## 8.7 File Naming Conventions for Modules

MODSIM III's compilation manager and other system utilities expect that a certain naming convention will be used for files which contain modules. MODSIM source files are expected to have the extension **.mod**. The file name is composed by preceding the module name with an **M** for **MAIN** module, **D** for **DEFINITION** module and **I** for **IMPLEMENTATION** module. The **.mod** extension is then added. The examples below show how this is done:

Module Name	File Name
MAIN MODULE Alpha	MAlpha.mod
DEFINITION MODULE Beta	DBeta.mod
IMPLEMENTATION MODULE Beta	IBeta.mod
C++ code for Beta	Beta.cpp

**Figure 8-5. File Naming Conventions for Modules**

Note that the naming conventions extend to C++ source code files which may be part of a library. In other words, part of the implementation code may be provided in C++. When this is done, naming conventions for the procedure headings in the **DEFINITION** module must be followed. These naming conventions will be covered shortly.

The file systems on some computers allow only very short file names to be used. The PC FAT file system is one of these. It allows filenames to be at most eight characters in length. For systems such as these, the file names for modules having long names will be truncated. Thus, **MAIN MODULE AlphabetSoup** must be stored in a file named **MAlphabe.mod**. This means that no two modules may have names with the same first seven characters. Even though their modules names were unique, they could end up with identical file names. This restriction does not apply to Windows NT or Windows 95 operating systems when they are not using the MS-DOS based FAT file system.

Finally, file names for MODSIM modules are case sensitive on computers which recognize case in file names, which includes UNIX, Windows NT and Windows 95.

## 8.8 Including C or C++ Code in a MODSIM Program

The preferred way to include C/C++ routines in a MODSIM program is to provide a procedure heading in the **DEFINITION** or **IMPLEMENTATION** module which is followed by the keyword **NONMODSIM**. The C/C++ code is then provided in a file which follows the naming conventions outlined above.

The following table lists the MODSIM type and the matching type in C/C++ code. The file 'modsim.h', which is part of the MODSIM distribution should be checked in case a specific implementation of MODSIM deviates from this table.

MODSIM	C/C++
INTEGER	long
REAL	double
BOOLEAN	char
CHAR	unsigned char
STRING	char * to C
ANYREC	MS_RECORD
ANYOBJ	MS_OBJECT
enumeration	int
ARRAY OF CHAR	string from C

**Figure 8-6. MODSIM Types vs C/C++ Types**

Below is a MODSIM **DEFINITION** module followed by C++ and C files which include the implementation code. The procedure heading for a routine which will be provided in C++ is followed by the keyword **NONMODSIM**. In addition the **NONMODSIM** keyword can be followed by an optional string literal which distinguishes between different language linkage conventions. At present the strings "C" and "C++" are valid.

**Note:** **NONMODSIM** without any string literal defaults to C++.

```

DEFINITION MODULE Sample6;                               ← file DSample6.mod
PROCEDURE foo1(IN x : REAL) : INTEGER; NONMODSIM;
PROCEDURE foo2(IN x: REAL) : INTEGER; NONMODSIM "C";
END MODULE;

IMPLEMENTATION MODULE Sample6;                            ← file ISample6.mod
END MODULE;

#include <modsim.h>                                       ← file Sample6.cpp
MS_INTEGER foo1(MS_REAL x)
{
  MS_INTEGER n;
  cout << "x = " << x << "\n";
  return n;
}

#include <modsim.h>                                       ← file Sample6.c
MS_INTEGER foo2(MS_REAL x)
{
  MS_INTEGER n;
  printf ("x = %f\n", x);
  return n;
}

```

**Note:** The examples above include 'modsim.h'. This is the preferred way to interface MODSIM to C++ code as it helps ensure that C++ code will link without problems, and will be compatible with future releases of MODSIM.

ules

When passing parameters to a C/C++ routine it is important to note that **INOUT** and **OUT** parameters are handled as pointers. Here is a MODSIM procedure heading and the matching C++ function to illustrate the point:

```

DEFINITION MODULE Sample7;

    PROCEDURE Proc7(IN    x : REAL;
                   INOUT n : INTEGER) : CHAR; NONMODSIM;
END MODULE.

MS_CHAR Proc7(MS_REAL x, MS_INTEGER* n)
{
    ...
}

```

Because the MODSIM **STRING** type is fully dynamic, it carries with it some bookkeeping information for memory management. Because of this, it must be interfaced to C++ code carefully. This means that strings can be passed between C++ and MODSIM only in routines which are identified to the compiler as **NONMODSIM**. This ensures that memory management bookkeeping is done correctly.

The MODSIM **STRING** type can be passed directly into a C++ **char\*** variable. The MODSIM **STRING** type is null terminated. Strings passed in this way must not be disposed of, or modified by the C++ routine.

Strings coming from C++ to MODSIM must be passed into a MODSIM **ARRAY OF CHAR** or **FIXED ARRAY [0..n] OF CHAR**. The **ARRAY OF CHAR** can then be converted into a MODSIM **STRING** type using the built-in procedure **CHARTOSTR**. The C++ string must be null terminated. A C++ string must never be passed directly into a MODSIM **STRING** type variable. If this is done, the results are unpredictable, but invariably bad since the C++ strings lack the memory management fields used in MODSIM strings.

The example below illustrates passing a MODSIM string to C++ code and passing a C++ string back to MODSIM:

```

PROCEDURE foo(IN str: STRING); NONMODSIM;      ← file Itest.mod
TYPE
    Arr = ARRAY INTEGER OF CHAR;

PROCEDURE Test(IN cstr: Arr);
VAR
    str : STRING;
BEGIN
    str := CHARTOSTR(cstr); { convert to MODSIM string }
    OUTPUT("C++ string: ", str);
END PROCEDURE;

```

```

PROCEDURE TestStrings;
VAR
  str: STRING;
BEGIN
  str := "a MODSIM string";
  foo(str); { pass string to C++ routine foo }
END PROCEDURE;

#include <modsim.h>                               ⇐ file test.cpp

extern void test_Test(MS_CHAR* astr);
void foo(MS_STRING str)
{
  cout << "MODSIM string: " << str << "\n";

  char* cstr = "a C++ string";
  test_Test((MS_CHAR*)cstr); // calling MODSIM PROCEDURE Test
}

```

It is also possible, but not recommended, to use a C++ routine without following the naming conventions for file names and without identifying the procedure as **NONMODSIM** in a **DEFINITION** module. In this case, the link would have to be handled by the user. The user would then have to use the naming conventions followed by the compiler. The routine **Proc7** would then look like this:

```

MS_CHAR Sample7_Proc7(MS_REAL x, MS_INTEGER *n)
{
  ...
}

```

The MODSIM compiler uses the following conventions:

```

types :           <modulename>_<typename>
global variable : <modulename>_<variablename>
procedure :      <modulename>_<procedurename>
method name :    <methodname>_
local variables : <variablename>_

```

Although MODSIM III has no inherent limit on the length of identifiers, linkers often do have limits. Many machines have linkers with a 32 character limit. Users should keep this in mind. If the combined length of the module name and identifier exceeds 29 characters (32 minus the 3 underscore characters), it is possible that names which are unique in the MODSIM code would not be unique when rudely truncated to 32 characters by the machine's linker. When this happens, the linker may warn that duplicate symbols have been encountered.

## **Section II. Object-Oriented Programming**



## 9. Objects in MODSIM III

---

Objects in MODSIM are dynamically allocated data structures coupled with routines, called methods. The fields in the object's data structure define its state at any instant in time while its methods describe the actions which the object can perform. The values of the fields of an object can be modified only by its own methods, but the values of the fields can be “read” by any part of the program. We refer to the fields and methods of an object collectively as its **properties**.

The programmer declares an object type by specifying its fields and its methods. The **OBJECT** type is consistent in behavior to other types. Its closest relative is the **RECORD**. Object types can be used as fields of records or as fields of other objects. An array of objects can be declared. The object type can be used as a parameter in procedures and methods.

The **OBJECT** type is a dynamic data type, which means that it behaves in the same way as arrays and records behave. Object instances are allocated and deallocated by the programmer using **NEW** and **DISPOSE**.

We introduced a number of these aspects of the object data type in the first section of this manual. This was to draw attention to the many similarities between object types and the traditional data types. But in some ways it is a large leap from simpler types to object types because objects add several new programming capabilities to the programmer's toolbox. Although these new capabilities are more easily classified as evolutionary than revolutionary, the effect of their introduction on programming technique and style has been profound.

Some of the new capabilities provided by objects are:

- **Encapsulation of data and code:** Tying together the fields which describe the object's state with the procedures (called methods) which define its behaviors. Controlling access to the fields.
- **Inheritance:** Once an object type has been defined, new types can be defined based on the existing type. Each descendant in the hierarchy can add its own fields and method definitions to those of its ancestors.
- **Message passing:** An object's methods are invoked by sending a message to the object asking it to perform a specific method.
- **Polymorphism:** Allowing differing object types in a hierarchy to share the same method name but provide their own implementation of that method. This results in a generic invocation producing different behaviors appropriate to the object being referenced.

- **Hierarchical types:** A descendant is type compatible with any of its ancestors.

When a variable of type **OBJECT** is declared, the result is a **reference variable**. The object's reference variable behaves in the same manner as reference variables for arrays and records.

## 9.1 Object Type Versus Object Instance

Once an object type has been declared, instances of that type of object are created using the built-in procedure **NEW**. The object type serves as a template or specification from which individual object instances are created by **NEW**. Each object instance has its own set of fields of the type and number described in the object type declaration.

## 9.2 Scope of an Object's Fields

Within an object's methods, all of its fields are visible. In other words, the object's fields are global in scope to the object's methods. Variables declared within an object's methods, of course, are local to that particular method.

## 9.3 Object Type Declaration / Object Declaration

An object type declaration is similar to a record type declaration in that each includes a list of fields, but an object type declaration also includes the headings for any methods the object will define. A **METHOD** is nearly identical to a **PROCEDURE** except that it is encapsulated in an object:

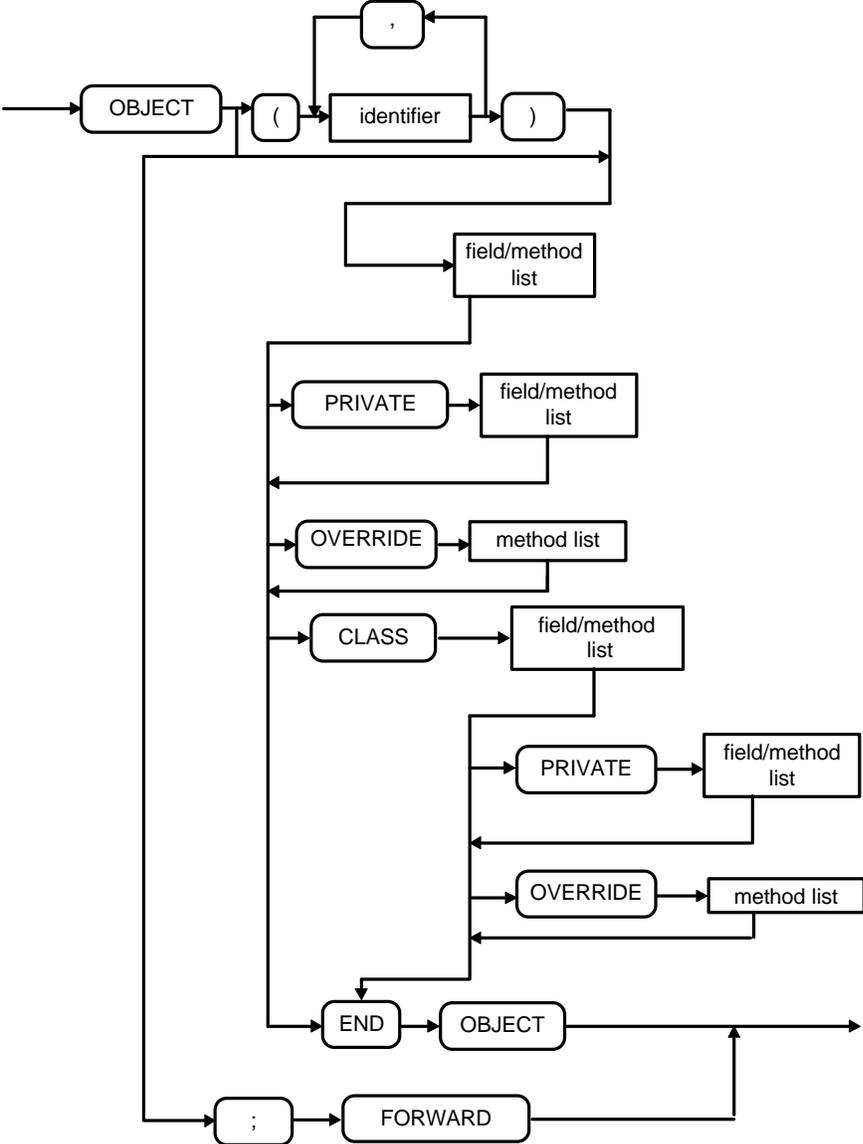
```

TYPE
  VehicleObject = OBJECT                               ← object type declaration
    course      : [ 0 .. 359 ];
    speed       : INTEGER;
    position    : PositType;
    ASK METHOD GoTo(IN destination : PositType);
  END OBJECT;

...
OBJECT VehicleObject;                                ← object declaration
  ASK METHOD GoTo(IN destination : PositType);
  BEGIN
    ...
    executable code goes here
    ...
  END METHOD;
END OBJECT;

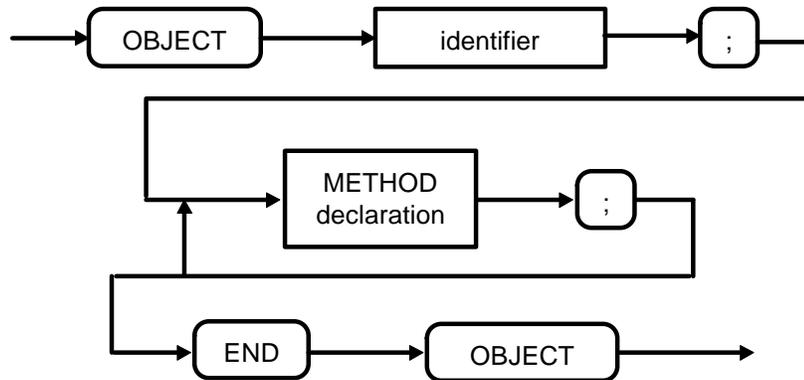
```

Object types are declared in the **TYPE** section of a module. For each **METHOD** heading which is mentioned in the **object type declaration** there must be a full **METHOD** declaration in a separate **object declaration**. The object declaration is a separate, named block which contains all of the **METHOD** declarations. So there are two elements involved in defining an object.



**Figure 9-1. Syntax of an Object Type Declaration**

The **object type declaration** contains the “interface” to the object. All fields are defined here and the heading for each **METHOD** is listed.



**Figure 9-2. Syntax of an Object Declaration**

The **object declaration** is a separate and distinct block which bears the same name as its corresponding object type declaration. Its only purpose is to hold the full **METHOD** declarations for the object type.

It is reasonable to ask why the object declaration has been split into two pieces. In most large MODSIM programs, the object type declarations would be placed in a **DEFINITION** module and the object declaration, which contains the declaration of the methods, would be placed in an **IMPLEMENTATION** module. This allows the definition, or interface, of an object to be made visible in a **DEFINITION** module while hiding its implementation in the **IMPLEMENTATION** module. In a program which consists of only a main module, the object type declaration would come first, followed later by the object declaration.

A method which returns a function result is referred to as a **function method**. A method that does not return a function result is known as a **proper method**. In the previous example, **GoTo** is a proper method.

The method may optionally include a list of parameters. The syntax of the parameter list, the type of the parameters, the parameter qualifier and the type of a function method, follow the same rules as procedures.

#### 9.4 METHOD Declarations

Other than the use of the keywords **ASK METHOD**, **TELL METHOD** or **WAITFOR METHOD** instead of **PROCEDURE**, methods are defined using a syntax similar to procedure declarations. Their heading is listed in the object type declaration and their full declaration is placed in the object declaration block.

The declaration of the method, or the body of the method which contains the executable code, is found in the corresponding object declaration block.

After an object type has been declared, its methods must be declared in a corresponding object declaration block. Typically, the object type declaration is placed in a **DEFINITION MODULE**, and the corresponding object declaration is placed in an **IMPLEMENTATION MODULE**.

For example, the corresponding object declaration for the `VehicleObject` type declaration could look like this:

```
OBJECT VehicleObject;
  ASK METHOD GoTo(IN destination : PositType);
  BEGIN
    implementation code ...
  END METHOD;
END OBJECT;
```

## 9.5 Scope of Fields and Variables in Objects

We mentioned that objects are typically declared and used in a library module. This is a good place to digress a bit and review the scope of fields and variables from the perspective of objects defined in modules:

- Any variable declared in a module external to any object type declaration or object declaration block is global to the entire module. There will be only one copy of the variable. The variable is visible to the methods of every object instance as well as every procedure in the module. If the variable is declared in the **DEFINITION** module instead of the **IMPLEMENTATION** module, it will also be visible to any other module which imports it.
- Any field declared within an object type declaration will be visible in the usual sense within that object's methods. Each instance of an object type has its own separate copies of all of the fields. In other words there will be a separate copy of each field for each object instance which is created. From outside the object, we can **ASK** the object for the value of any of the fields, but we cannot directly change their value with an assignment.
- Any variable declared within the body of a method will be visible only within that method. There will be a unique copy of that local variable for each invocation of that method.

## 9.6 Object Reference Variables

The declaration of an object type implicitly defines a new data type of the same name, known as the **reference type**. Variables declared to be of a reference type are known as **reference variables**. When a reference variable for an object is declared, it is initialized to `NILOBJ`. `NILOBJ` is a built-in constant which is analogous to `NILREC` and `NILARRAY`. It means that the object reference variable is not referring to any object.

Objects are like records in that they are created and destroyed dynamically, at run-time, by the built-in procedures **NEW** and **DISPOSE**. Each object which is created by **NEW**, according to the type specification for that object type, is called an **object instance**. Like arrays and records, once an object type has been defined, multiple instances of objects of that type can be dynamically created.

Objects are like arrays and records which have been dynamically created with the **NEW** procedure. They are not tied to a particular variable. An object instance can be referred to by one, many or no reference variables. Likewise, a reference variable can refer to one object or to **NILOBJ**, i.e. no object.

A reference variable contains a **reference value** which identifies a particular instance of an object type. Programs will often have many instances of a given object type. All of these instances share an identical structure, but each will have distinct values in its fields to represent its current state.

For example, defining the object type **AircraftObject** implicitly defines a corresponding reference type **AircraftObject**:

```

TYPE
  AircraftObject = OBJECT
    altitude : INTEGER;
    wingAC   : AircraftObject;
    TELL METHOD Land;
  END OBJECT;

VAR
  Squadron: ARRAY INTEGER OF AircraftObject;
  .
  .
PROCEDURE AirControl;
VAR
  Tiger20, Puma33: AircraftObject;
BEGIN
  ...
END PROCEDURE;

```

In the above example, the field **wingAC**, the elements of the global variable **Squadron**, and the local variables **Tiger20** and **Puma33** are all reference variables for objects of type **AircraftObject**. Note that the declaration for **AircraftObject** uses the type **AircraftObject** as one of its own fields. The identifier is used before its own definition is complete. This is one of two contexts in the language where this rule is relaxed. The other context is in fields of records.

Reference variables can be used in a manner similar to other types of variables. The variable declaration:

```
Squadron : ARRAY INTEGER OF AircraftObject;
```

is an example.

Fields of objects containing reference variables can indicate relationships between objects. In the example above, the field `wingAC` is a reference variable of type `AircraftObject` which is used to access an aircraft's accompanying wing aircraft, which is another object instance of type `AircraftObject`.

## 9.7 Class Variables (Fields) and Methods

Class variables and methods are specific to an object declaration just as object fields and methods are. The difference is that only a single instance of a class variable ever exists, as opposed to object variables (fields) which are duplicated for each instance creation of the object type. If an object type declaration contains class variables and/or fields, then the object description itself may be considered an entity or *meta* class. Class variables and methods may be referenced directly from the object type itself.

Class variable and method declaration is part of an object's declaration. The keyword **CLASS** follows all instance field and method declarations including any **OVERRIDE** or **PRIVATE** sections. Following the **CLASS** statement, class fields (variables) and methods may be declared. This section may contain an **OVERRIDE** and **PRIVATE** section as well. Class variables and methods are inherited just as instance fields and methods are inherited, and class methods may be overridden to modify their behavior as appropriate for the derived object type.

For example:

```
PlaneObj = OBJECT
  LoggedHours   : REAL;
  Altitude      : REAL;
  Destination   : CityType; (* declared elsewhere *)
  TELL METHOD FlyTo(IN city : CityType);

  CLASS
    PlaneQueue  : QueueObj;
    ASK METHOD CreateQueue;
  END OBJECT;
```

Suppose we desired an object type that could limit the number of instances ever simultaneously in existence; for example, a pool of resources where each member of the pool is a resource object itself. We could declare the following object type:

```
ResourcePoolObj = OBJECT(ResourceObj)
  OVERRIDE
    ASK METHOD ObjInit;
    ASK METHOD ObjTerminate;
  CLASS
    numberAllowed : INTEGER;
    ASK METHOD SetNumberAllowed(IN num : INTEGER);
  END OBJECT;

OBJECT ResourcePoolObj;
  ASK METHOD ObjInit;
```

```

BEGIN
    DEC(numberAllowed);
END METHOD;

ASK METHOD ObjTerminate;
BEGIN
    INC(numberAllowed);
END METHOD;

ASK METHOD SetNumberAllowed(IN num : INTEGER);
BEGIN
    numberAllowed := num;
END METHOD;
END OBJECT;

```

In our initialization statements we can then instantiate the object type so that it will keep track of the availability of resource objects:

```
ASK ResourcePoolObj TO SetNumberAllowed(34);
```

Notice that the type name itself is used as a reference for the **ASK** construct. This is only allowed for class variables and methods. If an instance of **ResourcePoolObj** had been created it could be used to reference any class variables or methods as well. Also, note that within an object type's methods class variables and methods may be referenced and class variables assigned as if they were regular fields or methods of the object type.

When we are ready to create a new instance of **ResourcePoolObj** we check the **numberAllowed** field and proceed accordingly:

```

...
VAR
    Resource : ResourcePoolObj;
BEGIN
    IF ASK ResourcePoolObj numberAllowed > 0
        NEW(Resource);
    ELSE
        (* appropriate processing *)
    END IF;
...

```

In addition to ASK methods, both TELL and WAITFOR methods may be declared as CLASS methods.

## 9.8 Object Type Checking and the ANYOBJ Type

The special type **ANYOBJ** provides an escape from strict type checking of reference variables in MODSIM. It is analogous to the type **ANYREC** for records.

A variable of type **ANYOBJ** can be used to hold a reference value of any object type. No type-checking is performed during an assignment to or from a variable of type **ANYOBJ**.

It is a way to circumvent MODSIM's type checking. There may be circumstances when this is necessary. Usually the relaxation of type checking is used when building general purpose procedures or methods which are designed to operate on any object instance, without regard to its type. For example:

```
PROCEDURE SwitchObjects(INOUT firstObj : ANYOBJ;
                       INOUT secondObj: ANYOBJ);
VAR
  temp : ANYOBJ;
BEGIN
  temp      := firstObj;
  firstObj  := secondObj;
  secondObj := temp;
END PROCEDURE;
```

The procedure above will exchange objects of any type. This saves us the chore of writing a separate switch procedure for each object type. This ability is used in many of the library modules. Of course, as with the **ANYREC** type, this powerful capability is a two-edged sword. Since it circumvents strong type checking of parameters, it leaves the careless programmer vulnerable to run-time errors which may be difficult to debug.

Just as it is not possible to reference fields of a record which has been assigned to a variable of type **ANYREC**, it is not possible to reference fields of an object which has been assigned to a variable of type **ANYOBJ**. This is because the **ANYOBJ** typed variable carries no information about the structure of the object it is referring to.

## 9.9 Allocating and Deallocating Objects

An object instance is allocated by calling the standard procedure **NEW**, which takes as its argument a reference variable of the desired type. The reference value for the object instance is returned in the argument, and each field of the instance is automatically initialized as appropriate.

For example:

```
VAR
  Tiger20: AircraftObject;
  ...
  NEW(Tiger20);
```

The above call allocates an object instance of type **AircraftObject** and returns its value in the reference variable **Tiger20**.

**Note:** It is not sufficient to simply declare the reference variable to obtain access to an object instance. The reference variable contains **NILOBJ** (analogous to **NILARRAY** and **NILREC**) until it is assigned a reference value by a call to **NEW** or by an assignment statement.

An object instance is deallocated by calling the standard procedure **DISPOSE**, which takes as its argument an object reference variable.

For example:

```
DISPOSE(Tiger20);
```

deallocates the object instance which was allocated in the previous example.

An object stored in a variable of type **ANYOBJ** can be passed to **DISPOSE**. Since objects carry their actual type information with them, **DISPOSE** is able to identify the object and handle it properly.

### 9.10 ObjInit & ObjTerminate

Some object types may require initialization of their instances before they are used. If a method called **ObjInit** has been defined for an object type, then the method will be invoked automatically by **NEW**.

A complementary method called **ObjTerminate** is used to perform “cleanup” before deallocating objects. This method is called automatically, if it exists, by **DISPOSE** before it deallocates the object instance.

### 9.11 ObjClone

The built-in **CLONE** function can be used to make a copy of an object instance. When **CLONE** is passed an object instance to copy, it will do the following:

1. Allocate space for a new object instance of the same type passed in.
2. Copy the values in the fields of object instance passed in to the new copy.
3. Invoke the new object instance's **ObjInit** method if one exists.
4. Invoke the object type's **ObjClone** method, if one exists.

The **ObjClone** method is analogous to the **ObjInit** and **ObjTerminate** methods. The **ObjClone** method can be used to perform any more complex behaviors which the user wants to associate with the copy.

If the programmer overrides an existing **ObjClone** method, the overridden method must be invoked with the **INHERITED** statement to ensure that all behaviors associated with copying defined by ancestors are carried forward.

## 9.12 Proto Objects

MODSIM provides a special object type declaration facility that allows programmers to design and implement objects in a general purpose and high-level way, while allowing the use of these objects to be refined as appropriate to individual applications. For example, group objects (queues, stacks, btrees, etc) are provided in the MODSIM runtime library support. These objects have been implemented so that any object may be placed in such a group. When used in applications, groups generally contain only one kind of object or objects derived from a single ancestor object type. However, the compiler cannot ensure that only instances of the “correct” object type are added to a user’s groups since it has no way of knowing which types to include or exclude. For this purpose proto-objects are employed.

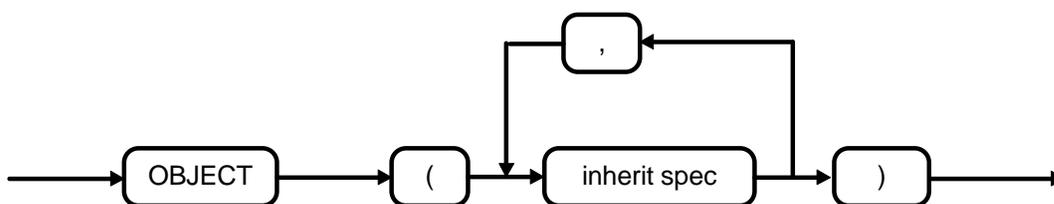
A proto-object type declaration is almost identical to a regular object type declaration. But, with proto-object types the user can indicate that certain field, method return and formal argument types are *replaceable*. Replaceable types may be any object type including the built-in generic **ANYOBJ** or the built-in generic **ANYREC**. A replaceable type may only be replaced with a compatible type. In the case of **ANYOBJ**, any object type may be used. Similarly, for **ANYREC** any record type may be used. If the replaceable type is a specific object type, then only object types derived from the replaceable type may be used.

Any field, method parameter or method return type may be declared as replaceable. An example of a proto-object declaration is shown below:

```
Obj = OBJECT
  field : #Aobj;
  ASK METHOD foo(IN a : #Aobj) : #Aobj;
END OBJECT;
```

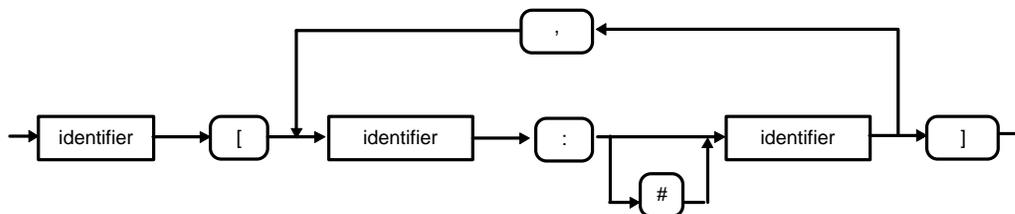
Here, 'field' is an **Aobj** which is replaceable and the method 'foo' has a parameter of type **Aobj** and returns an **Aobj**, both of which are replaceable.

To derive a substitute replaceable type the syntax shown in figure 9-3 is used.



**Figure 9-3. Syntax for Substituting a Replaceable Type**

where 'inherit spec' is:



**Figure 9-4. Syntax for 'inherit spec'**

For example:

If we want to have a queue that could contain only `PlaneObj`s or object types derived from planes, we can restrict `QueueObj` so that the compiler will ensure that only the correct type of object can be added to the queue:

```
PlaneQueueObj = OBJECT(QueueObj[ANYOBJ:PlaneObj])
  (* any additional fields and methods needed *)
END OBJECT;
```

Remember that the replacement type must be compatible with the replaceable type. Any use of the new type will cause the compiler to ensure that the correct type object (or record) instances are passed to the queue object's methods and that its return values (from `Remove()`, for example) are assigned to variables of a compatible type.

The following example illustrates how to derive and substitute multiple levels of proto-objects:

```
PlaneObj = OBJECT
  (* field and methods *)
END OBJECT;

JetObj = OBJECT(PlaneObj)
  (* fields, methods and overrides *)
END OBJECT;

PlaneQueueObj = OBJECT(QueueObj[ANYOBJ:#PlaneObj])
  (* notice that QueueObj is a PROTO with the *)
  (* replaceable type ANYOBJ *)
  (* this object substitutes PlaneObj for ANYOBJ -*)
  (* thereby restricting this queue to hold only *)
  (* PlaneObj things. By placing '#' before *)
  (* PlaneObj in the substitution, subsequent *)
  (* derivations may replace PlaneObj with their *)
  (* own object derived from a PlaneObj *)
END OBJECT;
```

```

JetQueueObj = OBJECT(PlaneQueueObj[PlaneObj:JetObj])
  (* in this declaration a further refined queue *)
  (* is defined by substituting JetObj for *)
  (* PlaneObj. since JetObj is NOT preceded *)
  (* by '#' no further substitution would be *)
  (* enabled. however, if desired a '#' could *)
  (* precede JetObj in order to allow *)
  (* further refinements. *)
END OBJECT;

```

Proto-object types may be used “as is.” That is, it is not required that users derive their own type or substitute types for the replaceable types. If the proto-object type is used without replacements, the object type will use the replaceable type(s) as a default. Also, if a proto-object type specifies more than one replaceable type, the user may selectively replace zero, one or more of them.



# 10. Methods and Fields of Objects

---

The object type declaration must be accompanied by an object declaration which includes the full declaration of the object's methods. The heading of each method is listed in the object type declaration, but the full definition of the method is done in the object declaration block.

A **METHOD** differs from a **PROCEDURE** in several important ways:

- The method is tied to an object. It can only be invoked by sending a message to an object instance requesting that the method be performed.
- Unlike procedures, there can be any number of methods named with the same identifier. Each one can have different implementation code. They can be distinguished from each other because each is tied to a different object type.
- Some methods can elapse simulation time but procedures cannot.

Methods come in three forms: **ASK** methods, **TELL** methods, and **WAITFOR** methods. The form is specified when they are declared. There are important distinctions between **ASK**, **TELL**, and **WAITFOR** as pertains to simulation, but for now we will simplify the distinction somewhat. Since **WAITFOR** methods may only be called as part of the **WAIT FOR** statement we will defer showing examples of **WAITFOR** method invocations until we discuss the **WAIT FOR** construct in Section III.

## 10.1 Invoking an Object's ASK and TELL METHODS

Methods differ in one crucial way from procedures; they are always invoked with reference to a specific object:

<pre>ASK   TELL <i>object</i> [ TO ] <i>method</i> [ (<i>parameter list</i>) ]</pre> <p style="text-align: center;">or</p> <pre>value := ASK <i>object</i> [ TO ] <i>method</i>([<i>parameter list</i>]);</pre>
---

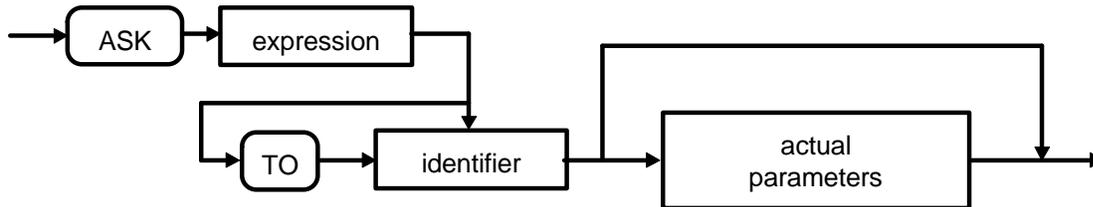
**Figure 10-1. Method Invocation**

**TO** is a “noise word” which can be optionally specified in method calls to make the code more readable, but it has no effect during execution.

An **ASK** statement behaves exactly like a procedure call. When the **ASK** statement is executed, the object is requested to invoke the method. The calling code then *waits* for the invoked method to finish execution before proceeding past the **ASK** statement. **ASK**

methods are not allowed to pass any simulation time, so, in a simulation, the action just described takes place at one instant of simulation time.

Since the **ASK METHOD** is like a procedure, it can either be a proper method or a function method which returns a value.

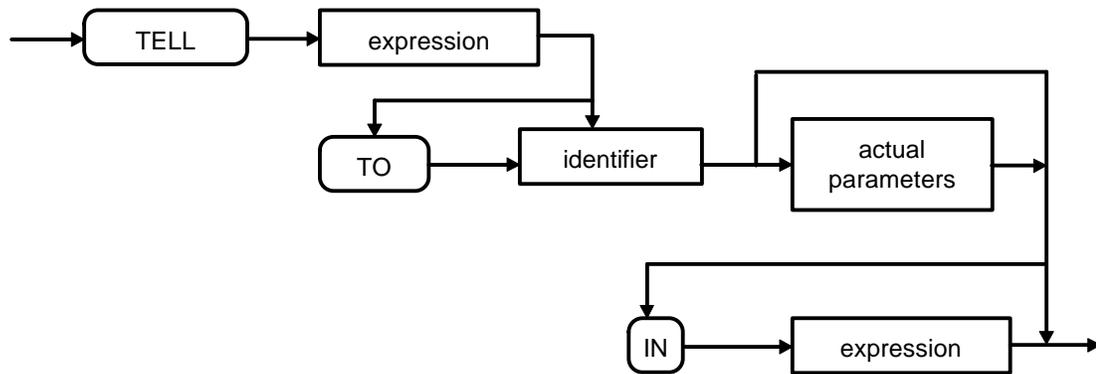


**Figure 10-2. Syntax of the ASK Statement**

The **TELL** method is also known as a **delayed method call**. It is essentially an asynchronous call. The calling code executes the **TELL** statement which requests the object to invoke the method. **The calling code then proceeds past the TELL statement without waiting for the invoked method to complete execution** or, for that matter, even to start. **TELL** methods are allowed to pass simulation time.

The **TELL METHOD** is only available in one form; the proper method. It is not possible to define a **TELL** method which acts like a function method and returns a value. Since the **TELL** method is called asynchronously and there is no further connection between the calling code and the **TELL** method, there would be no place to which a return value could be passed. Actually there is a place in the invoker's code to which a return could be made, but the invoking routine is not waiting there to regain control. It has continued execution past that spot in the code. For the same reason, **INOUT** and **OUT** parameters are not allowed in **TELL** methods. In Section III we will cover the **TELL** method and asynchronous calls in more detail.

The **WAITFOR** method has characteristics of both the **TELL** method and the **ASK** method. Its use is constrained to the **WAIT FOR** statement. As with **TELL** methods, **WAITFOR** methods are allowed to pass simulation time by means of the **WAIT** statement. Unlike **TELL** methods, **WAITFOR** methods may pass values back through **IN** and **INOUT** parameters. In Section III we will revisit the **WAITFOR** method and discuss the **WAIT FOR** conditional control construct.



**Figure 10-3. Syntax of the TELL Statement**

Note that the syntax of the **TELL** statement allows options not covered here. These will be discussed in Section III.

Here are two examples of method invocations; i.e. sending messages to objects requesting that methods be performed:

```

TELL Aircraft TO FlyTo(OHare);
location := ASK Ship position();

```

The first statement is an asynchronous call. We are asking the object instance named **Aircraft** to execute its method called **FlyTo**. We have provided an input parameter, presumably telling it where to go.

The second statement is a synchronous call of a function method. We are asking the object instance named **Ship** to execute its method called **position**. Presumably this will return the ship's current location, which is then assigned to the variable **location**.

**ASK** and **TELL** methods of an object are invoked from outside of the object using the **ASK** or **TELL** keyword, the object's reference value, the method name, and any arguments to the method. Note that the reserved word **TO** is optional. It is one of the few optional elements of MODSIM's syntax:

```

ASK { or TELL } object [ TO ] method();

```

There is also a short-cut notation which can be used to invoke **ASK** and **TELL** methods of an object. It is not necessary to use the **ASK** or **TELL** syntax within any of the object's methods to invoke its own methods. Methods can be invoked as if they were locally defined procedures.

## 10.2 Built-in Reference Constant SELF

If it is necessary to refer to the object itself, within one of its methods, we use the built-in reference type constant **SELF**.

If the three statements above had been placed respectively within methods of an **Aircraft** and **Ship** object, they would look like this:

```

FlyTo(OHare);
      or
TELL SELF TO FlyTo(OHare);

location := position();
      or
location := ASK SELF position();

```

**SELF** may also be used when an object wants to identify itself to another object, as in:

```
TELL Tiger20 TO ReportDistance(SELF);
```

This would request **Tiger20** to report its distance from the object making the request.

**Note:** The usual rules which apply to constants also apply to **SELF**.

## 10.3 Referencing an Object's Fields

The **ASK** method is also used for another purpose. In order to determine the value of an object instance's fields, the **ASK** method is used in a manner similar to a function call. In essence we ask the object for the value of its fields. However, the value of an object instance's fields can only be modified within its own methods.

The form is:

```
ASK object field
```

For example:

```
range := ASK Car fuelLeft * ASK Car MPG;
```

This statement requests the value of two fields of the object instance named **Car**. An expression is formed from the two requests and the car's range is computed based on its remaining fuel and its mileage rate. Note that, in contrast to invoking a function method, no empty parameter list is used since these are fields which are being referenced.

There is also a shortcut notation which can be used to reference fields of an object instance. Within any of the object's methods, it is not necessary to use the **ASK** syntax. Fields can be referenced as if they were local variables.

If the statement above had been placed within a method of a `Car` object, it would look like this:

```
range      := fuelLeft * MPG;
           OR
range      := ASK SELF fuelLeft * ASK SELF MPG;
```

Here is another example which shows how fields of an object are referenced from outside the object:

```
IF ASK Tiger20 position <> HomeBase
  TELL Tiger20 TO ProceedTo(HomeBase);
  OUTPUT("Not at home base, but returning");
ELSE
  OUTPUT("Already at home base");
END IF;
```

If the same piece of code were in one of the object's own methods, it would look like this:

```
IF position <> HomeBase
  ProceedTo(HomeBase);
  OUTPUT("Not at home base, but returning");
ELSE
  OUTPUT("Already at home base");
END IF;
```

In the second line above we could also have used:

```
TELL SELF TO ProceedTo(HomeBase);
```

There is a shorthand version of `ASK`. Occasionally, you may have nested `ASK`s and the notation can get a bit cumbersome. The `'.'` (dot) may be used to access fields and/or `ASK METHODS` of an object instead of or in combination with the `ASK` specifier. This alternative has been provided to simplify nested `ASK` expressions in code.

Example:

```
ASK (ASK object group) numberIn;
ASK (ASK object group) Remove();
```

may be replaced, respectively, with:

```
object.group.numberIn;
object.group.Remove();
```

## 10.4 Monitoring of Fields or Variables

Monitoring is a group of powerful features that allow behavior to be attached to the operations of accessing or setting the value of any variable or field. The monitoring operation is separate and hidden from the processing that uses or sets the value.

You can use monitoring in many ways. For example, you can monitor a variable and update a screen plot whenever the value changes. You could also represent a value in storage in a different form that you use for processing the value, e.g., an encrypted or compressed form. Monitoring can also be used as a debugging tool, allowing you to watch over certain key variables or fields without altering the main flow of program logic. Monitoring can also allow you to separate the operation of a pure simulation model from the statistics gathering that obtains simulation results.

A variable or field (whether of an object or record) may be monitored. Monitoring may be specified as being left, right, or left and right. Left monitoring means that any time a value is given to the variable or field, the monitoring methods that you have specified will be invoked (i.e., that variable or field is on the left hand side of an assignment). Right monitoring invokes the specified methods whenever the variable or field is referenced (i.e., the variable or field is on the right hand side of an assignment).

You perform monitoring by replacing the data type of a variable or field with a monitor type that has been declared for the desired data type. Any valid data type can be monitored, including enumeration type, subrange type, INTEGER, REAL, STRING, BOOLEAN, object type, record type, and array type.

As well as providing all capabilities of the desired data type, monitoring also invokes your **LMONITOR** and/or **RMONITOR** methods for the monitor type.

In order to obtain the benefits of monitoring you must provide three elements:

1. Define a monitor object.
2. Provide an implementation of the monitor methods.
3. Attach the monitor object to a variable or field.

Monitoring can be either static or dynamic. Static monitoring is part of the program's complete run while dynamic monitors can be added or removed during a run.

Monitoring is a rich feature because it can support type structures and inheritance. For instance, you can use dynamic monitoring to attach a monitor to a field of a single instance of some object. This marked spy object participates in operation of a simulation model, but can gather a trace of its own unique history or provide special reports.

### 10.4.1 Example of Static Monitoring

```

MAIN MODULE MonXmp;
  MonitorSample = MONITOR INTEGER OBJECT
    LMONITOR METHOD SetNewValue;
    RMONITOR METHOD GetOldValue;
  END OBJECT;
  OBJECT MonitorSample
    LMONITOR METHOD SetNewValue;
    BEGIN
      OUTPUT("SetNewValue to ", NEWVALUE);
    END METHOD;
    RMONITOR METHOD GetOldValue;
    BEGIN
      OUTPUT("GetOldValue, which is ", VALUE);
    END METHOD;
  END OBJECT;
VAR
  queuelen : LRMONITORED INTEGER BY MonitorSample;
BEGIN
  queuelen := 0;
  queuelen := queuelen + 1;
END MODULE

```

### 10.4.2 Defining Monitor Objects

A monitor object type is declared with the full generality of any other object type declaration, as shown in figure 9-1.

### 10.4.3 Syntax

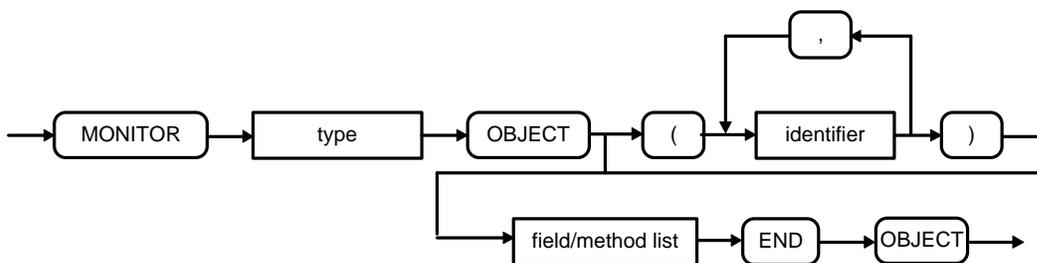


Figure 10-4. Syntax for Monitor Objects

### 10.4.4 Semantics

A monitor type may have as many left- and right-hand monitoring methods as desired. When there is more than one monitoring method for a given direction they will be called in the order in which they are defined.

A monitor type may inherit from other objects. The parents of a monitor type may be either monitor types or non-monitor types. An object that inherits from a monitor object must be a monitor object and must be for the same data type. Only a monitor type may have monitor methods.

A monitor type may be constructed for another monitor type, and a variable or field may be monitored by more than one monitor object.

## 10.5 Implementation Features for Monitor Methods

### 10.5.1 Syntax

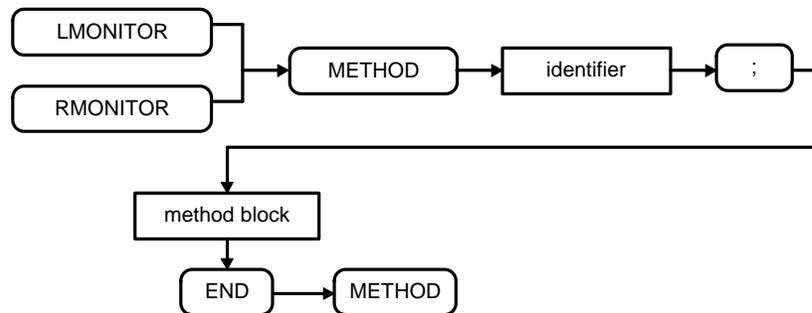


Figure 10-5. Syntax of Monitor Methods

### 10.5.2 Semantics

A monitor method can access three special quantities: **VALUE**, **NEWVALUE**, and **UPDATEVALUE**.

**VALUE** may be accessed from all monitor methods. It provides the last contents of the monitored variable or field. Its type is identical to the declared monitor object type.

**NEWVALUE** is available in **LMONITOR** (left monitor) methods. It gives the value that the variable or field is scheduled to acquire.

**UPDATEVALUE** may be called from **LMONITOR** methods. Such calls allow the method to modify the **NEWVALUE** of the variable or field. Subsequently invoked methods will have the modified **NEWVALUE**. After all left monitor methods have been invoked the variable or field will be assigned the value of **NEWVALUE**.

## 10.6 Attaching a Monitor Object to a Variable or Field

### 10.6.1 Syntax for Simple Fields

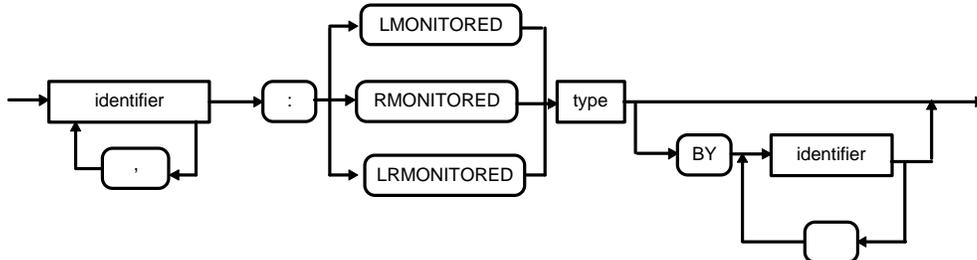


Figure 10-6. Syntax for Simple Fields

### 10.6.2 Syntax for Monitor Types

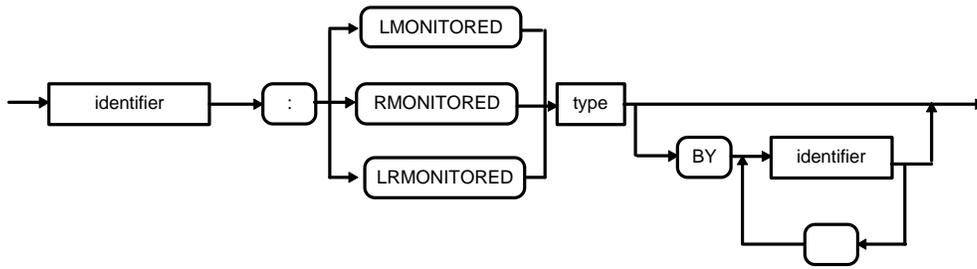


Figure 10-7. Syntax for Monitor Types

### 10.6.3 Semantics

To declare a simple monitored variable or field use the syntax for simple fields. You can also declare an entire type to be monitored using the syntax for monitor types.

The "BY" list is optional. When present it is a list of one or more monitors. All of these monitors must be monitors for the same data type. The listed monitors, also known as the "static monitors" have monitor objects created for them automatically and are automatically attached to the monitored variable. In addition static monitors are automatically disposed when the monitored variable goes out of scope.

Because monitors are full-fledged objects, the `ObjInit` method (if provided) will allow them to initialize, and `ObjTerminate` (if provided) will allow them to clean up gracefully.

When a variable or field is declared as `LMONITORED` or `LRMONITORED`, each `LMONITOR` method of all of its attached monitor objects will be invoked. Similarly, when a variable of field is declared as `RMONITORED` or `LRMONITORED`, all of the attached `RMONITOR` methods will be invoked.

## 10.6.4 Dynamic Monitors

During a run, additional monitors may be added to a variable or field that has been declared to be monitored. This is done by creating (with **NEW**) a monitor object of the correct type and using the **ADDMONITOR** procedure to add this "dynamic monitor" to the variable's monitor objects. By default, the monitor is enabled, that is, **ADDMONITOR** performs **ACTIVATE**.

During a run, you can use the **ACTIVATE** and **DEACTIVATE** procedures, as appropriate, to turn the operation of a monitor object on or off.

You are responsible for deallocating the dynamic monitor object, so before you **DISPOSE** of a **RECORD** or **OBJECT** containing a monitored field, you should see that **DEACTIVATE** and **REMOVEMONITOR** are performed, as appropriate.

Since each monitor object has its own fields and methods, it may behave just as any object.

**GETMONITOR** is provided to obtain the reference to one of the monitor objects of a monitored variable or field. The built-in procedure **GETMONITOR** takes two arguments: a monitored variable or field and a monitor object type name. It returns the object of the specified type. The return value can be used to access fields and methods of the monitor object.

Because of inheritance, it is possible that a given field will have monitors for its type and its base types. It is also possible to use **ADDMONITOR** to attach several monitor objects for the exact same type. When this is done all of the monitors will operate. The **GETMONITOR** function will only return the first monitor for any specific type. If there are more, or if you add a dynamic monitor for a type that has a static monitor, you may need to do additional bookkeeping.

# 11. Inheritance

---

A new object type can be defined in terms of an existing object type. This is called **inheritance**. The newly derived object type is then termed a **derived type** of that **base type**. The derived type will have all the fields and methods of the base type.

The derived type will typically define additional fields and/or methods not present in the base type. It may also **override** the implementation of a method defined in an underlying object type and replace it with its own.

MODSIM III also supports a form of inheritance known as multiple inheritance in which a new object type is defined in terms of two or more existing object types.

A method in a derived type can invoke a method of the same name in an underlying or base type, by use of the **INHERITED** keyword. This is useful when the programmer wishes to append new code to an old method of the same name. The **INHERITED** keyword effectively inserts the entire old method in the new method with one statement.

Any method not explicitly overridden by the derived type is automatically inherited from the base type. Similarly, the derived type also inherits all fields of its base type.

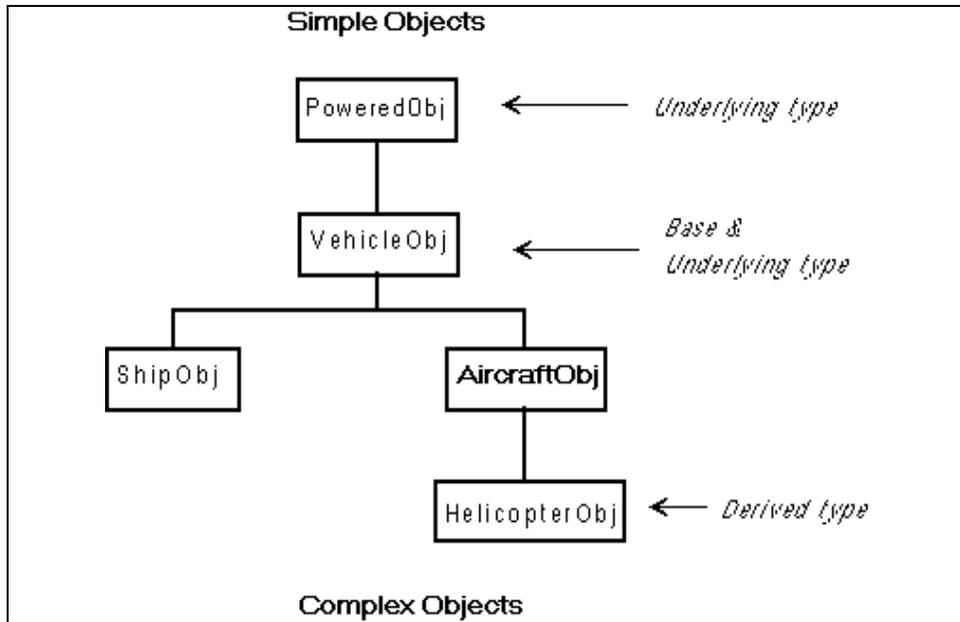
While a derived type can override and replace inherited methods, it cannot redefine inherited fields. It can, however, add new fields and new methods of its own.

Attempts to cast down object references to an object type which is not an ancestor will always cause a runtime error.

## 11.1 Hierarchical Object Types

The object type is special in a number of ways. The capability to inherit the fields and methods of an object and elaborate on them is a powerful feature. If the language were to impose the traditional type rules on the objects involved in an inheritance hierarchy, this would limit the usefulness of objects.

Because of this feature, an object in a hierarchy is considered to be compatible with the type of any of its ancestors. The following figure shows a hierarchy of objects all descended from one common object, the **PoweredObj**.



**Figure 11-1. Object Type Hierarchy**

If we look at the object hierarchy tree from the perspective of the `AircraftObj`, we can describe the relationship of objects in the tree in the following way.

All of the types above `AircraftObj` in the tree are known as **underlying types**. They are the ancestors of `AircraftObj`.

`VehicleObj`, the object immediately above `AircraftObj`, from which it was descended, is the **base type** of `AircraftObj`. `AircraftObj` inherited all of its fields and methods from `VehicleObj`.

Finally, any connected object below `AircraftObj` in the tree is a **derived type** of `AircraftObj`. They are descendants of `AircraftObj`.

The hierarchical type rules for objects state that a reference value for an object can safely be assigned to a reference variable of one of its underlying object types. The converse is not true.

If we had an object instance of type `HelicopterObj`, we could safely assign it to a variable of type `VehicleObj` and then invoke its methods or check the values of its fields. This is because the `HelicopterObj` has inherited all of the fields and methods of `VehicleObj`.

MODSIM III will not allow the programmer to assign an object instance of type `VehicleObj` to a variable of type `HelicopterObj`. Consider that the `HelicopterObj`

had defined a method call `Hover`. The `VehicleObj` would not know what to do if asked to execute the `Hover` method. Serious trouble would ensue.

The hierarchical type rules we just described apply in all situations. Thus, if a procedure or method was expecting a parameter of type `VehicleObj`, we could safely and legally invoke the routine with a variable of type `HelicopterObj`.

## 11.2 Coercion of Objects

MODSIM III allows explicit type coercion of objects, so that carefully crafted code may employ a direct and safer mechanism to accomplish assignment of parent (base) class object instances to child (derived) class variables and vice versa. Type checking will be performed to ensure that the coercion type is either an ancestor or child of the instance type. Coercion is specified by preceding the instance variable to be coerced by the desired type name and surrounding the instance variable with parentheses.

**Example:**

```

parent = OBJECT
    foo : INTEGER;
    ...
END OBJECT;

child = OBJECT(parent)
    bar : REAL;
    ...
END OBJECT;

VAR
    p : parent;
    c : child;
    ...

```

```

(1)    c := child(p); (* force p to be accepted as a
                    child type object *)
        p := parent(c);(* force c to be accepted as a
                    parent type object *)

```

Coercion should be used sparingly and very carefully. Remember that regardless of the type of the left-hand side of an assignment, the right-hand side remains an instance of the object class from which it was **NEW**'ed. It is not necessary to expressly cast child class objects when assigning them to parent class variables, but for clarity of code it is useful to do so. When assigning parent class object instances to child class variables explicit casting is required. This is a potentially dangerous, although occasionally necessary, operation since you are “fooling” the compiler into allowing reference to fields and/or

methods that the parent class instance may not actually have. In the above example, after the first assignment (1) the compiler would allow:

```
ASK c bar;
```

However, `c` might have been NEW'ed as a parent class object which does not have a `bar` field. This type of problem would not express itself until runtime and would cause a runtime error to be generated.

### 11.3 Object Inheritance

As we mentioned earlier, most of the object types a programmer uses are built upon the definitions of other object types, either those from the standard MODSIM library, or user-defined object types. This is a common way to define an object type.

To show how a new object type inherits the attributes of its ancestor and builds upon it, we can continue with the example shown in the type hierarchy tree above. Consider how the `HelicopterObj` was defined based on the existing `AircraftObj`. For that matter it is worth examining the entire hierarchy:

```
DEFINITION MODULE MovingMod;

TYPE
  fuelType = (Unleaded, Diesel, AvGas, JetFuel);

  locationType = RECORD
    latitude,
    longitude : REAL
  END RECORD;

  EngineObj = OBJECT
    power      : INTEGER;
    fuel       : fuelType;
    kmPerLtr  : REAL;
    TELL METHOD ConsumeFuel;
  END OBJECT;

  VehicleObj = OBJECT( EngineObj )
    position   : locationType;
    course    : [ 0..359 ];
    speedKm   : INTEGER;
    fuelLevel : REAL;
    TELL METHOD GoTo(IN destination: locationType);
    TELL METHOD Stop;
  END OBJECT;

  AircraftObj = OBJECT( VehicleObj )
    altitude  : INTEGER;
  END OBJECT;
```

```

HelicopterObj = OBJECT( AircraftObj )
  inHover : BOOLEAN;
  TELL METHOD Hover(IN posit : locationType;
                  IN alt    : INTEGER);
END OBJECT;
END MODULE.

```

The derived object has access to all of the properties of its base type, in addition to its own unique properties, so we could implement the `Hover` method using properties of both `AircraftObject` and `HelicopterObject`. `HelicopterObject` is a composite of all of its ancestors plus the fields and methods it has declared on its own. This means that it has the following fields and methods:

```

power      : INTEGER;
fuel       : fuelType;
kmPerLtr   : REAL;
position   : locationType;
course     : [ 0..359 ];
speedKm    : INTEGER;
fuelLevel  : REAL;
altitude   : INTEGER;
inHover    : BOOLEAN;

TELL METHOD ConsumeFuel;
TELL METHOD GoTo(IN destination: locationType);
TELL METHOD Stop;
TELL METHOD Hover(IN posit : locationType;
                 IN alt    : INTEGER);

```

## 11.4 Overriding Methods

There will be occasions when one of the methods a derived object type has inherited is not appropriate or needs to be changed in some way. In this case, the new object type declaration can explicitly state in the object type declaration that it is overriding the inherited method. It must then provide a replacement for the overridden method in its object declaration block.

Assume that the `VehicleObj` type defines a `Stop` method which looks like this:

```

TELL METHOD Stop;
BEGIN
  speedKm := 0;
END METHOD;

```

If it were necessary to provide a more elaborate `Stop` method for the `AircraftObject`, this is how the object type declaration would look:

```

...
AircraftObj = OBJECT( VehicleObj )
  altitude : INTEGER;
  OVERRIDE

```

```

    TELL METHOD Stop;
END OBJECT;

```

Then we would provide a replacement method declaration with new code in the object declaration block:

```

OBJECT AircraftObj
  TELL METHOD Stop;
  BEGIN
    ... make sure we're on the ground first!
    altitude := 0;
    speedKm := 0;
  END METHOD;
END OBJECT;

```

This is how polymorphism is achieved. The `VehicleObj` has a method called `stop` which simply sets its speed to zero. The `AircraftObj` also has a method called `stop`, but this method executes some code to ensure that the aircraft is back on the ground before it sets the speed to zero. Each object executes a different behavior when sent the message to stop. It will always be appropriate to do the following:

```

  TELL SomeObj TO Stop;

```

as long as `SomeObj` is either a `VehicleObj` or is derived from `VehicleObj`.

## 11.5 Extending Inherited Behaviors

In some cases, the overriding method completely replaces the method from the underlying type. This is what we just did with the `stop` method for `AircraftObj`. However, in other cases, it may be desirable to merely extend the underlying method. In these cases the new method can invoke the overridden method as part of its behavior and then provide additional code which further describes its behavior.

To invoke the overridden method from the base type we precede a standard method invocation with the reserved word `INHERITED`.

For example, to implement the proper method `GoTo` for an `AircraftObj`, it may be easier to build upon the existing `GoTo` code defined for its base type, `VehicleObj`. Once the inherited `GoTo` method has been overridden in the object type declaration, a replacement is provided which calls the original method.

```

OBJECT AircraftObject;

  TELL METHOD GoTo(IN destination: locationType);
  BEGIN
    { some flying-specific code }
    INHERITED GoTo(destination);
    { more flying-specific code }
  END METHOD;END OBJECT;

```

Thus, the `GoTo` method for an `AircraftObj` would perform some unique calculations, and then invoke the `GoTo` method from the underlying object type; in this case, `VehicleObj`.

An inherited call can be performed for a function method as well. Such a method might contain a statement such as:

```

ASK METHOD FuelAmount(IN TankNum: INTEGER) : REAL;
...
SomeVar:= R1Num * (INHERITED FuelAmount(TankNum)) - 4.0;
...

```

Operations to be performed “before” and “after” a particular method can be handled in MODSIM by the ordering of code before and after the inherited call. In general, each method that uses inherited code will take the form:

```

BEGIN
  { code preparing for the inheritance }
  INHERITED thisMethod(args);
  { code using the inheritance }
END METHOD;

```

This mechanism is both simple and versatile, and is appropriate for all single-path inheritance combinations of methods. When an object inherits methods from more than one object type, a somewhat more complex approach is necessary. This will be described in the following sections on multiple inheritance.

### 11.5.1 Overriding the `ObjInit` Method

There will be occasions when the new object defined through inheritance will want to elaborate the `ObjInit` method of its ancestor. The `ObjInit` method can be overridden just like any other method. However, if this is done, the original `ObjInit` method must be invoked with an inherited call. This is also true for `ObjClone` and `ObjTerminate`. It is very important to ensure that any previously defined `ObjInit` methods are invoked. If this rule were not observed, then some crucial part of an object's initialization could be inadvertently omitted. This is particularly important with objects imported from libraries. The user may not be aware of the initialization requirements for these objects.

## 11.6 Multiple Inheritance

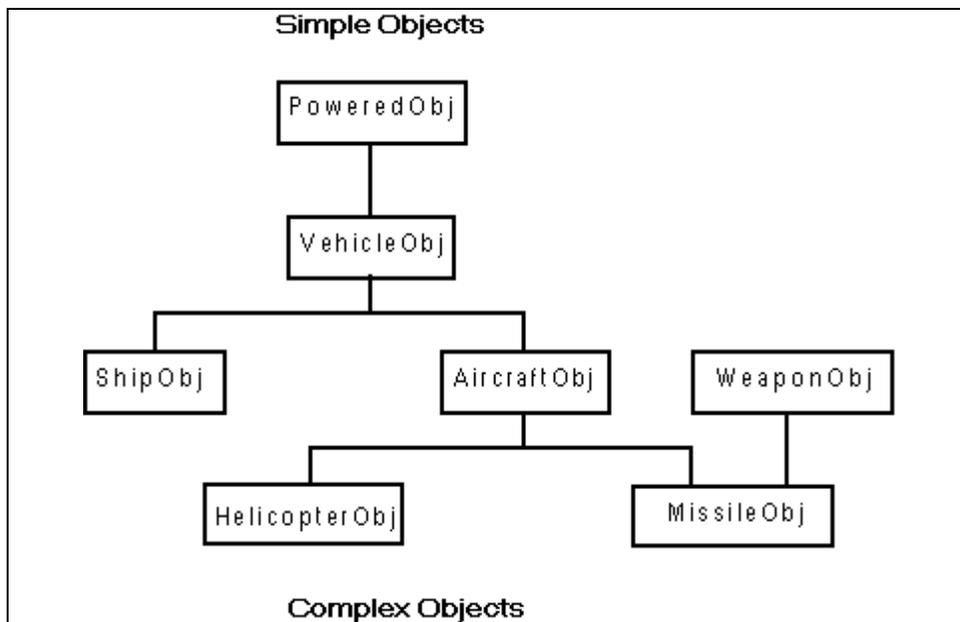
MODSIM III allows an object type to be defined in terms of more than one base object type. This is called **multiple inheritance**.

When a new object type is defined in this way, it has a copy of each field and each method of each of its base types. Like powerful features in any system, this can be a two-edged sword. If the base types from which the new object type have been derived have used the same identifiers to name any of their fields or methods, we are left with an ambiguous situation. MODSIM provides facilities to resolve these conflicts.

### 11.6.1 Declaring Multiple Base Types

To define an object type in terms of multiple base types, each base type is listed in the object type declaration:

```
MissileObj = OBJECT(AircraftObj, WeaponObj)
    define more fields and methods here...
END OBJECT;
```



**Figure 11-2. Multiple-Path Inheritance**

In this case we have used the existing **AircraftObj**, added all the fields and methods of **WeaponObj** and then added more fields and methods unique to the new **MissileObj**.

## 11.7 Resolving Conflicting Field Names

If field identifiers of the same name exist in two or more of the base types, the derived object type will contain a field for each one. Obviously, any attempt to reference those fields in the derived object type would be ambiguous, particularly if some of the fields with matching names were of differing types. Because of this, MODSIM does not allow references of this sort and will flag them as a compile-time error.

If a field from a base type must be accessed and some other base type has a field of the same name, extra code must be provided to disambiguate the field. This code can assign the reference value of the object to an object of the desired base type, and then unambiguously access the desired field.

Consider the situation which would occur if the **AircraftObj** and the **WeaponObj** from which **MissileObj** was derived each had a weight field. And just to make things more difficult, the **WeaponObj**'s weight field is of type **REAL** and expressed in kilograms. The **AircraftObj**'s weight field is of type **INTEGER** and is expressed in pounds.

Assume we have three reference variables called **Aircraft**, **Weapon** and **Missile** to match their respective types. If we assign an instance of **MissileObj** to all three reference variables, we have the following situations:

```

Missile := ASK armory TO Issue(CruiseMissile);
Aircraft := Missile;
Weapon := Missile;

n := ASK Missile weight    ← illegal reference
n := ASK Aircraft weight  ← n gets Aircraft's weight (an INTEGER)
x := ASK Weapon weight    ← x gets Weapon's weight (a REAL)

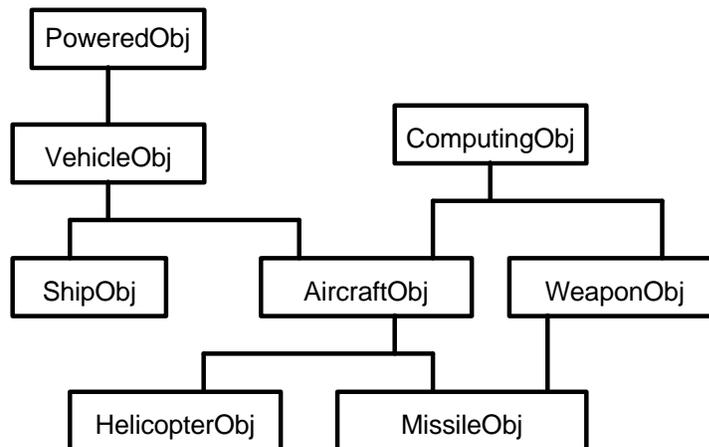
```

## 11.8 Resolving Common Method Names

Cases where two or more of the base types have methods of the same name are permitted only when the method is derived from a common ancestor. If there is not a common ancestor, the MODSIM compiler produces an error message.

The definition of the object that joins the ancestors must override the common method if any of the intervening ancestors overrides it. Otherwise, polymorphism will not be able to work for this method and the MODSIM compiler will produce an error message.

You can supply a completely new method implementation or, as with normal inheritance, the inherited method can be invoked as part of the implementation. When the method is inherited from multiple ancestors, a qualified form of the inherited method invocation can be used to specify the desired version of the method.



**Figure 11-3. Common Ancestor**

As an example, we can consider the `MissileObj` which was derived from an `AircraftObj` and a `WeaponObj`. Assume that each of the base types has a method called `FindTarget`. Observe that the `FindTarget` method is itself a method of some `ComputingObj` from which both `AircraftObj` and `WeaponObj` inherit:

```

DEFINITION MODULE ...
...
  ComputingObj = OBJECT
    ASK METHOD FindTarget(IN enemy: VehicleObj);
  END OBJECT;
...
  AircraftObj = OBJECT(VehicleObj, ComputingObj)
...
    OVERRIDE
      ASK METHOD FindTarget (IN enemy: VehicleObj);
    END OBJECT;
  WeaponObj = OBJECT(ComputingObj)
...
    OVERRIDE
      ASK METHOD FindTarget(IN enemy: VehicleObj);
    END OBJECT;

```

The inheriting object must override the common method and provide its own:

```

MissileObj = OBJECT(AircraftObj, WeaponObj)
...
  OVERRIDE
    ASK METHOD FindTarget(IN enemy: VehicleObj);
  END OBJECT;

```

If the common method is to be invoked in the implementation of the inheriting object, a qualified inherited call must be used. The qualified inherited call explicitly specifies the desired version of the method.

Continuing with the previous `MissileObj` as an example, the implementation might provide the following method:

```
OBJECT MissileObj;
  ASK METHOD FindTarget(IN enemy: VehicleObj);
  BEGIN
    ...
    INHERITED FROM AircraftObj FindTarget(enemy);
    ...
  END METHOD;
END OBJECT;
```

A qualified inherited call requires the qualifier to be a base type of the object that is being defined. The `INHERITED FROM` syntax cannot be used to access methods of unrelated objects. In the example above this means that the inherited call for the `FindTarget` method can only be qualified by one of the two base types of `MissileObj`; either `WeaponObj` or `AircraftObj`. We could not inherit the `Dive` method from `SubmarineObj`, since we are not descended from it.

### 11.8.1 Combining Multiple Inherited Methods

In many cases, it may be useful to invoke the inherited methods from multiple ancestors in the derived type's method. This can be done since the `INHERITED` statements are qualified to avoid ambiguity.

Elaborating on the previous example, we could do the following:

```
OBJECT MissileObj;

  ASK METHOD FindTarget(IN enemy: VehicleObj);
  BEGIN
    ...
    INHERITED FROM AircraftObj FindTarget(enemy);
    ...
    INHERITED FROM WeaponObj FindTarget(enemy);
    ...
  END METHOD;
```

### 11.8.2 Overriding the `ObjInit` Method in Multiple Inheritance

It is always necessary to ensure that the `ObjInit` method for an object is invoked if it exists. This ensures that all initialization code for an object is accomplished. In multiply inherited objects it is necessary to override any existing `ObjInit` methods, and to then invoke each of the inherited methods. Thus, if we defined a new object type in the following way:

```

c = OBJECT(a, b);
...
OVERRIDE
    ASK METHOD ObjInit;
END OBJECT;

```

In the object declaration block where method declarations are placed we would do the following:

```

OBJECT c;
    ASK METHOD ObjInit;
    BEGIN
        INHERITED FROM a ObjInit;
        INHERITED FROM b ObjInit;
    END METHOD;
END OBJECT;

```

Of course, we might want to add additional initialization code appropriate to the new object type, but, at a minimum, we would have to do this much in the replacement `ObjInit` method.

## 11.9 Conflicting Field and Method Names

If a method name from one base type is the same as a field name from another base type, MODSIM flags this as a compile-time error. There is no way to resolve this conflict except by renaming one of the fields. This is intentional.

No conflict resolution mechanism has been provided in this case since it would lead to code which, although it could be understood by the compiler, would be confusing or misleading to those responsible for code maintenance.

## 12. Data Hiding and Data Sharing

---

The modular organization of MODSIM programs encourages the separation of the definition of procedures and methods from the details of their implementation. Since other modules can import from the definition module but cannot see the implementation module, this provides both a **data hiding** and **data sharing** capability.

Data sharing is also available from the perspective of a single module. Since any constants, types, procedures or variables defined in a module are visible throughout that module, this allows data sharing between any object types defined in that module.

Data hiding is also available from the perspective of a single module. There are occasions when some fields or methods of an object type should not be used except by the object itself. The data hiding requirement for an object-oriented language parallels that for other software engineering efforts, and thus MODSIM supports encapsulation both indirectly through modules and directly through **PRIVATE** fields and methods.

Fields or methods declared as **PRIVATE** can be referenced only within methods of the object itself, or within methods of derived object types.

### 12.1 PRIVATE Fields and Methods

Here is an expansion of the previous object type declarations. We have added a few more methods and one field. One method and one field have been declared to be private to this object:

```
TYPE
  AircraftObj =
  OBJECT( VehicleObj )
    altitude : INTEGER;
    TELL METHOD ClimbTo(IN height: REAL);
    TELL METHOD Circle;
    ASK METHOD FindTarget(IN enemy: VehicleObj);
  PRIVATE
    liftCoefficient : REAL;
    ASK METHOD CalcLiftCoeff;
    WAITFOR METHOD DeployLandingGear;
  OVERRIDE
    TELL METHOD Stop;
  END OBJECT;
```

The **PRIVATE** section lists all of the fields and methods which are part of the object type declaration, but which cannot be accessed from outside of the object.

When an object type is imported from a definition module, all of its field and method identifiers are also imported, except those which have been declared to be **PRIVATE**.

Thus, if we had imported `AircraftObj` into a module and declared a reference variable of that type called `plane`, we could do the following:

```
fuelLeft := ASK plane fuelLevel;  
TELL plane TO ClimbTo(2500.0);  
ASK plane TO FindTarget(tank);
```

Because `liftCoefficient` is a private field and `CalcLiftCoeff` is a private method of type `AircraftObj` we could not do the following:

```
ASK plane TO CalcLiftCoeff;  
OR  
coeffOfLift := ASK plane liftCoefficient;
```

## **Section III. Simulation**



# 13. Process-based Discrete-Event Simulation

---

MODSIM has powerful and flexible capabilities for dealing with discrete-event simulation. Each object is capable of carrying on multiple, concurrent activities, each of which can elapse simulation time. An activity is an event scheduled by an object instance using a **TELL** or **WAITFOR** method which is capable of elapsing simulation time. The activities can operate autonomously or they can synchronize their operation. Any or all activities of an object can be interrupted, if necessary.

Not only can one object instance have multiple **TELL** and/or **WAITFOR** methods carrying on activities simultaneously with respect to simulation time, but any one method of the object instance can be invoked multiple times. Each of these method invocations can be carrying on an activity at the same time.

In MODSIM all of the bookkeeping to schedule activities and later execute them is taken care of by the system.

## 13.1 Simulation Time

Before going any further we need to discuss the concept of **simulation time**. A discrete-event simulation program models a sequence of events. Typically, we are concerned with the model only at certain points in simulation time when an event occurs which may change the state of objects in the model.

The units of time used by the simulation are dimensionless. They can represent whatever granularity of time is appropriate for the simulation - years, months, days, hours, minutes, seconds, milliseconds, or nanoseconds. It is up to the user to explicitly perform any unit conversions.

Simulation time is automatically maintained by MODSIM. The current simulation time can be determined by invoking the real-valued function **SimTime()**, which may be imported from **SimMod**.

At any instant of simulation time there can be multiple, concurrent activities. In reality, on traditional sequential computer architectures, the activities which appear to be happening at the same point of simulation time are carried out sequentially by the computer in actual “wall clock” time. Once all of the activities scheduled for a particular instant of simulation time have been carried out, the simulation clock is advanced to the next point in simulation time when an activity has been scheduled.

### 13.2 The System's Pending List - Objects' Activity Lists

To keep track of all activities which have been scheduled, MODSIM keeps a **pending list**. The pending list is an ordered list of all objects which have scheduled activities. The object with the most imminent activity is ordered first in the list.

Each object instance, in turn, keeps its own list of activities which it has scheduled. The object instance's activity list is ordered by the most imminent activity. Thus, we have a two-dimensional structure.

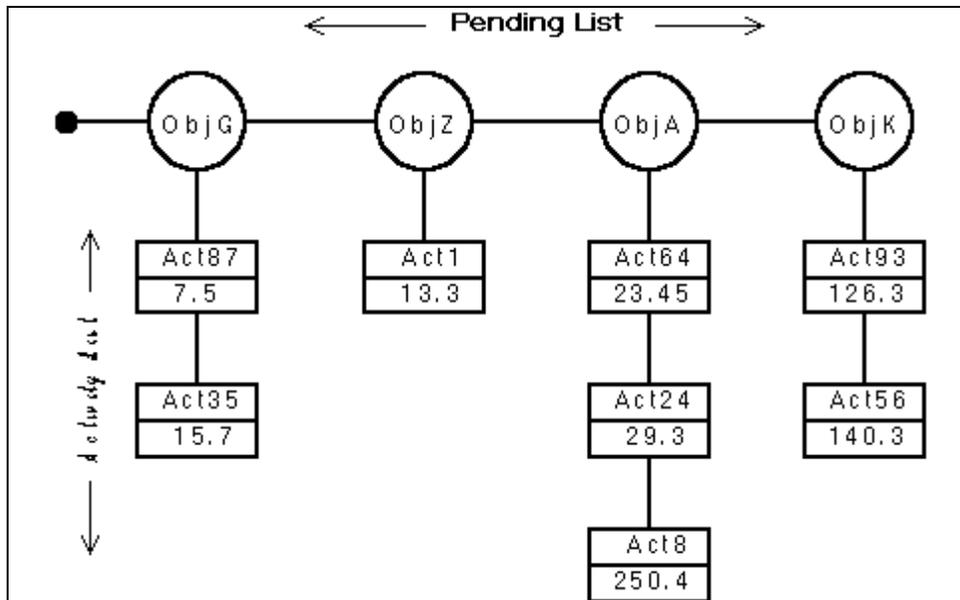


Figure 13-1. The Pending List

### 13.3 Process-oriented vs Event-oriented Simulation

The classical approach to discrete-event simulation is **event-oriented**. In this approach, individual routines are written to describe each discrete event 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 routine for the object currently being manipulated. In the simple

bank model, the event routine “Customer engages services of teller” would schedule the next event routine, “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. Following the logic flow which describes the behavior of a customer would be like tracing through a sequence of GOTO statements in a large BASIC program.

The **process** approach simplifies larger models by allowing all of the behavior of an object in a model (e.g. bank customers) to be described in one or more **TELL** and/or **WAITFOR** methods which allow for the passage of time at one or more points in the method.

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 instance. 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, process 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 the one supported in MODSIM. It exploits object-oriented programming features to simplify both the original development and the subsequent maintenance of large models.

A simulation model written in MODSIM defines a system in terms of processes because 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. The process statements are expressed sequentially, in a manner analogous to the system being described. This practice is recommended by standard design methodologies.

## 13.4 Time Elapsing Methods - the WAIT Statement

The time elapsing **TELL** or **WAITFOR** method is the construct which supports this process oriented approach to simulation. In a **TELL** or **WAITFOR** method it is possible to use a **WAIT** statement to indicate that simulation time should elapse at some point or points in the method.

Each **WAIT** statement in a method is considered to be an activity of the parent object. When a **WAIT** statement is encountered, the **TELL** or **WAITFOR** method suspends execution. When the specified amount of simulation time has elapsed, the **TELL** or **WAITFOR** method resumes execution. We say that the object has carried out an activity.

We will examine the syntax of the **WAIT** statement before examining this capability further.

### 13.4.1 The WAIT Statement

A **WAIT** statement specifies the reason for the wait, a sequence of statements to be executed after the **WAIT** is successfully completed, and an optional sequence of statements to be executed if the **WAIT** is interrupted.

The structure of a **WAIT** statement is similar to that of an **IF** statement. The syntax is:

```

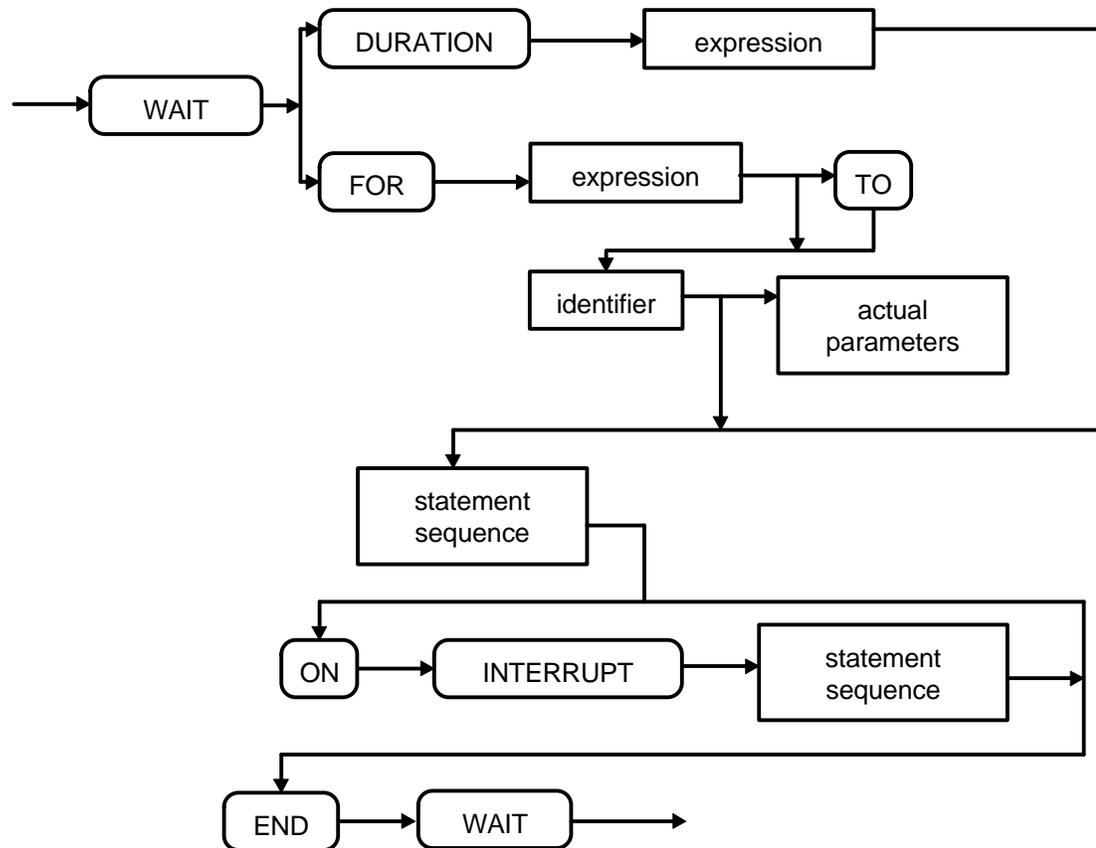
WAIT reason
      StatementSequence
      [ ON INTERRUPT StatementSequence ]
END WAIT;
```

where **reason** is a keyword, **DURATION** or **FOR**, followed by any required identifiers. The **ON INTERRUPT** clause is optional. If the **WAIT** is “successful”, the first statement sequence is executed. If the **WAIT** is “not successful”, the statement sequence after the **ON INTERRUPT** is executed, instead. In either case, execution continues after the **END WAIT** unless one of the statement sequences alters the flow of control.

With all forms of the **WAIT** statement, the **ON INTERRUPT** clause specifies exception code to be executed if the **WAIT** statement is interrupted. The techniques and tools used to interrupt activities of a process will be covered in the next chapter.

If the optional **ON INTERRUPT** clause is omitted and a **WAIT** is interrupted, a run-time error will occur.

A **WAIT** statement can only appear in a **TELL** or **WAITFOR** method. A violation of these rules will be flagged at compile-time.



**Figure 13-2. Syntax of the WAIT Statement**

The most basic **WAIT** is one for a specific period of time. A wait for a specified period of simulation time is achieved by the **WAIT DURATION** statement. The syntax of the statement is:

```

WAIT DURATION timevalue
  Statement Sequence
  [ ON INTERRUPT Statement Sequence ]
END WAIT;

```

where **timevalue** is an expression of type **REAL**.

There are two other variations of the **WAIT** statement which will be covered in more detail shortly. One allows the **TELL** or **WAITFOR** method to wait until another method which is invoked completes execution. Another variation allows the **TELL** or **WAITFOR** method to wait until some triggering event occurs.

### 13.5 The Asynchronous TELL and WAITFOR Calls

Earlier chapters introduced the **ASK**, **TELL**, and **WAITFOR** method calls for proper methods. Although all three “send a message” to the receiving object, the three state-

ments differ in how they interact with simulation time and in the case of the **WAITFOR** method, where it may be called.

In many cases, when an object is sent a message to invoke one of its methods, we want to know that the invoked method has completed before we perform the next step. For example, for an **AircraftObj** to land on a runway, it first must have one properly assigned to it, as in:

```

ASK controller TO AssignRunway(myrunway, assignOK);
IF assignOK
    destination := myrunway;
ELSE
    destination := alternateAirport;
END IF;
...

```

In this case, the simulation logic requires that the **AssignRunway** method for object **controller** be complete before the following **IF** statement is executed. The **ASK** statement is comparable to an ordinary procedure call, i.e., the **AssignRunway** method is required to complete before the next statement is executed.

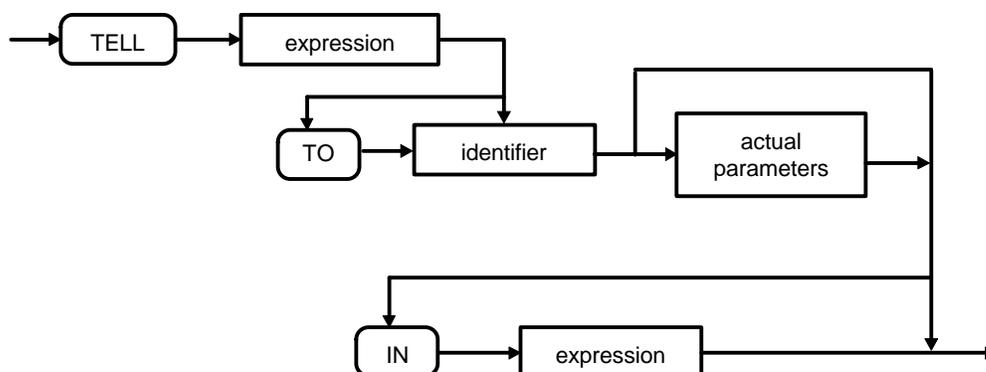
If the activity simulated by a method will elapse an interval of simulation time, it may not be necessary or appropriate for the invoker to pause while that method completes. The invoking code may wish to send a message to another object, invoking one of its time-elapsing methods, and then continue, without waiting for the activity to complete.

This capability is provided by the **TELL** statement. The invoking process executes the **TELL** statement and then continues on without waiting for the invoked time-elapsing method to complete (or even to start) execution. The complete syntax of the **TELL** statement is:

```

TELL object [TO] method[(arguments)] [ IN delay ]

```



**Figure 13-3. Syntax of the TELL Call**

lation

The **TELL** statement can appear in any method or procedure. It is used to invoke **TELL** methods, and may not be used to invoke **ASK** methods. **TELL** methods are proper methods with **IN** parameters only.

A **TELL** method cannot be a function method and cannot have **OUT** or **INOUT** parameters, since there is no place to which this returned information can be passed. The invoking code has proceeded past the **TELL** statement without waiting for any return information.

To take an example, the commander of a unit might want to start a unit enroute to a particular location, using code such as:

```

TELL METHOD DeployTo(IN dest : locationType);
VAR
    unit: UnitObj;
BEGIN
    ...
    TELL unit TO flyTo(dest);
    ...
END METHOD;

```

In this case, the **DeployTo** method would complete execution at the same simulation time it began, no matter how long it eventually took the unit to **flyTo** the destination. Also note that, though this is a **TELL** method, it performs no **WAITS**. Since it is a **TELL** method, however, this means that it can be invoked at some time in the future as in:

```

TELL unit TO DeployTo(PointAlpha) IN 20.0;

```

This is how methods can be scheduled to start execution at some future point in simulation time.

The remaining type of method, the **WAITFOR** method, is somewhat of a hybrid between the **TELL** and **ASK** methods. Like the **TELL** method, the **WAITFOR** method may elapse simulation time. Unlike the **TELL** method, it may only be invoked by a **WAIT FOR** statement and may modify its **OUT** and/or **INOUT** parameters. This is possible because the invoking method will not continue until the **WAITFOR** method finishes. Consequently, there is a place to return to. In these later respects the **WAITFOR** method bears some similarity to the **ASK** method.

For example, a cashier might need to wait for a customer to pay before turning merchandise over to her/him. Paying could elapse simulation time. However, the cashier's activity will not resume until the customer has paid. This is an ideal case for a **WAITFOR** method. Simulation time must pass and something is to be passed back:

```

OBJECT Cashier;
    ...
    WAITFOR METHOD GetPayment(OUT AmtTendered:REAL);
    BEGIN
        ...

```

```

        { code implementing method }
        ...
    END METHOD;
    ...
END OBJECT;

```

### 13.6 Synchronizing Activities

In some simulation scenarios, two activities must operate synchronously. One activity starts a second activity and then suspends its execution and waits over a period of simulation time for the second activity to complete before it resumes execution.

To accomplish this, MODSIM provides the **WAIT FOR** statement:

```

WAIT FOR object [ TO ] method[ (arg) ]
    Statement Sequence
[ ON INTERRUPT Statement Sequence ]
END WAIT;

```

The effect of this statement is to:

```

TELL object [TO] method [(args)];

```

and then wait for the method to complete. Once the invoked method completes execution, the statement sequence after the **WAIT FOR** is executed. If the invoking method is interrupted while still waiting for the invoked method to complete, the statement sequence after the **ON INTERRUPT** is executed.

The obvious question to ask is “Why not use an **ASK** method since it is synchronous?” The answer is simple... an **ASK** method cannot elapse simulation time. So we need a technique like this which overcomes the inherently asynchronous nature of **TELL** methods.

**WAITFOR** methods are a special case of the **TELL** method developed specifically for this case. **WAITFOR** methods allow the user to exploit the fact that the invoking method will not proceed until the method being waited for returns. This guarantee enables **WAITFOR** methods to modify their **OUT** and/or **INOUT** parameters while still allowing simulation time to elapse.

An **INHERITED** method can be invoked using the **WAITFOR** construct:

```

WAIT FOR INHERITED tellmethod()

OBJECT TowPlaneObj;
    WAITFOR METHOD TakeOffClearance(OUT stat:SType);
    BEGIN
        WAIT FOR INHERITED TakeOffClearance
            WAIT FOR glider TO SignalReady
                stat := cleared;
    END

```

lation

```

        ON INTERRUPT
            stat := aborted;
        END WAIT;
    END METHOD;
END OBJECT;

```

Also, an object may schedule one of its own **TELL METHODS** for future execution directly (without the **TELL SELF**):

```
tellmethod() IN 5.0
```

### 13.6.1 The Terminate Statement

Shortly, we will discuss the way in which any of the forms of the **WAIT** statement can be interrupted. However, there is a control statement which is unique to the **WAIT FOR** statement. It is the **TERMINATE** control statement.

The **TERMINATE** statement is executed from within a **TELL** or **WAITFOR** method, as it implies, to terminate execution of that **TELL** or **WAITFOR** method. It has an important further effect. If the **TELL** or **WAITFOR** method which is being terminated was invoked with the **WAIT FOR** method, the invoking method is terminated as well. The effect is recursive. It will continue up a chain of **WAIT FOR** calls.

In the other forms of the **WAIT** statement two conditions can occur:

- The **WAIT** completes normally
- The **WAIT** is interrupted before it is finished.

In the **WAIT FOR** statement a third condition is possible:

- The routine invoked by the **WAIT FOR** terminates, so the method which contains the **WAIT FOR** also terminates.

For example:

```

    IF SimTime() >= StopTime
        TERMINATE;
    END IF;

```

The **TELL** or **WAITFOR** method being waited for can belong to any object.

To illustrate use of the **WAIT FOR**, suppose a combat simulation includes a logistics capability. The deployment process for a combat unit might include a method which waits while an **AircraftObj** flies the unit to its desired destination:

```

TELL METHOD Deploy(IN dest : locationType);
VAR
  ourtransport: TransportObj;
BEGIN
  ourtransport := TransportManager.nextTransport;
  WAIT FOR ourtransport TO FlyTo(dest)
    TELL Hq MyStatusIs(Arrived);
  ON INTERRUPT
    TELL Hq MyStatusIs(Delayed);
  END WAIT;
END METHOD;

```

When the **WAIT FOR** statement is encountered, **ourtransport** is asked to execute its **FlyTo** method. The **Deploy** method waits for the **FlyTo** method to complete before it proceeds to its next statement.

### 13.7 Arbitrary Synchronization with Trigger Objs

Some processes will need to wait until a specified condition occurs. For these situations, MODSIM provides a special object type, **TriggerObj**, which, along with the **WAIT FOR** statement, allows a method to pause and wait until some condition occurs.

The syntax of the statement is:

```

WAIT FOR trigger object [ TO ] Fire
  Statement Sequence
[ ON INTERRUPT
  Statement Sequence ]
END WAIT;

```

When the **WAIT FOR ... Fire** statement is encountered, the method suspends and waits until the trigger object's **Trigger** method is invoked by some other method. At that time the statement sequence after the **WAIT FOR ... Fire** is executed. If the trigger object's **InterruptTrigger** method is invoked, the statement sequence after the **ON INTERRUPT** is executed, instead. A trigger object can have any number of methods waiting for it to **Trigger** or **InterruptTrigger**.

Taking the example of an **AircraftObj**, a refueling method might prudently wait until the plane is on the ground before requesting that the tanks be “topped off”, as in:

```

landedSignal : TriggerObj;
...
IF flying
  WAIT FOR landedSignal TO Fire      ← i.e. wait until some other
  END WAIT;                          method releases
END IF;                               trigger landedSignal
ASK airport TO assignRefueler(tankTruck);
WAIT FOR tankTruck TO refuel(SELF, fuelCapacity);
END WAIT;

```

### 13.8 Multiple Process Activities

To construct realistic simulation models, it is often necessary to model a physical object which can perform several operations simultaneously. A tank in a ground combat model, for instance, may be required to perform movement, communications and target acquisition activities simultaneously. Although this is a fairly common situation, it has traditionally been difficult to model, particularly when the activities may interact.

To support such models, MODSIM allows an object to do more than one activity at once. For example, a process object may be in the middle of one operation when it receives a message to perform a different, conflicting operation. In response, the object can:

- Interrupt the conflicting time-elapsing method which is waiting
- Ignore the new request
- Defer the new request.

### 13.9 Activity Tie-breaking, Time Advance and Activity Trace

It is sometimes necessary to arbitrate the order of activities (**TELL** methods) scheduled for identical simulation times and/or to be notified when simulation time is about to be advanced. To accomplish these fine-tuning controls, an object called **SimControlObj** has been provided in the MODSIM runtime library module **SimMod**.

In the case of tie-breaking, to specify which activity should be executed next, an instance of a **SimControlObj** derivative is created and its method **SetTieBreaking** is invoked with a **TRUE** argument. At any point during the simulation when two or more methods are scheduled for the current simulation time the **ChooseNext** method of the **SimControlObj** derivative instance will be invoked and passed a group containing the activity records (**ACTID** type) of all such methods. By overriding the **ChooseNext** method, users can select which method will be activated next. The owner object and method name of an **ACTID** may be obtained by using the **SimMod** procedures **ActivityOwner** and **ActivityName**, respectively, passing an **ACTID** as an argument. The user must return one of the **ACTID**'s from the group, which will be the next active method. Users may enable and disable this mechanism with calls to **SetTieBreaking** (**TRUE** and **FALSE** arguments, respectively) at any point in a simulation.

For example:

```
FROM SimMod IMPORT SimControlObj, ActivityGroup, Activity-
Name,
                    ActivityOwner;

TYPE
  MyControlObj = OBJECT(SimControlObj)
```

```

    OVERRIDE
      ASK METHOD ChooseNext(IN group: ActivityGroup) : ACTID;
    END OBJECT;

MyObj = OBJECT
  TELL METHOD tmeth1;
  TELL METHOD tmeth2;
END OBJECT;

OBJECT MyControlObj
  ASK METHOD ChooseNext(IN group: ActivityGroup) : ACTID;
  VAR
    act: ACTID;
  BEGIN
    OUTPUT("The following TELL methods scheduled activities");
    OUTPUT ("for the same time");
    FOREACH act IN group
      OUTPUT(ActivityName(act), " of ", OBJTYPENAME(ActivityOwner(act)));
    END FOREACH;
    RETURN group.Last;
  END METHOD;
END OBJECT;

OBJECT MyObj;
  TELL METHOD tmeth1;
  BEGIN
    OUTPUT("got to tmeth1");
  END METHOD;

  TELL METHOD tmeth2;
  BEGIN
    OUTPUT("got to tmeth2");
  END METHOD;
END OBJECT;
VAR
  ControlObj: MyControlObj;
  obj1, obj2: MyObj;
  . . .
  NEW(ControlObj);
  ASK ControlObj TO SetTieBreaking(TRUE); {turn tie-breaking on }

  NEW(obj1);
  NEW(obj2);

  TELL obj1 TO tmeth1 IN 10.0;

```

lation

```
TELL obj2 TO tmeth2 IN 10.0;
```

```
StartSimulation;
```

Time update notification is obtained similarly. An instance of a `SimControlObj` derivative is created and its method `SetTimeAdvance` is invoked with a `TRUE` argument. By overriding the `TimeAdvance` method, users will be notified when simulation time is about to change and be passed the value to which simulation time is to be set. Users may perform any desired behaviors within this method, including scheduling `TELL` methods for the current or later simulation time. As with tie breaking, notification may be enabled and disabled, by calls to `SetTimeAdvance` at any point in a simulation.

Finally, `SimControlObj` may be used to trace activity calls using the same mechanism which was used in tie-breaking and time advance. Just before an activity is activated or reactivated, the `'ActivityTrace'` method of `SimControlObj` is called allowing a step-by-step trace through the simulation. The `'SetActivityTrace'` method controls this feature.

**Note:** When the above features are used together, it is only necessary to derive one instance of `SimControlObj`.

### 13.10 Interrupting Activities

MODSIM has provisions for interrupting and stopping any or all activities prematurely. Any time-elapsing method can be interrupted by invoking the `Interrupt` procedure which takes as its parameters the object reference value of the object to be interrupted and the name of the particular method to be interrupted. The `Interrupt` procedure is imported from `SimMod`. For example:

```
Interrupt(Puma20, "ProceedTo");
```

An `Interrupt` does not take place immediately, but is scheduled like a `TELL` method. If you wish an `Interrupt` to take place immediately you should include a `WAIT 0.0` statement. Interrupting an activity that is waiting will cause it to execute the `ON INTERRUPT` clause of the `WAIT` statement. If there is no `ON INTERRUPT` clause, a run-time error will occur.

In MODSIM, every object maintains an `ActivityList` which is an ordered list of activities scheduled for that object. The activities are ranked by the time each activity is scheduled to finish its `WAIT`.

An activity record is placed on the object's activity list each time one of the object's time-elapsing methods executes a `WAIT`. The activity record contains all of the information needed to resume execution of a time-elapsing method after its `WAIT` is complete or it has been interrupted. Neither the pending list nor an object's activity list which con-

tains these activity records is directly accessible to the user. These are all maintained internally by MODSIM. The user's access to these facilities is through procedures (such as the **WAIT**, **Interrupt**, **TERMINATE**) and the facilities of the trigger object.

The **Interrupt** procedure scans the object's activity list and interrupts the most imminent activity which matches the given name. If there are no matches, nothing happens. We could do the following:

```
Interrupt(Puma20, "flyTo");
```

and the object instance **Puma20**'s **flyTo** method's **WAIT** would be interrupted.

If a method contains multiple **WAIT** statements, then whichever one is currently waiting is interrupted. If it is important to the user to conditionally control which **WAITs** are interrupted, then the method can be broken into separate methods for each activity, or a status can be set before each wait, and then checked by the interrupting code.

The **TERMINATE** statement is used by any time-elapsing method which wants to finish execution prematurely. It not only stops execution of the current method, but also **TERMINATES** the method which invoked it using a **WAIT FOR**. The effect of the **TERMINATE** is recursive. In other words the invoking routine becomes **TERMINATED** and therefore **TERMINATES** the method which invoked it. Like the **WAIT** statement, the **TERMINATE** statement may only appear within a **TELL** or **WAITFOR** method.

To summarize:

- The **Interrupt** procedure is used *from outside* an object's time-elapsing method to “wake up” the method before it completes the **WAIT**. The interrupted method resumes execution by performing the statement sequence after the **ON INTERRUPT**.
- The **TERMINATE** method is used *from inside* a process object's **TELL METHOD** or **WAITFOR** method to prematurely stop execution of the method (and the method which called this method, if it was invoked using the **WAIT FOR** construct).

### 13.10.1 Interrupting Methods and ACTID

The built-in type **ACTID** allows users to capture a reference to a specific activation of a **TELL** method. Bear in mind that **TELLing** an object to invoke a method schedules that method's execution. The method does not actually begin execution until its turn arrives in the pending activity list. Similarly, using the **WAIT FOR** construct schedules the invocation of an object's method.

If a user wishes to interrupt a particular invocation of a **TELL** method at a later time in the application, then a handle with which to reference this invocation must be retained by the user. This can be accomplished by declaring a variable or field to be of type **ACTID**

lation

and then using this variable as the left-hand side of an assignment statement and the desired **TELL** method invocation as the right-hand side.

For example:

```

VAR
    activity : ACTID;
BEGIN
    activity := TELL anObject TO doSomething; (1)
    ...

```

After executing (1) **activity** will contain a reference to the specific instance of method **doSomething** of object instance **anObject**. If the user wants to do a **WAIT FOR** this method, the **ACTID** reference may be substituted for the actual scheduling of the **TELL** method:

```

...
    WAIT FOR activity END WAIT;
...

```

As you can see, if **activity** were global or a field of an object, then more than one method could **WAIT FOR** the same invocation of **doSomething**. If an unassigned activity or an already completed activity is used in the **WAIT FOR**, a runtime error will result.

In order to interrupt this invocation of **doSomething** you can import the procedure **'InterruptMethod'** from the **MODSIM** library module **SimMod**. **InterruptMethod** takes an **ACTID** as its argument and will notify the method that it has been interrupted and schedule it to execute in the current simulation time.

Note that since **WAITFOR** methods may only be invoked by the **WAIT FOR** construct, they may not be used on the right hand side of an assignment statement. Consequently, **WAITFOR** method invocations may not be assigned to **ACTID** variables.

**THISMETHOD** is a built-in constant of type **ACTID** that can only be accessed within **TELL/WAITFOR** methods. For example:

```

TELL METHOD foo;
VAR
    a : ACTID;
BEGIN
    a := THISMETHOD;
    ...

```



## 14. Grouping Objects

---

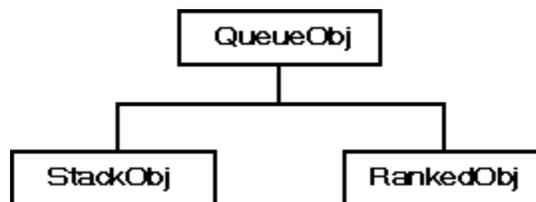
A language which makes use of dynamic data structures, such as objects, needs a way to group related objects in a disciplined way. This is especially true for simulations, which typically group objects queueing for a resource (the proverbial bank teller), or a series of events scheduled to happen at a specific time. Such associations are referred to as **groups** in MODSIM.

Objects may be selectively added to, or removed from a group. A MODSIM program can iterate through a group examining the members of that group. Groups are untyped so that they can hold a mixture of object types. An object can belong to any number of groups.

When groups are used, the ordering may be either implicit or explicit. The implicitly ordered, or ranked group, will always have one ordering for the same objects in the group, regardless of the order in which they were added. These are normally associated with ascending or descending sorts based on one or more fields, such as a list of activities sorted on initiation time.

Groups may also have an explicit ordering that depends on how the group is added to. The most common are the queue and stack groups which are FIFO and LIFO lists, respectively.

Here is the type hierarchy of MODSIM's built-in groups which can be imported from **GrpMod**.



**Figure 14-1. Built-in Groups**

### 14.1 Using Group Objects

It is often useful in applications to gather objects into some logical association. MODSIM's library support for this association mechanism is provided through the groups declared in **DGrpMod.mod**. There are a variety of grouping types: queue, stack, ranked, btree. Since these groups are provided as objects, you can easily derive your own object type and modify the behavior as required.

All groups in MODSIM are declared as proto-objects with the type of groups that can be added and removed being replaceable. These groups can, of course, be used "as is" and will default to an **ANYOBJ** type group. This means that the compiler will not check for

any particular type of object and will allow any object to be added to such a group and will assume that any assignment of return values from the group is correct.

Insertion/Removal Order:

```

QueueObj/StatQueueObj    : First-In-First-Out (FIFO)
StackObj/StatStackObj    : Last-In-First-Out (LIFO)
RankedObj/StatRankedObj  : User-Determined-In-First-Out
BTreeObj/StatBTreeObj   : Key-Determined-In-First-Out

```

User order groups: **RankedObj** and **BTreeObj**

A **RankedObj** group will insert new objects into the group based upon the user defined method **Rank**. In order to use a **RankedObj**, users must derive their own object from a **RankedObj** and then override the method **Rank**, as appropriate to the objects to be ordered in the tree. The **Rank** method must return one of -1, 0 or 1 depending on the relative ordering of its two arguments. If the first argument is to precede the second argument then -1 should be returned. If the arguments are considered equal, then 0 is returned. Otherwise, (first argument is to succeed second argument in order) the value 1 is returned.

Similarly, a **BTreeObj** is an ordered group of objects. The underlying structure of a **BTreeObj** is more efficient for groups that are to be added to, and deleted from, randomly and often. The group members must be identifiable with a **STRING** key, determined by the user. The key need not be unique, although this is the usually case. If the key is non-unique, subsequent insertions with the same key will be inserted after already present objects with that key. You determine the order of the **BTreeObj** by deriving your own object type from **BTreeObj** and overriding the **Key** method. This method must return a **STRING** which will be the key associated with the object being inserted. The argument to the **Key** method is an object reference of an object being added or specifically removed (**RemoveThis method**) from the group.

## 14.2 The Queue Group

The following methods are defined for the built-in **QueueObj** object type:

```

ASK METHOD Includes(IN candidate : ANYOBJ) : BOOLEAN;
ASK METHOD Add(IN NewMember : ANYOBJ); { behind Last }
ASK METHOD Remove(): ANYOBJ; { removes First }
ASK METHOD First() : ANYOBJ;
ASK METHOD Last() : ANYOBJ;
          { First ... candidate ... Last
            <- Prev | Next ->          }
ASK METHOD Next(IN candidate : ANYOBJ) : ANYOBJ;
ASK METHOD Prev(IN candidate : ANYOBJ) : ANYOBJ;

```

```

ASK METHOD RemoveThis(IN member : ANYOBJ);
ASK METHOD AddBefore(IN ExistingMember,
                    NewMember : ANYOBJ);
ASK METHOD AddAfter(IN ExistingMember,
                   NewMember : ANYOBJ);

```

Deletion of objects from the group can be accomplished with either the **Remove** or **RemoveThis** methods. The **Remove** method always deletes the first object in the group and returns a reference to the deleted member. The **RemoveThis** method takes an object reference as an argument and removes that particular object (regardless of its order within the group) from the group. If **RemoveThis** is passed **NILOBJ** as an argument or the object parameter is not a member of this group, a runtime error will occur.

The **Add** method places an object at the back end of the group, while **Remove** takes it from the front of the group. The exact insertion of items can be altered using the **AddAfter** or **AddBefore** methods, rather than the usual **Add**. **RemoveThis**, **AddBefore** and **AddAfter** can be used to circumvent the inherent FIFO discipline of this object. These methods will insert an object just after or just before another object already in the group. These methods cannot be used with **RankedObj** or **BTreeObj** as they would destroy the group's ability to do normal insertions properly.

**First**, **Last**, **Next** and **Prev** return reference values for those objects without changing the composition of the group.

The **Includes** method determines whether a specific object is part of a particular group *without traversing the group*. This is an important efficiency consideration. Each object keeps an internal list of groups to which it belongs. The **Includes** method interrogates this list, which is likely to be shorter than most groups, to determine its answer.

**QueueObj** also has defined the field **numberIn** which can be queried to determine the number of objects in a group.

### 14.3 The Stack Group

The **stackObj** type inherits all of the fields and methods of the **QueueObj**. It overrides the **QueueObj**'s **Add** method and substitutes an **Add** method which places objects at the front of the group instead of the back.

### 14.4 The Ranked Group

The **RankedObj** type inherits all of the fields and methods of the **QueueObj**. It overrides the **QueueObj**'s **Add** method and substitutes an **Add** method which inserts new objects into the group using a **Rank** method to determine the object's proper position:

```

ASK METHOD Rank(IN a, b: ANYOBJ) : INTEGER;

```

The user overrides the default **Rank** method and substitutes one which returns the following values: **-1** if **a < b**, **0** if **a = b** and **1** if **a > b**. The user decides how the comparisons, e.g. **a > b**, are to be made.

Since the **IN** parameters to the **Rank** method are of type **ANYOBJ**, the user will need to assign the referenced values to variables of the appropriate type before attempting comparison of any fields. As an example, the following implementation for method **Rank** could be used to rank a group of cargo objects according to their weight field:

```

CargoObj = OBJECT
  weight    : INTEGER;
  cube      : INTEGER;
  priority  : priType;
END OBJECT;

ASK METHOD Rank(IN a, b: ANYOBJ) : INTEGER;
VAR
  BoxA, BoxB: CargoObj;
BEGIN
  BoxA := a;  BoxB := b;
  IF ASK BoxA weight < ASK BoxB weight
    {Replace -1 with 1 if ordering is descending}
    RETURN -1;
  END IF;
  IF ASK BoxA weight > ASK BoxB weight
    {Replace 1 with -1 if ordering is descending}
    RETURN 1;
  END IF;
  RETURN 0;
  {Returning 0 means it is ranked after the last of
  others of the same value}
END METHOD;

```

This ranks the group of objects in increasing order, e.g., 1, 2, 3, 4, .... To reverse this order, the two **RETURN** statements would be switched. Of course, the user could provide a more elaborate **Rank** method which based the ranking on the values of more than one field.

## 14.5 Statistical Groups

In addition to the three basic group types (**QueueObj**, **StackObj**, **RankedObj**), three statistically accumulating groups have been added: **StatQueueObj**, **StatStackObj** and **StatRankedObj**. These new groups will acquire statistical data based upon the number of objects in the group, both with and without respect to time. Methods have been provided in order to easily provide this information.

Each group type described in the MODSIM library module **GrpMod** has a parallel statistical acquisition group described in the same module. The statistically capable groups all begin with **Stat** followed by the group type name. For example, **QueueObj** is mirrored by **StatQueueObj**.

Each of the statistic groups can provide the maximum and minimum number of objects ever in the group (methods **Maximum** and **Minimum**). The method **Count** counts the number of times that membership in the group has changed. The **Mean**, **StdDev** (standard deviation) and **Variance** may be obtained, as well as **WtdMean**, **WtdStdDev** and **WtdVariance**. The **Wtd** variety means that the statistic is based upon the length of simulation time the **numberIn** field was a particular value. In other words, the statistic is weighted with respect to time.

If users require other statistics or operations based upon the number of objects in a group, all statistically acquiring group objects have a **MONITORED INTEGER** field called **number**. The user may add their own monitors to this field to further capture relevant information and behavior (see **MONITORING** section).

These statistical groups function identically to the basic groups.

## 14.6 Iterating Through a Group

The **FOREACH** statement is the most general and efficient way to iterate through members of a group. However, if the user wants to go through the members without changing the group, the following construct is preferred:

```

VAR
    member : MyObject;
    group  : QueueObj;
    ...

BEGIN
    member := ASK group First();    (1)
    WHILE member <> NILOBJ
        (* perform tasks on member objects *)
        member := ASK group Next(member);
    END WHILE;

    (* ***** OR ***** *)

    member := ASK group Last();    (2)
    WHILE member <> NILOBJ
        (* perform tasks on member objects *)
        member := ASK group Prev(member);
    END WHILE;

```

This **WHILE** loop will go through all the members of a group successively assigning their reference values to the variable **member**. In (1) the iteration will go from the beginning to the end of the group (removal order), in (2) the iteration will go from the end to the beginning of the group (reverse removal order). A runtime error will occur if **NILOBJ** is passed to the method **Next** or **Prev**.

If the members of a group are to be successively removed from a group the following statements may be used:

```
VAR
    member    : MyObject;
    group     : QueueObj;

...

BEGIN
    WHILE ASK group numberIn > 0
        member := ASK group TO Remove();
        (* perform processing on 'member' *)
    END WHILE;
```

Although the examples demonstrate using `QueueObj`, any group object type or any object type derived from a group object type will behave in the same way.

# 15. Statistical Distributions: RandomObj

---

Random variables are available in MODSIM III through the library object **RandomObj**, which can be imported from **RandMod**. The programmer creates an object of type **RandomObj**, and queries it for successive random numbers.

The random numbers generated by the **RandomObj** are in the range:

$$0.0 < \text{Sample} < 1.0$$

The samples can also be drawn from a number of statistical distributions. The distributions which are supported are:

Distribution	Return Type
UniformReal	REAL
UniformInt	INTEGER
Exponential	REAL
Normal	REAL
Gamma	REAL
Beta	REAL
Triangular	REAL
Erlang	REAL
LogNormal	REAL
Weibull	REAL
Poisson	INTEGER
Binomial	INTEGER

There are support methods which can be used to set the seed, reset to the original seed, or to return the antithetic variate, e.g. **1 - Sample** instead of **Sample**.

Each time an instance of **RandomObj** is created, its **ObjInit** method will set it to a default seed. The sequence of random numbers drawn from **RandomObj** will always be the same given the same seed. In other words, the **RandomObj** returns a pseudo-random number stream.

The random numbers follow from the initial **seed** number which the **RandomObj** is given. A new seed may be set at any time. The following is a simple example:

```
MAIN MODULE Rand1;
FROM RandMod IMPORT RandomObj;
VAR OurRand : RandomObj;
    R1Num    : REAL;
    IntNum   : INTEGER;
BEGIN
    NEW(OurRand);
    FOR IntNum:= 1 TO 20
```

```

    R1Num := ASK OurRand UniformReal(-100.0, 100.0);
    OUTPUT(R1Num);
  END FOR;
END MODULE.

```

This program will print 20 **REAL** typed samples from the Uniform distribution in the range  $-100.0 < \text{Sample} < 100.0$ .

The random numbers are reproducible. The following example generates the same numbers twice with two different objects:

```

MAIN MODULE Rand2;
FROM RandMod IMPORT RandomObj;
VAR
  OurRand1 : RandomObj;
  OurRand2 : RandomObj;
  R1Num1   : REAL;
  R1Num2   : REAL;
  IntNum   : INTEGER;
BEGIN
  NEW(OurRand1);
  NEW(OurRand2);
  FOR IntNum := 1 TO 20
    R1Num1 := ASK OurRand1 Normal( 50.0, 4.0);
    R1Num2 := ASK OurRand2 Normal( 50.0, 4.0);
    OUTPUT(R1Num1, "    ", R1Num2);
  END FOR;
  DISPOSE(OurRand1);
  DISPOSE(OurRand2);
END MODULE.

```

MODSIM III uses the same multiplicative congruential pseudo-random number generator as SIMSCRIPT II.5. Its period is  $2^{31}$ . The behavior of the random number generator is the same on all machines on which MODSIM is run. Thus, the same random number streams will result wherever models are run.

There are ten different predefined random number streams, numbered 1..10. By default, instances of **RandomObj** have a seed from stream number 1. MODSIM also defines a stream number 0 which is identical to stream number 1. These predefined random number streams are identical to those provided in SIMSCRIPT II.5 and yield the same sequence of random numbers.

We can modify the above example by altering the seed of one of the streams, using the **SetSeed** method and the **FetchSeed** procedure:

```

MAIN MODULE Rand3;
FROM RandMod IMPORT RandomObj, FetchSeed;
VAR
  OurRand1      : RandomObj;
  OurRand2      : RandomObj;
  Seed2 : INTEGER;

```

```

R1Num1    : REAL;
R1Num2    : REAL;
IntNum    : INTEGER;
BEGIN
  NEW(OurRand1);
  NEW(OurRand2);
  Seed2 := FetchSeed(3); {Get seed from stream 3}
  ASK OurRand2 TO SetSeed(Seed2);
  FOR IntNum := 1 TO 20
    R1Num1 := ASK OurRand1 UniformReal(-100.0, 100.0);
    R1Num2 := ASK OurRand2 UniformReal(-100.0, 100.0);
    OUTPUT(R1Num1, "    ", R1Num2);
  END FOR;
  DISPOSE(OurRand1);
  DISPOSE(OurRand2);
END MODULE.

```

Note that the `setSeed` will take any positive `INTEGER` as a parameter. It is not necessary to use one of the ten predefined seeds.

MODSIM III also provides a non object-oriented random number generator through a procedure called `Random`. This procedure uses the particular machine's random number generator and will vary in the stream it provides on different machines.

Statistics gathering can also be easily accomplished by using the statistical monitor objects defined in `DStatMod.mod`. This module provides four basic statistical monitor objects. There are two types of these monitors, one that is not weighted with respect to time and another that is. Both types are each provided to monitor `INTEGERS` and `REALS`.

Also included in this module are some predefined types that will enable statistics acquisition: `SINTEGER`, `SREAL` (no time weighting), `TSINTEGER`, `TSREAL` (time weighting), `BINTEGER`, `BREAL` (both no time and time weighted accumulation). As the names imply, those types with the `INTEGER` suffix may be used to define `INTEGER` variables or fields and those types with the `REAL` suffix may be used to define `REAL` variables or fields.

These objects will provide basic statistical values for the monitored data point. If your application requires additional computations you may derive your own monitor from these monitor object types and define your own monitored type. As with any monitor object, statistical monitor objects may be specified either statically or dynamically for any variable or field declared to be monitored.

Here is an example using `DStatMod`:

```

FROM StatMod IMPORT SINTEGER, IStatObj;

VAR
  num: SINTEGER;

```

```
i: INTEGER;  
  . . .  
FOR i := 1 TO 4  
  OUTPUT ("Number?")  
  INPUT(num);  
END FOR  
OUTPUT ("average is ",  
  ASK (GETMONITOR (num, IStatObj)) Mean ());  
  . . .
```

## 16. Resource Objects

---

A common requirement in modeling applications is the notion of a blocking request for resource acquisition and the companion notion of releasing the resource back to the available pool. This mechanism is provided in MODSIM's library as the **ResourceObj** declared in **DResMod.mod**. Since it is provided as an object, any additional requirements such as resource preemption or resource contention may easily be added to an object derived from this class.

The **ResourceObj** has been designed and implemented to afford a great deal of functionality and still remain flexible enough to allow users to derive their own resource object types from it. **ResourceObj** is provided as a proto-object so that the user, if desired, can restrict the type of object which may request a particular resource.

A **ResourceObj** provides an asynchronous blocking mechanism, meaning that it allows simulation time to elapse while waiting for a resource. Resources are a finite pool of elements that may be acquired for some period of simulation time. Once acquired by an object, a resource is unavailable for subsequent requests until it is returned to the resource pool. Applications will usually attach specific meanings to resources, such as drive devices, machines, labor, etc.

The **ResourceObj** will automatically accumulate statistics and, if desired, plot a histogram on both allocation history and pending queue history. The default when this object is created is that statistics are turned off, but methods are provided which will individually turn on those statistics relevant to your application.

### 16.1 Acquiring Resources

Resources may be obtained using one of four methods: **Give**, **TimedGive**, **PriorityGive** and **GetResource**. Each of these methods takes at least two arguments: an object reference and the number of resources requested. All of the request methods must be invoked using the **WAIT FOR** invocation of methods so that the method can block until the resource is available. When the requesting method returns normally (i.e., is not interrupted) from the **WAIT FOR**, the object reference has acquired control of the requested number of resources and retains control of them until a **Take-Back** or **Transfer** method is executed. The **Give** and **PriorityGive** methods will always, and only, return normally to the requesting (blocked) method. However, the **TimedGive** and **GetResource** methods may return by interrupting the waiting method. In the case of an interrupt, the requesting object has NOT received the required resources within the specified simulation time interval and must proceed accordingly.

Users must provide an **INTERRUPT** section in the **WAIT FOR** statement if using the **TimedGive** or **GetResource** request methods. Otherwise, if a timeout occurs and the method is interrupted, a runtime error will result.

### 16.1.1 Differences between Request Methods

The **Give** request will block the requesting method until the requested resource(s) become available. This is the simplest and most common request method for **ResourceObjs**.

If the user has a situation where the request for the resource must be filled within a specific simulation time period then the **TimedGive** method should be used. This method will block the requesting method until the resource(s) are available or until the specified time period has elapsed. In the former case, the requesting method will return normally from the **WAIT FOR**. In the latter, the requesting method's **WAIT FOR** will be interrupted. This indicates to the requesting method that it has not received control of the requested resources and that the specified time period has elapsed.

Another, common requirement for resource acquisition is queuing requests based upon priority. If this is a requirement of your application, use the **PriorityGive** request method. This takes, in addition to the basic arguments, a **REAL** number priority. The higher the priority the more forward in the pending list the request will be placed. Naturally, if the request can be filled immediately, the priority makes no difference. This type of request will NOT preempt the resources from an object which has already acquired them. It is up to the user to coordinate such activity through the methods of the objects involved in such a transaction. The **Transfer** and **Cancel** methods can be useful for such a situation. Also, the user should be aware that if they are using a variety of request methods to the same resource object, any request method which does not require priority specification (i.e., **Give** and **TimedGive**) assumes a priority of 0.0.

Finally, the **GetResource** request method combines the timeout and priority properties of **TimedGive** and **PriorityGive** and allows the user to specify both a time period and priority for the request. Again, as in the **TimedGive**, the user must provide an **ON INTERRUPT** clause in the event that the request times out.

To summarize the different methods of resource acquisition:

**Give:** block until resource is available.

**TimedGive:** block until resource is available or specified simulation units have elapsed (timeout). If timeout occurs, requesting method will be interrupted.

**PriorityGive:** block until resource is available but queue object on pending list based upon priority value — higher numbers get more priority.

**GetResource:** a combination of **TimedGive** and **PriorityGive**.

## 16.2 Changing the Set of Resources

A **ResourceObj** assumes it has a homogeneous set of resources and that these resources are completely equivalent in that it does not matter which one is given to whom. To initiate the resource pool the user does a **Create** method giving the number of resources to begin with in the pool. This should be done before any requests are made. Otherwise, the **ResourceObj** will consider itself to have zero resources available and all requests will block.

The number of resources available from the pool is controlled by four methods:

- **Create**
- **IncrementResourcesBy**
- **DecrementResourcesBy**
- **TakeBack**

To increase the total pool number of resources the **IncrementBy** method is used. This will make the total number of resources ever available equal to the previous maximum available plus the increment. If any requests are pending at the time of the **IncrementBy**, they will be filled, if possible. To reduce the total number available use the **DecrementBy** method. This method will wait to gather as many resources as specified and remove them from the available pool. It will wait until they are returned to the pool if they are not immediately available, and it will decrement the pool before any pending or new requests are fulfilled.

The maximum number of resources is kept in the **ResourceObj** field **MaxResources**. The field **Resources** contains the currently available number of resources and the field **PendingResources** contains the number of resources pending filling (not the number of requests but the total number of resources requested).

Once a requesting method returns normally from a request for resources, the object reference given in the request is considered to have control of the resource(s). To return the resource(s) to the available pool, the **TakeBack** method is used giving the owner object reference and the number of resources being returned. An attempt to return more resources than assigned to an object will result in a runtime error. Rather than returning the resource(s) to the available pool, the user can decide to transfer the “ownership” of the resource(s) to another object. Again, any attempt to transfer more resources than owned by the object will result in a runtime error. The user must provide code to take care of letting the receiving object know it now has control of the resources and the relinquishing object no longer has control. This is not done by **ResourceObj**.

If a user wishes to revoke a request for all or some of requested resources by a particular object, the **Cancel** method is provided. This method will remove the specified number of resources from the request of an object on the pending list. Again, the user must pro-

vide and invoke the necessary methods to notify the requesting object of the change in its request status.

### **16.3 Statistics of Resources**

**ResourceObjs** have the ability to keep statistics on both allocated and pending resource requests. By default, statistical accumulation is off. To activate it for the allocation list use **SetAllocStats** and pass the **TRUE** value. For the pending list use **SetPendStats** and pass the **TRUE** value.

Either of these lists can be set up to accumulate histograms using either **SetAllocHistogram** or **SetPendingHistogram**.

## **Section IV. Input / Output**



# 17. Input / Output

---

There are a number of ways in which to do input and output in MODSIM. We have already seen the free-format **INPUT** and **OUTPUT** statements which allow simple, unformatted input and output to the default I/O device (usually a CRT and its keyboard).

MODSIM also provides a standard library module, **IOMod**, which contains a stream I/O object called **StreamObj**. This object allows the user to stream oriented input and output to other devices and files. **IOMod** also contains a number of support routines which interface with the machine's file system. MODSIM also supports two other modes of I/O: random access file I/O and indexed sequential file I/O.

At the end of this chapter a few of the I/O routines from **IOMod** are discussed.

## 17.1 INPUT & OUTPUT Statements

MODSIM III provides two standard built-in procedures, **INPUT** and **OUTPUT** for doing non-object oriented, free formatted I/O.

The **INPUT** procedure takes one or more arguments. The **OUTPUT** procedure takes zero or more arguments. The arguments may be any of the following types:

**INTEGER, REAL, CHAR, STRING**

For example:

```
OUTPUT("Input height & weight for item number", n);
INPUT(height, weight);
OUTPUT;
```

In addition to the types listed above, the **OUTPUT** statement accepts and correctly handles values of user defined enumerated types.

The **INPUT** procedure reads values for each argument from the system's standard input, if it exists. The **OUTPUT** procedure writes the value of each argument to the system's standard output, followed by a newline character. When it is used without arguments, it writes a newline character alone.

You can include an object reference in an **OUTPUT** statement. By default the hexadecimal address of the object will be printed. If the user supplies an **ObjPrint** method then the result of this method will be printed:

```
myObj = OBJECT
  Name : STRING
  ASK METHOD ObjPrint() : STRING;
  ASK METHOD SetName(IN TestString : STRING);
```

```

    . . .
END OBJECT

ASK METHOD ObjPrint : STRING;
BEGIN
    RETURN Name;
END METHOD;

VAR
    obj : MyObj
BEGIN
    . . .
    NEW(obj);
    ASK obj TO SetName("foo");
    OUTPUT(obj);
    . . .

```

The result will be to print “foo”.

MODSIM III also provides two procedures for constructing formatted strings:

```

    PRINT [(expressionlist)] WITH formatstring
    SPRINT [(expressionlist)] WITH formatstring

```

Both will generate a string based upon the provided formatting string. The formatting string contains a field specification which indicates field width and, in the case of **REALS**, precision. The **PRINT** function will automatically output the constructed string to **stdout** with an appended **newline**. **SPRINT** will return the constructed string as a MODSIM **STRING** type which you may use, as appropriate.

A format string may be a literal, constant or variable **STRING**. The formatting is indicated by embedding field specifications within the string. Asterisks (\*) are used to indicate field widths and, in the case of **REALS**, precision. Fields may be left, right or centered justified. The defaults are that numbers and strings are right justified. You may override or explicitly indicate this by making the last character of your field specification one of '<', '>' or '~' for left, right or centered, respectively.

Here is a sample of code using **PRINT**:

```

CONST
    format="    *****    ***.**    ***<    ";
VAR
    s : STRING;
    r : REAL;
    i : INTEGER;
    str : STRING;

```

```

BEGIN
  s := "values";
  r := 32.854;
  i := 28;
  PRINT (s,r,i) WITH format;
  str := SPRINT (s,r,i) WITH format;

```

## 17.2 Stream I/O Using StreamObj

In MODSIM stream I/O is implemented using the `StreamObj` object. Files may be read from or written to, but it is not possible to reposition to an arbitrary position in the file. Below is a simple example of its use in which a text file called “sample.txt” is read and printed out.

```

FROM IOMod IMPORT StreamObj,
FileUseType(Input);

VAR
  Strm      : StreamObj;
  textLine : STRING;

  . . .

  NEW(Strm);
  ASK Strm TO Open("sample.txt", Input);
  WHILE NOT (ASK Strm eof)
    ASK Strm TO ReadLine(textLine);
    OUTPUT(textLine);
  END WHILE;
  ASK Strm TO Close;
  DISPOSE(Strm);

  . . .

```

The following paragraphs contain a partial list of the methods and procedures associated with `StreamObj`.

### 17.3 ASK Methods of StreamObj

**Open**(IN FileName: STRING; IN IOdirection: FileUseType)

Description: Opens the specified filename for the specified use. There are three special filenames: `stdin`, `stdout`, and `stderr`. `stderr` is the system default device for error messages.

**Close**

Description: Closes the file associated with the object.

**Delete**

Description: Deletes the file associated with the object.

**ReadChar(OUT ch: CHAR)**

Description: Reads a character from the stream.

**ReadInt(OUT n: INTEGER)**

Description: Reads an integer from the stream.

**ReadReal(OUT x: REAL)**

Description: Reads a **REAL** value from the input file/device. Can read exponential notation as well as standard real notation.

**ReadString(OUT str: STRING)**

Description: Reads characters up to, but not including, the next space, tab, carriage return or end of file.

**ReadLine(OUT str: STRING)**

Description: Reads a character string from the current position up to, but not including, the newline character.

**WriteChar(IN ch: CHAR)**

Description: Writes a character to the stream.

**WriteInt(IN num, fieldwidth: INTEGER)**

Description: Writes an integer number to the stream.

**WriteHex(IN num, fieldwidth: INTEGER)**

Description: Writes an integer in hexadecimal notation.

**WriteReal(IN num: REAL; IN fieldwidth, precision: INTEGER)**

Description: Writes a real number to the stream.

**WriteString(IN str: STRING)**

Description: Writes a string to the stream.

**WriteLn**

Description: Writes a newline character to the stream.

## 17.4 Procedures of IOMod

**ExistsFile(IN fname: STRING): BOOLEAN**

Description: Returns **TRUE** if the file exists, **FALSE** if not. **fname** can be a full path or a filename in the current directory.

**DeleteFile(IN fname: STRING)**

Description: Deletes the specified file, if it exists.

**FileSize(IN fname: STRING): INTEGER**

Description: Returns the size of the specified file, in bytes.

**ReadKey(): CHAR**

Description: Reads one character from the console with no echo. It does not require a Carriage Return, Enter or Newline before returning the character.

Each Stream I/O object also has two fields which report status:

```
eof      : BOOLEAN;  
ioResult : INTEGER;
```

The **eof** field, which signifies end-of-file, becomes **TRUE** as soon as the last item has been read from the file and nothing remains to be read. The **ioResult** field takes on a value which indicates the status of the last I/O activity. A normal completion, with no error, leaves a value of zero in this field. The current implementations place a non-zero value in the field to indicate an error.



## 18. Graphics and Animation

---

Graphics capabilities of SIMGRAPHICS II are available for use with MODSIM. They include the following features:

- Animation:** Icons created off-line using the graphical editor move and orient themselves according to simulation logic.
- Graphic Editor:** Used to create and edit icons, input forms, graphs and charts.
- Presentation Graphics:** Used to create charts and graphs off-line using the graphical editor. The charts and graphs are then tied to variables in the program.

You will use these features through an object-oriented interface to the various graphic objects.

These advanced graphical development capabilities are fully documented in the *SIMGRAPHICS II User Manual*.



# Appendices



## Appendix A. Glossary

---

<b>activity:</b>	A <b>WAIT</b> statement in a <b>TELL</b> or <b>WAITFOR</b> method. The place in an object's <b>TELL</b> or <b>WAITFOR</b> method where simulation time elapses.
<b>base type:</b>	With respect to objects: The immediate ancestor or the immediate underlying object type of an object type.  With respect to arrays: The type of each element in the array.
<b>behavior:</b>	A method of an object implements the object's behavior.
<b>component:</b>	Either a field or method for an object.
<b>conflicting methods:</b>	This occurs when two or more of the base types in a multiple inheritance have a method with the same name.
<b>derived type:</b>	An object type defined in terms of one or more existing object types.
<b>dynamic binding:</b>	The type of each operand and operation is determined at run-time; most object-oriented languages, including MODSIM, are based on dynamic binding. MODSIM uses dynamic binding only for field references and method calls, not for other operations such as <b>+</b> , <b>-</b> , <b>AND</b> , etc.
<b>dynamic data type:</b>	One of: <b>ARRAY</b> , <b>RECORD</b> , <b>OBJECT</b> . The memory for an instance of each of these types is explicitly allocated and deallocated by the programmer using the <b>NEW</b> procedure and deallocated using the <b>DISPOSE</b> procedure.
<b>encapsulation:</b>	Packaging the fields which define the state of an object and the methods which define its behaviors within one object definition.
<b>enumeration:</b>	A user-defined <u>ordered</u> set of literal values - e.g. <b>workday = ( Mon, Tue, Wed, Thu, Fri )</b>
<b>field:</b>	One of the variables associated with a particular object or record type.

<b>fixed data type:</b>	One of: <b>INTEGER</b> , <b>REAL</b> , <b>STRING</b> , <b>CHAR</b> , <b>BOOLEAN</b> , enumeration, subrange, <b>FIXED ARRAY</b> , <b>FIXED RECORD</b> . These data types are automatically allocated and deallocated on entry to and exit from the block in which their variables are declared.
<b>function method:</b>	A method which returns a value. Only <b>ASK</b> methods can return a value. <b>TELL</b> and <b>WAITFOR</b> methods cannot be function methods.
<b>function procedure:</b>	A procedure which returns a value.
<b>group:</b>	A structure used to associate objects. There are three basic group object types: <b>StackObj</b> , <b>QueueObj</b> , <b>RankedObj</b> .
<b>inheritance:</b>	The definition of one object type in terms of another, already-existing object type.
<b>instance:</b>	One particular array of an array type. One particular record of a record type. One particular object of an object type.
<b>invoke:</b>	To call a procedure or method. To cause a procedure or method to execute.
<b>member:</b>	An object which is contained within a group.
<b>message:</b>	The name of a method; “sending message A to B” is an equivalent way of saying “ask object B to perform method A” or “perform method A with object B”.
<b>method:</b>	A routine which describes an object's behavior. Similar to a procedure, however a method is <b>always</b> associated with an object.
<b>object:</b>	A dynamic data structure that includes an associated list of methods.
<b>ordinal type:</b>	The subset of scalar types which have a <u>known</u> ordering. In other words, given one value which belongs to the type, it is possible to state what the next or previous value would be. The following are ordinal types: <b>INTEGER</b> , <b>CHAR</b> , <b>BOOLEAN</b> , enumerations and subranges.

<b>pass by reference:</b>	When a parameter in a parameter list is shared by both the invoking and the invoked routine. Parameters with the <b>INOUT</b> and <b>OUT</b> qualifier are passed by reference.
<b>pass by value:</b>	When a copy of a parameter in a parameter list is made and passed in to the invoked routine. Parameters with the <b>IN</b> qualifier are passed by value.
<b>private property:</b>	A property with a scope limited to the methods of an object type or derived object types.
<b>process:</b>	Process-based simulations allow methods of objects to describe a series of related activities rather than being limited to defining simply one event per method.
<b>proper method:</b>	A method that has no return value. Can be either a <b>TELL</b> , <b>ASK</b> or <b>WAITFOR</b> method.
<b>proper procedure:</b>	A procedure that has no return value.
<b>property:</b>	A characteristic or attribute of an object type. Specifically either a method or field of the object type.
<b>public property:</b>	A property of an object that is available for use outside the methods of that object type.
<b>qualified inherited call:</b>	In a multiple inheritance, an invocation of an inherited method of a specific base type, as in :  <b>INHERITED FROM SomeObject aMethod;</b>
<b>record:</b>	A data structure which consists of a collection of fields which may be variables of differing types.
<b>reference type:</b>	Each array, record and object type has an implicit reference type, which is used to define variables that refer to a specific instance of that type - analogous to a pointer type.
<b>reference variable:</b>	A variable that references a specific instance of an array, record or object type; a variable of the reference type.
<b>routine:</b>	A general term for a sub-routine, procedure, function or method.

- scalar type:** A type which has only one element or component part and can be used to scale, measure or quantify things. The following are scalar types: **INTEGER**, **REAL**, **CHAR**, **BOOLEAN**, enumerations and subranges. An example of something which would not be a scalar type is an array, record or object type.
- shared variable:** A variable which is shared by all the methods of a particular object type. In other words, a variable defined outside the scope of an object so that it will be visible to all instances of that object type. Usually a shared variable is defined globally, within a module.
- simple data type** One of the following types: **INTEGER**, **REAL**, **CHAR**, **BOOLEAN**, **STRING**, enumerations, subranges.
- strong typing:** The type of each operand, parameter and operation is fixed at compile-time. MODSIM, Ada and Pascal are characterized by strong typing.
- structured data type:** One of the following aggregate types: **ARRAY**, **RECORD**, **OBJECT**, **FIXED ARRAY** or **FIXED RECORD**.
- TELL method:** A proper method which is executed asynchronously. It can elapse simulation time. If it has a parameter list, only **IN** parameters are allowed. **WAIT** statements are allowed in **TELL** methods.
- time-elapsing method:** A **TELL METHOD** which contains at least one **WAIT** statement.
- underlying type:** If type **A** is derived from type **B**, or from some type which is in turn derived from **B**, then **B** is said to be an underlying type of **A**.
- WAITFOR method:** A proper method which is executed asynchronously. It can elapse simulation time. Unlike a **TELL** method, a **WAITFOR** method's parameter list allows **OUT** and **INOUT** parameters. **WAIT** statements are allowed in **WAITFOR** methods.

## Appendix B. Reserved Words

---

The following is a complete list, with descriptions, of the reserved words in MODSIM III.

### ACTID

Example: `act : ACTID`  
`. . .`  
`act := TELL obj TO GoTo(x, y);`

Description: Built-in type which is used to represent simulation activities. Routines in the MODSIM run time library (SimMod) can interpret this data type.

### ALL

Example: `FROM SomeModule IMPORT ALL Colors;`

Description: Specifies that all enumerated constants of the enumerated type are to be imported.

### AND

Example: `Expr1 AND Expr2`

Description: A **BOOLEAN** operator. If both **BOOLEAN** expressions are **TRUE** then the entire expression is **TRUE**. If either **BOOLEAN** expression is **FALSE**, the entire expression is **FALSE**. If the first expression is **FALSE**, the second condition is not evaluated.

### ANYARRAY

Example: `anyAr : ANYARRAY;`  
`ar : ARRAY INTEGER OF STRING;`  
`. . .`  
`anyAr := ar;`  
`ar := anyAr;`

Description: Built-in type which can be used to represent any array type. It overcomes MODSIM's strict typing.

**Note:** Use with caution.

### ANYOBJ

Example: `PROCEDURE foo(IN n : ANYOBJ);`  
`. . .`  
`foo(Obj);`

Description: Built-in type which can be used to represent any object type. It overcomes MODSIM's strict type checking.

**Note:** Use with caution.

**ANYREC**

Example: `n : ANYREC;`  
`rec : MyRec;`  
`. . .`  
`rec := n;`

Description: Built-in type which can be used to represent any record type. It overcomes MODSIM's strict type checking.

**Note:** Use with caution.

**ARRAY**

Example: `VAR x : ARRAY INTEGER OF REAL;`

Description: Declares an array type with the given index type and element type.

**AS**

Example: `IMPORT StreamObj FROM IOMod AS OutputObj;`

Description: Changes the name of the imported definition.

**ASK**

Example: `ASK Obj1 TO MoveForward;`

Example: `Pos := ASK Tank1 CurrentPos;`

Description: References a field or invokes an **ASK** method of the specified object. Since **ASK** methods are not allowed to elapse simulation time, the invoked method will be completed before program control passes to the next statement.

Example: **TYPE**

`CarObj = OBJECT`  
`ASK METHOD Move(IN x, y : INTEGER);`

Description: Part of the method heading for an **ASK** method.

Example: **OBJECT** CarObj

`ASK METHOD Move(IN x, y : INTEGER);`

Description: Part of the method declaration within an object block.

**BEGIN**

Example: `MAIN MODULE MainMod; ....BEGIN...END MODULE.`

Example: `BEGIN ... END PROCEDURE;`

Description: Identifies the beginning of a sequence of executable statements.

**BOOLEAN**

Example: `isDone : BOOLEAN;`

```

. . .
IF isDone
. . .
END IF;
```

Description: Built-in type which is used to represent either **TRUE** or **FALSE**.

**BY**

Example: `FOR i := 0 TO 20 BY 2`

Example: `FOR i := 20 DOWNTO 0 BY 2`

Description: Optional qualifier which describes the size of the increment in a **FOR** statement. The control variable of the loop may be of any ordinal type, but the increment must be an integer expression.

**CALL**

Example: `CALL procvar(x,y);`

Description: Invokes the procedure assigned to `procvar` passing the optional argument list.

**CASE**

```

Example: CASE NewCar
          WHEN Saab, Chrysler:
            OUTPUT("Family car");
          WHEN Porsche:
            OUTPUT("Sports car");
          OTHERWISE
            OUTPUT("A what?");
          END CASE;
```

Description: Defines a multiple branch conditional statement. The expression after the word **CASE** is evaluated. If its value matches any of the choices after the word **WHEN**, that statement sequence is executed. If the value doesn't match any choice, the statement sequence after the **OTHERWISE** is executed.

**CHAR**

Example: `ch : CHAR;`  
`. . .`  
`ch := 'i';`

Description: Built-in type which is used to represent single characters.

**CLASS**

Example: `TYPE`  
`Obj = OBJECT`  
`CLASS`  
`f : INTEGER`  
`ASK METHOD foo;`  
`END OBJECT;`  
`. . .`  
`ASK Obj TO foo; { no instances have been created }`  
`i := Obj.f;`  
`. . .`

Description: Defines a section in which the fields and methods are independent of any particular instance of the object, and may be referenced as such.

**CONST**

Example: `CONST`  
`Sky = blue; pi = 3.14159;`

Description: Precedes a series of constant declarations. The type of the constant depends on the type of the literal or expression used to define it.

**DEFINITION**

Example: `DEFINITION MODULE Transport;`  
`. . .`  
`END MODULE.`

Description: Identifies the module as a **DEFINITION** module, in which variables, object types, etc. are described.

**DIV**

Example: `b := 7 DIV 2;`

Description: Integer division operator. In the example, **B** will be set to 3. See also **MOD**.

**DOWNTO**

Example: `FOR k := 20 DOWNTO 0 BY 2`

Description: This indicates that a **FOR** loop's control variable should be decremented rather than incremented after each iteration. Note that the increment amount is always stated as a positive integer.

**DURATION**

Example: **WAIT DURATION 4.0**

Description: Indicates execution of a **TELL** or **WAITFOR** method should be suspended for the specified amount of simulation time, unless interrupted. In the example above, the **WAIT** pauses for **4** units of simulation time.

**ELSE**

Example: **IF Door = Open**  
           **OUTPUT("Door was open.");**  
**ELSE**  
           **OUTPUT("Door was closed.");**  
**END IF;**

Description: If the Boolean expression evaluates to **FALSE**, the statement block following **ELSE** is executed.

**ELSIF**

Example: **IF fuelLevel > 12500**  
           **status := ContinueMission;**  
**ELSIF fuelLevel > 3500**  
           **status := ReturnToBase;**  
**ELSE**  
           **status := LowFuelEmergency;**  
**END IF;**

Description: Included in an **IF** statement to allow multiple conditions.

**END**

Example: **BEGIN ... END PROCEDURE;**

Example: **WHILE... END WHILE;**

Description: Marks the end of a control statement, structure declaration, block or module. Always followed by an identifier which specifies what is being ended, e.g. **END FOR**, **END MODULE**, **END METHOD**, **END RECORD**, **END OBJECT**, etc.

**EXIT**

Example: **LOOP**  
           **...**  
           **IF n > 37**

```

        EXIT;
    END IF;
    ...
END LOOP;

```

Description: The **EXIT** statement may be used to break out of any of the loop statements: **WHILE**, **REPEAT**, **FOR**, or **LOOP**.

#### **FALSE**

```

Example:  VAR
          b : BOOLEAN;
          . . .
          b := FALSE;
          . . .

```

Description: One of the two **BOOLEAN** constants; the other being **TRUE**.

#### **FIXED**

```

Example:  arrType = FIXED ARRAY [1..10] OF REAL;

```

Description: Indicates that the array is a fixed rather than a dynamic array.

```

Example:  recType = FIXED RECORD
          name : STRING;
          age  : INTEGER;
          END RECORD;

```

Description: Indicates that the record is a fixed rather than a dynamic record.

#### **FOR**

```

Example:  FOR k := 1 TO 5 ... END FOR;

```

Description: A **FOR** loop repeats the enclosed statement sequence until the loop control variable exceeds the terminating value. If no **BY** statement clause is included, the step defaults to 1.

```

Example:  WAIT FOR SomeObj TO SomeMethod;

```

Description: One of the three optional forms of the **WAIT** statement, in which one method waits for another activity to complete.

#### **FOREACH**

```

Example:  FOREACH obj IN myQueueObj
          . . .
          END FOREACH;

```

Description: A construct to allow iteration over a group of objects or records, usually the group is one derived from **GrpMod** (for objects) or **ListMod** (for records).

**FORWARD**

Example: `PROCEDURE Recurse(IN n : INTEGER); FORWARD;`

Description: Used to declare the existence of a procedure before the full declaration is specified. This is useful when routines have a cyclic calling pattern.

Example: `TYPE`

`SomeThing = OBJECT; FORWARD;`

Description: Used to declare the existence of an object type before its full declaration so it can be referred to by another object. Useful when two or more object types have fields which refer to each other.

**FROM**

Example: `FROM GrpMod IMPORT QueueObj;`

Description: Specifies the definition module from which a definition is to be imported.

Example: `INHERITED FROM SomeObj SomeMethod;`

Description: Specifies the base object type from which an inherited method is to be invoked.

**IF**

Example: `IF a = 3`  
`OUTPUT(a);`  
`END IF;`

Description: If the Boolean expression is **TRUE**, the following statement sequence will be executed. If it is **FALSE**, the next clause (if any) will be executed (see **ELSIF**, **ELSE**).

**IMPLEMENTATION**

Example: `IMPLEMENTATION MODULE Transport;`  
`...`  
`END MODULE.`

Description: Identifies the module as an **IMPLEMENTATION** module.

**IMPORT**

Example: `FROM SimMod IMPORT ProcessObj;`

Description: Imports the item named by the identifier from the specified module and adds its definition to the scope of the importing module.

**IN**

Example: **PROCEDURE PrintIt(IN textLine: STRING);**

Description: The **IN** qualifier appears in a formal parameter list and specifies the direction in which data will flow. **IN** parameters are passed by value.

Example: **TELL SomeObj TO SomeMethod IN 10.0;**

Description: The **IN** qualifier specifies that the method should be invoked in that many units of simulation time.

**INHERITED**

Example: **ASK METHOD ProceedTo;**

```
BEGIN
    . . .
    INHERITED ProceedTo;
    . . .
END METHOD;
```

Description: When a new object type is derived from an existing type, and a method is overridden in the new object, the old method code is still available, and a call qualified with the word **INHERITED** can be used to invoke it.

**INOUT**

Example: **PROCEDURE Capitalize(INOUT text: STRING);**

Description: This declares a formal parameter to be both an input and an output parameter to a routine. It is passed by reference.

**INTEGER**

Example: **i : INTEGER;**

```
. . .
i := 5 + j;
```

Description: Built-in type which is used to represent integers (whole numbers).

**INTERRUPT**

Example: **WAIT DURATION 3.0**

```
    OUTPUT("Wait completed");
ON INTERRUPT
    OUTPUT("Wait was interrupted");
END WAIT;
```

Description: If the **WAIT** statement is interrupted, the **ON INTERRUPT** clause is executed.

**LMONITOR**

Example: `StrMonObj = MONITOR STRING OBJECT`  
`. . .`  
`LMONITOR METHOD laccess;`  
`. . .`  
`END OBJECT;`

Description: Declares a method within a monitor object which is called automatically just before any attempt is made to change the value of a monitored variable.

#### LMONITORED

Example: `StrMonVar = LMONITORED STRING BY StrMonObj;`  
`. . .`  
`str : StrMonVar;`

Description: Declares a variable type to be left monitored. The **LMONITOR** methods of any monitor objects attached to this variable will be called before the variable changes value.

#### LOOP

Example: `LOOP`  
`. . .`  
`IF n > 37`  
`EXIT;`  
`END IF;`  
`. . .`  
`END LOOP;`

Description: The enclosed code will repeat until an **EXIT** statement is executed.

#### LRMONITORED

Example: `StrMonVar = LRMONITORED STRING BY StrMonObj;`  
`. . .`  
`str : StrMonVar;`

Description: Declares a variable type to be left and right monitored. The **LRMONITOR** methods of any monitor objects attached to this variable will be called before the variable changes value or is read.

#### MAIN

Example: `MAIN MODULE AirportModel;`  
`. . .`  
`END MODULE.`

Description: Identifies the module as the **MAIN** module of a program.

**METHOD**

Example: **ASK METHOD Shoot(IN Angle: REAL);**

Description: Keyword for a method heading.

**MOD**

Example: **IntNum:= 7 MOD 2;**

Description: **MOD** is used to obtain the “remainder” of an integer division. In the example above, the **INTEGER** variable **IntNum** will be set to 1.

**MODULE**

Example: **MAIN MODULE AirportModel;**

**...**

**END MODULE.**

Description: Used to delimit a module.

**MONITOR**

Example: **RealMonObj = MONITOR REAL OBJECT**

**...**

**END OBJECT;**

Description: Used to declare a monitor object, in which **LMONITOR**, **RMONITOR** and **LRMONITOR** methods may be declared.

**NEW**

Example: **NEW(obj);**

**NEW(rec);**

**NEW(arr, 1..10);**

Description: Used to create instances of dynamic data types, objects, records and arrays.

**NILARRAY**

Example: **IF arr = NILARRAY**

**OUTPUT("unallocated array");**

**END IF;**

Description: The value of an array reference variable before it is created.

**NILOBJ**

Example: **IF obj = NILOBJ**

**OUTPUT("unallocated object");**

**END IF;**

Description: The value of an object reference variable before it is created.

**NILREC**

Example: `IF rec = NILREC  
    OUTPUT("unallocated record");  
END IF;`

Description: The value of a record reference variable before it is created.

**NONMODSIM**

Example: `PROCEDURE foo; NONMODSIM;`

Description: Specifies that a procedure heading in a **DEFINITION** module defines a routine which will be provided in C++.

**NOT**

Example: `IF NOT (k = 3) ...`

Description: Inverts **TRUE** and **FALSE** in a **BOOLEAN** expression.

**OBJECT**

Example: `TYPE  
    Boat = OBJECT  
    ...  
    END OBJECT;`

Description: Used to delimit an object type declaration.

Example: `OBJECT Boat;  
    ...  
END OBJECT;`

Description: Used to delimit an object declaration.

**OF**

Example: `VAR x : ARRAY [0..10] OF REAL;`

Description: Indicates the type of the elements of an **ARRAY**.

**ON**

Example: `ON INTERRUPT ...`

Description: Optional part of a **WAIT** statement which precedes the code to be executed when a **WAIT** statement is interrupted.

**OR**

Example: `( x < 3.5 ) OR ( n > 5 )`

Description: A **BOOLEAN** operator. If either or both of the **BOOLEAN** expressions are **TRUE**, the expression will be **TRUE**. If the first condition is **TRUE**, the second condition is not evaluated.

**OTHERWISE**

Example: **CASE NewCar**  
           **WHEN Saab, Chrysler:**  
             **OUTPUT("Family car");**  
           **WHEN Porsche:**  
             **OUTPUT("Sports car");**  
           **OTHERWISE**  
             **OUTPUT("A what?");**  
           **END CASE;**

Description: See **CASE**. This identifies the “default” case in a **CASE** statement.

**OUT**

Example: **PROCEDURE CurrentTime(OUT Time: INTEGER);**

Description: Declares a formal parameter of a routine to be for output only. The parameter is passed by reference, and is initialized on entry to the called routine.

**OVERRIDE**

Example: **TYPE**  
           **Bicycle = OBJECT**  
             **...**  
           **OVERRIDE;**  
             **ASK METHOD GOTO(IN x, y : INTEGER);**  
             **...**  
           **END OBJECT;**

Description: Indicates that an inherited method is to be overridden. The new method is then specified in the object block.

**PRIVATE**

Example: **Boat = OBJECT**  
           **...**  
           **PRIVATE**  
             **ASK METHOD Report(IN rpt: STRING);**  
             **status: INTEGER;**  
             **...**  
           **END OBJECT;**

Description: Declares methods and fields to be accessible only from within the object's own methods.

**PROCEDURE**

Example: **PROCEDURE PrintIt(IN text: STRING);**

Description: Keyword for a procedure heading.

**PROTO**

Description: Redundant in MODSIM III.

**REAL**

Example: `rNum : REAL;`

`. . .`

`rNum := 5.635`

Description: Built-in type which is used to represent floating point (fractional numbers).

**RECORD**

Example: `TYPE`

`CustFil = RECORD`

`Age: INTEGER;`

`Name: STRING;`

`END RECORD;`

Description: Used to define a record type. A record is a collection of fields which may be accessed as a group, or individually by referring to a specific field name.

**REPEAT**

Example: `REPEAT`

`INC(k);`

`OUTPUT(k);`

`UNTIL k = 5;`

Description: Repeat the enclosed code until the **BOOLEAN** expression is **TRUE**. The **BOOLEAN** expression is evaluated after each iteration.

**RETURN**

Example: `PROCEDURE Sum(IN i,j : INTEGER): INTEGER;`

`BEGIN`

`RETURN i+j;`

`END PROCEDURE;`

Description: The **RETURN** statement is used to exit from a function procedure and specify the function result. When used in a proper procedure without a result argument, it simply exits the procedure.

**REVERSED**

Example: `FOREACH obj IN myQueueObj REVERSED`

`. . .`

**END FOREACH;**

Description: Used in a **FOREACH** statement to specify the iteration will proceed from the last item to the first.

### **RMONITOR**

Example: **ChMonObj = MONITOR CHAR OBJECT**

```

    . . .
    RMONITOR METHOD raccess;
    . . .
END OBJECT;

```

Description: Declares a method within a monitor object which is called automatically just before any attempt is made to access the value of a monitored variable.

### **RMONITORED**

Example: **ch : RMONITORED CHAR BY ChMonObj;**

Description: Declares a variable type to be right monitored. The **RMONITOR** methods of any objects attached to this variable will be called before the variable is read.

### **SELF**

Example: **IF obj = SELF**

```

    . . .
END IF;

```

Description: An constant object reference variable which represents the object instance of the current method.

### **STRERR**

Example: **i := STRTOINT("abc"); { STRERR is TRUE }**  
**i := STRTOINT("135"); { STRERR is FALSE }**

Description: Represents the status of the last **STRTOINT** or **STRTOREAL** built-in procedure.

### **STRING**

Example: **str : STRING;**

```

    . . .
str := "hello" + "world";

```

Description: Built-in type which is used to represent a string (sequence of characters). Memory management of **STRING** is handled automatically by MODSIM.

**TELL**

Example: **TELL Carl TO StartMoving;**

Description: Invokes a **TELL** method of an object. A **TELL** method is invoked asynchronously and may contain **WAIT** statements. The **TELL** statement will not wait for the invoked method to be completed.

Example:

```
TYPE
    CarObj = OBJECT
        TELL METHOD StartMoving;
        ...
    END METHOD;
```

Description: Part of the method heading for a **TELL** method.

**TERMINATE**

Example: **TELL METHOD StartMoving;**

```
...
BEGIN
    ...
    IF (Location = WallLocat)
        OUTPUT("Crash.");
        TERMINATE;
    END IF;
    ...
END METHOD;
```

Description: A **TERMINATE** statement is used *from inside* a object instance's **TELL** or **WAITFOR** method to prematurely stop execution of that method and any method which invoked that method by means of a **WAIT FOR** statement.

**THISMETHOD**

Example: **TELL METHOD foo;**

```
VAR
    a : ACTID;
BEGIN
    a := THISMETHOD;
    . . .
```

Description: Built-in constant of type **ACTID** which represents a **TELL** or **WAITFOR** method activity. It is available within **TELL** and **WAITFOR** methods.

**TO**

Example: **ASK Obj1 TO Remove;**  
**TELL Obj1 TO Activate;**  
**WAIT FOR Carl TO Move;**

Description: **TO** is an optional “noise word” provided to make **ASK**, **TELL** and **WAIT** calls more readable.

**TRUE**

Example: **VAR**  
           **b : BOOLEAN;**  
           **...**  
           **b := TRUE;**

Description: One of the two **BOOLEAN** constants, the other being **FALSE**.

**TYPE**

Example: **TYPE**  
           **weekdays = ( Mon, Tue, Wed, Thur, Fri);**

Description: Precedes a series of type declarations.

**UNTIL**

Example: **REPEAT**  
           **INC(k);**  
           **OUTPUT(k);**  
           **UNTIL k = 5;**

Description: Identifies the terminating condition of a repeat loop.

**VAR**

Example: **VAR**  
           **Len: INTEGER;**

Description: Precedes a series of variable declarations.

**WAIT**

**WAIT FOR Carl TO GoTo(Garage)**  
**WAIT DURATION 5.0**  
**WAIT FOR Signal TO Trigger**

Description: A **WAIT** statement will suspend execution of the routine while simulation time elapses.

**WAITFOR**

Example:     **OBJECT**  
                  **ElevatorObj = OBJECT**  
                  . . .  
                  **WAITFOR METHOD Active(IN floor : INTEGER;**  
                  **OUT full : BOOLEAN)**  
                  . . .  
                  **END OBJECT;**

Description: A method which may only be called from a **WAIT FOR** statement. Unlike the **TELL** method, it may have **OUT** and **INOUT** parameters. In addition, the invoking method will not proceed until the method being waited for returns.

**WHEN**

Example:     **CASE NewCar**  
                  **WHEN Saab, Chrysler:**  
                  **OUTPUT("Family car");**  
                  **WHEN Porsche:**  
                  **OUTPUT("Sports car");**  
                  **OTHERWISE**  
                  **OUTPUT("A what?");**  
                  **END CASE;**

Description: Identifies a case in a **CASE** statement.

**WHILE**

Example:     **WHILE k < 5**  
                  **OUTPUT(k);**  
                  **INC(k);**  
                  **END WHILE;**

Description: Repeats the enclosed code while the **BOOLEAN** expression remains **TRUE**. The **BOOLEAN** expression is evaluated before each iteration.

**WITH**

Description: See **PRINT** and **SPRINT** in Appendix C.



## Appendix C. Built-in Procedures

---

For each of the procedures and functions listed below we have provided a procedure heading which describes the number and type of parameters and the type of the return value. Since these are built-in procedures, some may have special capabilities not available to user-defined procedures. For instance, the first procedure described can take either an **INTEGER** or **REAL** argument.

**ABS (IN arg : INTEGER or REAL) : INTEGER or REAL**

Description: Returns the absolute value of the argument. Return value is of same type as input.

**ACTIVATE(IN monvar : AnyMonitoredVar;  
          IN montype : AnyMonitoredObjectType)  
          : <montype> object reference**

Description: Activates a previously deactivated monitored variable.

**ADDMONITOR(IN monvar : AnyMonitoredVar;  
            IN monobj : AnyMonitorObjectVar)**

Description: Adds a monitor created by the user (dynamically) to the monitor list associated with the **monvar**.

**CAPIN ch : CHAR) : CHAR**

Description: Converts the input character to uppercase.

**CHARTOSTRIN chrArray : ARRAY OF CHAR) : STRING**

Description: Returns the **STRING** representation of an **ARRAY OF CHAR**.

**CHR(IN n : INTEGER) : CHAR**

Description: Converts an **INTEGER** in the range 0 to 255, inclusive, to the corresponding **CHAR**. For example, given an input of 65 it will return 'A'. If **n** falls outside the range 0 to 255, this routine returns **CHR(0)**.

**CLONE(IN d : anyDynType) : anyDynType**

Description: Takes any dynamic data type as an input and returns a copy. The dynamic data types are: **ARRAY**, **RECORD** and **OBJECT**.

If the input type is an object, its **ObjClone** method is invoked, if it exists.

```

DEACTIVATE(IN monvar : AnyMonitoredVar;
           IN montype : AnyMonitoredObjectType)
           : <montype> object reference

```

Description: Deactivates a monitored variable.

```

DEC(INOUT arg : AnyOrdinalType [; IN n : INTEGER])

```

Description: Decrements **arg** by **n**. i.e. **arg := arg - n**. If **n** is not specified, it defaults to **1**.

```

DISPOSE(IN refVar : AnyRefType)

```

Description: Deallocates the space pointed to by the argument. The argument can be of any dynamic data type; e.g. **ARRAY**, **RECORD** or **OBJECT**. The **refVar** is guaranteed to be initialized to **NILGBT**, **NILARRAY** or **NILREC** after the call.

If an object type is the input, **DISPOSE** executes the object's **ObjTerminate** method, if any, before deallocating the space used by the specified object instance.

```

FLOAT(IN n : INTEGER) : REAL

```

Description: Converts the argument to **REAL**.

```

GETMONITOR(IN monvar : AnyMonitoredVar;
           IN montype : AnyMonitoredObjectType)
           : <montype> object reference

```

Description: Returns the object type of a monitored object.

```

HALT

```

Description: Terminates a MODSIM program, returning control to the operating system or other calling program. **UtilMod** also defines a routine called **ExitToOS** which performs a halt while returning a status code to the operating system. In both cases any **PROCEDURES** registered with **ONEXIT** will be called before the program terminates.

```

HIGH(IN arr : AnyArrayType) : IndexType

```

Description: Returns the highest index of the array argument. The return type is the same as the index type used to define the array. The argument can also be an array element when **AnyArrayType** is a multi-dimension array.

```

INC(INOUT arg : AnyOrdinalType [IN n : INTEGER])

```

Description: Increments the given variable by the given amount, i.e. **arg := arg + n**. If **n** is not specified, it defaults to **1**.

**INPUT(OUT var1 : Sometype [ OUT var2 : Sometype ...] )**

Description: Reads from standard input and inserts the acquired values in each of the variables, sequentially. The input values may be separated by spaces, tabs, or newlines. Takes one or more parameters. **Sometype** must be one of: **CHAR**, **INTEGER**, **REAL**, or **STRING**. The types can be mixed.

**INSERT(INOUT str1 : STRING;  
      IN pos : INTEGER;  
      IN str2 : STRING)**

Description: Inserts **str2** at position **pos** in **str1**. If **pos** is less than or equal to zero, then **str2** is inserted ahead of **str1**. If **pos** is greater than the length of **str1**, then **str2** is inserted behind **str1**. If **str1** is null, **str2** is assigned to **str1**. If **str2** is null, **str1** is unchanged.

**INTTOSTR(IN n : INTEGER) : STRING**

Description: Returns the **STRING** representation of **n**.

**ISANCESTOR(IN objtype: AnyObjTypeIdentifier;  
          IN obj: AnyObjInstance) : BOOLEAN**

Description: Allows you to determine at runtime whether an object variable has a certain object type in its inheritance tree or is the object type itself.

**LOW(IN arr : AnyArrayType) : IndexType**

Description: Returns the lowest index of the array argument. The return type is the same as the index type used to define the array. The argument can also be an array element when **AnyArrayType** is a multi-dimension array.

**LOWER(IN str : STRING) : STRING**

Description: Returns a copy of **str** in which all upper-case characters have been changed to lower-case.

**MAX(ScalarType) : ScalarType**

Description: Returns the maximum value of the given type which can be represented by the computer. **MAX** may be used in constant expressions.

**MAXOF(IN arg1 : ScalarType  
      [IN arg2 : ScalarType ...]) : ScalarType**

Description: Returns the highest value from the list of scalar type arguments. All of the arguments in the list must be of the same scalar type.

**MIN(ScalarType) : ScalarType**

Description: Returns the minimum value of the given type which can be represented by the computer. **MIN** may be used in constant expressions.

**MINOF(IN arg1 : ScalarType  
[ IN arg2 : ScalarType ...]) : ScalarType**

Description: Returns the lowest value from the list of scalar type arguments. All of the arguments in the list must be of the same scalar type.

**NEW(OUT rec : AnyRecordType)**

Description: Allocates a new instance of a record and returns a reference to it.

**NEW(OUT obj : AnyObjectType)**

Description: Allocates a new instance of an object and returns its reference value. The object instance's **ObjInit** method is invoked automatically, if it exists.

**NEW(OUT array : AnyArrayType ;  
IN low..high : IndexType [ ; low..high : IndexType ...] )**

Description: Allocates memory for an array. Note that, for a multi-dimensional array, index ranges may be specified in separate **NEW** statements, and the **array** in that case would be specified by the already-defined indices.

**OBJTYPEID(IN objtype : AnyObjTypeIdIdentifier) : INTEGER**

Description: Given an object class type name, it returns a unique **INTEGER** valued identifier for that type.

**OBJTYPENAME(IN obj : ANYOBJ) : STRING**

Description: Given an object reference variable it returns a string which contains the object's type name. Note that the name which is returned is the original type name of the object, not the new name assigned if the object type was renamed in an **IMPORT** statement.

**OBJVARID(IN obj: ANYOBJ) : INTEGER**

Description: Given an object reference, returns a unique integer valued identifier for the object type.

**ODD(IN n : INTEGER) : BOOLEAN**

Description: Returns **TRUE** if the number is odd, **FALSE** if even.

**ONERROR(IN proc: AnyProcedure)**

Description: Registers **proc** with the system so that upon encountering a system error, **proc** will be invoked. Multiple procedures may be registered. They will be invoked in last-in-first-out order.

**ONEXIT(IN proc: AnyProcedure)**

Description: Same as **ONERROR** except **proc** will be invoked upon any exit from the program. Multiple procedures may be registered. They will be invoked in last-in-first-out order.

**ORD(IN arg : AnyOrdinalType) : INTEGER**

Description: Returns the ordinal value of the argument. For instance, the character 'A' has the ordinal value 65. If we define the enumeration: (**Mon, Tue, Wed, Thur, Fri**), then **ORD(Thur)** would return 3.

**OUTPUT([IN arg1 : Sometype] [ IN arg2 : Sometype ...] )**

Description: Writes the arguments to the standard output. A newline character is written after the last argument. If no arguments are given, a blank line is output. Takes zero or more arguments. **Sometype** must be one of: **CHAR, INTEGER, REAL, STRING**, or Object reference. The types can be mixed. If an object type is given as an argument, **OUTPUT** executes the object's **ObjPrint** method. If an **ObjPrint** method is not declared, the hexadecimal address of the object will be printed.

**POSITION(IN str1, str2 : STRING) : INTEGER**

Description: Returns the position of **str2** in **str1**. If **str2** is not completely contained in **str1**, returns 0. If either **str1** or **str2** is of length zero, a runtime error occurs.

**PRINT ([expressionlist]) WITH formatstring**

Description: Formats a string based on **formatstring** and outputs the constructed string to standard output with an appended new line.

**REALTOSTR(IN x : REAL) : STRING**

Description: Returns the **STRING** representation of **x**.

**REMOVEMONITOR(IN monvar : AnyMonitoredVar;  
                  IN monibj : AnyMonitoredObjectVar)**

Description: Removes a monitor created by the user (dynamically) from the monitor list associated with **monvar**.

**REPLACE(INOUT str1 : STRING;  
          IN      pos1,  
                  pos2 : INTEGER;  
          IN      str2 : STRING)**

Description: Replaces the part of **str1** from **pos1** to **pos2** with **str2**. e.g. if **str1 := "abcdefghijkl"** and **str2 := "WXYZ"**.

```
REPLACE(str1, 3, 4, str2) ⇒ str1 = "abWXYZefghijkl"
REPLACE(str1, 1, 1, str2) ⇒ str1 = "WXYZbcdefghijkl"
REPLACE(str1, 2, 11, str2) ⇒ str1 = "aWXYZl"
```

If **pos1** is 0, a runtime error occurs. If **pos1** is in **str1** but **pos2** is outside, the end of **str1** from **pos1** is replaced with **str2**. If both **pos1** and **pos2** are outside of **str1**, **str2** is concatenated to the end of **str1**.

**ROUND(IN arg : REAL) : INTEGER**

Description: Rounds the argument and returns the closest integer result. This is the algorithm:

```
IF arg >= 0.0
  RETURN( TRUNC( arg + 0.5 ));
ELSE
  RETURN( TRUNC( arg - 0.5 ));
END IF;
```

**SCHAR(IN str : STRING;  
IN pos : INTEGER) : CHAR**

Description: Returns the character at position **pos** in **str**. A run-time error occurs if **pos** falls outside of **str**.

**SIZEOF(AnyTypeName) : INTEGER;**

Description: Given any type specifier, this function returns the amount of memory space, in bytes, required to store a variable of that type. For example, **SIZEOF(INTEGER)** would return 4. **SIZEOF(PlayerRecType)** would return the number of bytes required to store a record of that type.

**SPRINT([expressionlist]) WITH formatstring**

Description: Returns a string constructed using **formatstring**. **formatstring** may be a literal, constant or variable **STRING**. Formatting is indicated by embedding asterisks to indicate field width and real number precision. Left justification, right justification, and centering are indicated by placing a <, >, and ~, as the last character of the field specification.

**STRLEN(IN str : STRING) : INTEGER**

Description: Returns the length of string **str**.

**STRTOCHAR(IN str : STRING;  
OUT chrArray : ARRAY INTEGER OF CHAR)**

Description: Converts **str** to an **ARRAY INTEGER OF CHAR**.

**STRTOINT(IN str : STRING) : INTEGER**

Description: Returns the **INTEGER** representation of **str**. If successful, sets the system defined variable **STRERR** to **FALSE**. If **str** cannot be converted, returns 0 and sets **STRERR** to **TRUE**.

**STRTOREAL(IN str : STRING) : REAL**

Description: Returns the **REAL** representation of **str**. If successful, sets the system defined variable **STRERR** to **FALSE**. If **str** cannot be converted, returns **0.0** and sets **STRERR** to **TRUE**.

**SUBSTR(IN pos1,  
pos2 : INTEGER;  
IN str : STRING) : STRING**

Description: Returns substring of **str** from **pos1** to **pos2**, inclusive. If the range lies outside of **str**, returns a null string. If **pos1** is in **str** and **pos2** falls outside, returns from **pos1** to end of string. If **pos1** is less than or equal to zero or **pos1** is greater than **pos2**, a run-time error occurs.

**TRACE**

Description: Output a series of messages (to standard output) which indicate the current call stack.

**TRUNC(IN arg : REAL) : INTEGER**

Description: Truncates **arg** to an integer.

**UPDATEVALUE(IN value : MonitorType)**

Description: Called from **LMONITOR** methods to modify the **NEWVALUE**.

**UPPER(IN str : STRING) : STRING**

Description: Returns a copy of **str** in which all lower-case characters have been changed to upper-case.

**VAL(IN OrdinalTypeName : OrdinalType;  
IN OrdNum : INTEGER) : OrdinalType**

Description: Returns a value, of the specified type, which has the given ordinal position. For instance, **VAL(CHAR, 65)** will return **'A'**.



## **Appendix D. Standard Library Modules**

---

This appendix contains alphabetical listings of all of the constants, variables, types, procedures and objects defined by the standard library modules.

## D.1 Module Name: Debug

### D.1.1 Description

Provides functionality to help debug MODSIM programs.

### D.1.2 Variables

None.

### D.1.3 Types

None.

### D.1.4 Procedures

#### **GetNumberArrays**

Parameters: None

Return Value: **INTEGER**

Description: Returns the number of currently allocated arrays.

#### **GetNumberStrings**

Parameters: None

Return Value: **INTEGER**

Description: Returns the number of currently allocated strings.

#### **GetNumberType**

Parameters: **IN typeid: INTEGER**

Return Value: **INTEGER**

Description: Returns the number of currently active objects/records with the id 'typeid.' The id of an object instance may be obtained through **OBJVARID**.

#### **ObjectDump**

Parameters: **IN object : ANYOBJ**

Return Value: None

Description: Prints general information regarding the object reference to **stdout**.

#### **PrintMemStats**

Parameters: **IN stream : StreamObj**

Return Value: None

Description: Prints out a formatted record containing the number of currently allocated arrays, strings, objects and records. For objects and records it will print out the name of the object/record type and the number allocated only if that number is greater than zero.

**WriteTrace**

Parameters: **IN filename : STRING**

Return Value: None

Description: In the case of a runtime error or a call to **TRACE** in a program, the result of the traceback will be printed to a file (in the current working directory) named '**filename**' - the default, if no call to **WriteTrace** is made, is to write the result to **stderr**. Only those procedures and methods compiled with the traceback option turned on will be included in the trace list.

## D.2 Module Name: GrpMod

### D.2.1 Description

Provides functionality to represent and iterate over groups of objects.

### D.2.2 Constants

None.

### D.2.3 Types

#### GroupOrderType

Type: enumeration  
 Constants: vFIFO {first in, first out}  
 vLIFO {last in, first out}  
 vRanked {ranked, - override Rank method}  
 Description: Determines the add behavior of groups.

#### StatINTEGER

Type: LRMONITORED INTEGER BY IStatObj, ITimedStatObj  
 Description: Provides a definition for a monitored integer type that will gather statistics with and without respect to time.

#### D.2.3.1 Object Types

The following is a list of objects which are documented in Appendix E:

```
QueueObj
StackObj
RankedObj
BTreeObj
StatQueueObj
StatStackObj
StatRankedObj
StatBTreeObj
SimQueueObj
```

Virtual objects - intermediate objects which are not used directly:

```
GroupObj
ExpandedGroupObj
BasicGroupObj
ExpandedBasicGroupObj
BasicRankedObj
BasicBTreeObj
StatGroupObj
BStatGroupObj
```

## D.2.4 Procedures

### **GetGroups**

Parameters:     **IN obj : ANYOBJ**  
                  **INOUT groups : QueueObj**

Return Value:   None

Description:    Generates a queue containing pointers to each group of which the specified (given) object is a member.

## D.3 Module Name: IOMod

### D.3.1 Description

Provides I/O interface functionality.

### D.3.2 Constants

None

### D.3.3 Types

#### **FileUseType**

Type: enumeration  
 Constants: **Input Output InOut Append Update CreateBinary**  
 Description: Used by the **Open** method of **StreamObj** to determine the type of file opening required. **Input**, **Output**, **InOut**, **Append** are text mode opens. **Update** and **CreateBinary** are binary mode opens.

### D.3.4 Procedures

#### **DeleteFile**

Parameters: **IN fname : STRING**  
 Return Value: None  
 Description: Removes the file **fname** from this disk storage device, if it exists. No error results from attempting to delete a non-existent file.

#### **FileAccessTime**

Parameters: **IN fname: STRING**  
 Return Value: **INTEGER**  
 Description: Returns time file last accessed in seconds past 1/1/70 00:00 GMT .

#### **FileExists**

Parameters: **IN fname: STRING**  
 Return Value: **BOOLEAN**  
 Description: Determines whether the file named **fname** (or directory) exists on the disk storage device. **fname** may be a full path specification for a file.

#### **FileModTime**

Parameters: **IN fname: STRING**  
 Return Value: **INTEGER**  
 Description: Returns time file last modified in seconds past 1/1/70 00:00 GMT.

#### **FileSize**

Parameters: **IN fname: STRING**  
Return Value: **INTEGER**  
Description: Returns the size of the file in bytes, or -1 if the file does not exist.

**ReadKey**

Parameters: None  
Return Value: **CHAR**  
Description: **ReadKey** reads one character from the console with no echo. It does NOT require a Carriage Return, Enter or Newline before returning with the character.

## **D.4 Module Name: ListMod**

### **D.4.1 Description**

Provides functionality to represent and iterate over collections of records.

### **D.4.2 Constants**

None

### **D.4.3 Types**

None

#### **D.4.3.1 Object Types**

The following is a list of objects which are documented in Appendix E:

```
QueueList
StackList
RankedList
BTreeList
StatQueueList
StatStackList
StatRankedList
StatBTreeList
```

Virtual objects - intermediate objects which are not to be used directly:

```
ListObj
BasicListObj
BasicQueueList
BasicStackList
BasicRankedList
BasicBTreeList
StatListObj
BStatListObj
BStatQueueList
BStatStackList
BStatRankedList
BStatBTreeList
```

**D.5 Module Name: MathMod****D.5.1 Description**

General purpose math procedures.

**D.5.2 Constants**

**pi**

Type: **REAL**

Value: 3.1415926535897932;

**e**

Type: **REAL**

Value: 2.7182818284590452;

**D.5.3 Types**

None.

**D.5.4 Procedures**

**ACOS**

Parameters: **IN x : REAL**

Return Value: **REAL**

Description: Arc cosine of x,  $-1 \leq x \leq 1$

**ASIN**

Parameters: **IN x : REAL**

Return Value: **REAL**

Description: Arc sine of x,  $-1 \leq x \leq 1$

**ATAN**

Parameters: **IN x : REAL**

Return Value: **REAL**

Description: Arc tangent of x from  $-\pi/2$  to  $\pi/2$

**ATAN2**

Parameters: **IN y : REAL**

**IN x : REAL**

Return Value: **REAL**

Description: Two argument (cartesian) form of this operation.

**CEIL**

Parameters: **IN x : REAL**

Return Value: **INTEGER**

Description: Returns the smallest integer not less than x.

**COS**

Parameters: **IN x : REAL**  
 Return Value: **REAL**  
 Description: Cosine of x

**EXP**

Parameters: **IN x : REAL**  
 Return Value: **REAL**  
 Description: Exponential function:  $e^x$

**FLOOR**

Parameters: **IN x: REAL**  
 Return Value: **INTEGER**  
 Description: Returns the largest integer not greater than x

**LN**

Parameters: **IN x: REAL**  
 Return Value: **REAL**  
 Description: Natural log of x,  $0 < x$

**LOG10**

Parameters: **IN x: REAL**  
 Return Value: **REAL**  
 Description: Base 10 log of x ,  $0 < x$

**POWER**

Parameters: **IN x: REAL**  
**IN y: REAL**  
 Return Value: **REAL**  
 Description: X raised to the y power

**SIN**

Parameters: **IN x: REAL**  
 Return Value: **REAL**  
 Description: Sine of x

**SQRT**

Parameters: **IN x: REAL**  
 Return Value: **REAL**  
 Description: Square root of x,  $0 \leq x$

**TAN**Parameters: **IN x: REAL**Return Value: **REAL**Description: Tangent of  $x$ ,  $x \in ]-\pi/2, \pi/2[$ .

## D.6 Module Name: OSMod

### D.6.1 Description

A portable operating system interface. This module contains procedure definitions that can be used to access various functions of the operating system. The procedures defined in **OSMod** are intended to provide an OS interface to write portable applications but not necessarily to exploit all features of the underlying operating system.

### D.6.2 Constants

#### OSOK, OSERROR

Type: **INTEGER**

Description: Returns values for calls that return 'success' (**ok**) or 'error' as their return value.

### D.6.3 Types

#### AccessTypeET

Type: enumeration

Constants: **ATRead** { file allows read access }  
**ATWrite** { file allows write access }  
**ATReadWrite** { both read and write ok }  
**ATExecute** { file is marked executable }  
**ATFileOK** { file exists and is accessible }

Description: Return type of **TestAccess** procedure.

#### DIRHNDL

Type: **ANYREC**

Description: Directory handle. Used for reading directories.

#### FileTypeET

Type: enumeration

Constants: **FTUnknown** { file type unrecognized or file doesn't exist }  
**FTDirectory** { file is directory }  
**FTOrdinary** { ordinary file }  
**FTCharSpecial** { Char special (Unix) }  
**FTBlkSpecial** { Block special (Unix) }  
**FTFIFO** { FIFO/ pipe }

Description: Return type of **FileType** procedure.

#### OSTimeRec

Type: **FIXED RECORD**

Fields: **sec** : **INTEGER** { seconds (0 - 59) }  
**min** : **INTEGER** { minutes (0 - 59) }  
**hour** : **INTEGER** { hours (0 - 23) }  
**mday** : **INTEGER** { day of month (1 - 31) }  
**mon** : **INTEGER** { month of year (1 - 12) }

```

year: INTEGER { year - 1900 }
wday: INTEGER { day of week (Sunday = 0) }
yday: INTEGER { day of year (0 - 365) }

```

Description: Time record structure to hold the components of the system time. Set by `LocalTime()`, read by `TimeRec2Asc()`.

## D.6.4 Procedures

### D.6.4.1 Miscellaneous OS Queries and Calls

#### `ClearScreen`

Parameters: None  
 Return Value: None  
 Description: Clear the text screen.

#### `ExitToOS`

Parameters: **IN** status: **INTEGER**  
 Return Value: None  
 Description: Exit to OS and return status as the return code.

**Note :** This routine exits immediately. It does not call MODSIM `ONEXIT` routines and does not call the MODSIM debugger. For an alternative see `ExitTOOS` in `UtilMod`.

#### `GetComputerType`

Parameters: None  
 Return Value: **STRING**  
 Description: Returns the type of the computer. See notes on `GetOSType()` below.

#### `GetCurrentDrive`

Parameters: None  
 Return Value: **STRING**  
 Description: Returns the current drive. For PCs (Windows 95, NT) this is the drive letter (with trailing colon). For UNIX systems an empty string is returned.

#### `GetEnv`

Parameters: **IN var**: **STRING**  
 Return Value: **STRING**  
 Description: Gets the value of environment variable `var`. Returns an empty string when `var` was undefined or the empty string. When the variable `var` was undefined the procedure `OSErrorCode` will return `OSERROR` (otherwise `OSOK`) so that 'empty' can be distinguished from 'undefined'.

#### `GetHostID`

Parameters: None  
 Return Value: **INTEGER**  
 Description: Returns a hardware specific serial number of the host. If none is available, '0' is returned.

**GetHostName**

Parameters: None  
 Return Value: **STRING**  
 Description: Returns the name of the host (if defined) on the network. When "hostname" is not supported on the OS, an empty string is returned.

**GetOSType**

Parameters:  
 Return Value: **STRING**  
 Description: Returns the OS type of the current OS. The return string contains multiple words that describe the OS, e.g. the company, the OS name, the version, common aliases, etc. NEVER DO AN EXPLICIT STRING COMPARE with the result this function, since the contents may change in the future as additional information becomes necessary. Instead use the built-in procedure **POSITION** to match the word you are interested in. YOU ARE GUARANTEED that components that are in the string WILL NEVER BE TAKEN OUT. Therefore, existing code will not break when additional information is added.

- All flavors of UNIX will contain the word **UNIX**
- All flavors of Windows will contain the word **WINDOWS**

**GetProgDir**

Parameters: **IN prog: STRING**  
 Return Value: **STRING**  
 Description: When **prog** is a path (i.e. contains directory separation characters), **GetProgDir** returns just the directory part of it. If it is just a name, **GetProgDir** searches through the current 'path' and returns the first directory that contains that file/executable or an empty string when not found.

**GetWindowSysType**

Parameters: None  
 Return Value: **STRING**  
 Description: Returns the type of the current window system. See **GetOSType()** above.

**GetWorkingDirectory**

Parameters: None  
 Return Value: **STRING**  
 Description: Get current working directory as string (no trailing slash). If there was an error, the empty string is returned.

**MaxFileNameLength**

Parameters: None  
 Return Value: **INTEGER**  
 Description: Returns the maximum length of a file name (i.e. a path component) on the current OS for the given drive. This includes only the name and not any mandatory extension part.

**OSErrorCode**

Parameters: None.  
 Return Value: **INTEGER**  
 Description: Error code of the last failed OS call.

**D.6.4.2 Time Routines****Delay**

Parameters: **IN seconds: INTEGER**  
 Return Value: None  
 Description: Delays '**seconds**' and then resumes execution.

**LocalTime**

Parameters: **IN timesecs: INTEGER**  
 Return Value: **OSTimeRec**  
 Description: Converts the given number of time seconds into minutes, hours, etc. and fills **OSTimeRec** with the results. Note that **OSTimeRec** is a local (static) variable in **LocalTime** so that each call overwrites that single time structure.

**MicroDelay**

Parameters: **IN microseconds: INTEGER**  
 Return Value: None  
 Description: Delays '**microseconds**' and then resumes execution.

**SystemRealTime**

Parameters: None  
 Return Value: **REAL**  
 Description: Returns number of seconds since a fixed time point in the past. Resolution is fractions of a second, if supported by operating system.

**SystemTime**

Parameters: None  
 Return Value: **INTEGER**  
 Description: Returns number of seconds since a fixed time point in the past.

**TimeRecToAsc**

Parameters: **IN ostimerec: OSTimeRec**  
 Return Value: **STRING**

Description: Transforms time data from `aOSTimeRec` record into character form:  
"Sat Apr 13 15:45:30 1991\0"

**TimeToString**

Parameters: IN `timesecs`: **INTEGER**

Return Value: **STRING**

Description: Returns ASCII representation of given time (seconds) in string form as described in `TimeRecToAsc`.

**D.6.4.3 Files and Directories****AppendSlash**

Parameters: IN `path`: **STRING**

Return Value: **STRING**

Description: Appends a directory separator character to '`path`' if the last character of '`path`' is not already a directory separator character.

**BaseName**

Parameters: IN `path`: **STRING**

Return Value: **STRING**

Description: Returns only the last name part of '`path`'. Possible trailing slashes in `path` are ignored, i.e. for "`/a/b/`", "`b`" is returned.

**ChangeDir**

Parameters: IN `path`: **STRING**

Return Value: **INTEGER**

Description: Change directory to `path`. Returns `OSOK` for ok, `OSERROR` on error.

**CloseDir**

Parameters: IN `dirhndl`: **DIRHNDL**

Return Value: **INTEGER**

Description: Close directory associated with `dirhndl`. Returns `OSOK` for ok, `OSERROR` for error.

**CopyFile**

Parameters: IN `from`, `to`: **STRING**

Return Value: **INTEGER**

Description: Copies file '`from`' to '`to`' and returns `OSOK` on ok, `OSERROR` in case of an error.

**DeleteFile**

Parameters: IN `path`: **STRING**

Return Value: **INTEGER**

Description: Remove file name. Returns `OSOK` for ok, `OSERROR` on error.

**DirName**

Parameters: IN `path`: **STRING**

Return Value: **STRING**

Description: Returns only the directory part (without the last directory separator character) of `path`. Possible trailing slashes in path are ignored.

#### **FileAccessTime**

Parameters: IN path: **STRING**

Return Value: **INTEGER**

Description: Time of last file access in seconds since `'system time starting point'` (see `'SystemTime'`), or 0 for **ERROR** (e.g. file does not exist).

#### **FileExists**

Parameters: IN path: **STRING**

Return Value: **BOOLEAN**

Description: Returns **TRUE** when file name exists **FALSE** otherwise.

#### **FileModTime**

Parameters: IN path: **STRING**

Return Value: **INTEGER**

Description: Time of last file modification in seconds since `'system time starting point'` (see `'SystemTime'`), or 0 for **ERROR** (e.g. file does not exist).

#### **FileSize**

Parameters: IN path: **STRING**

Return Value: **INTEGER**

Description: Returns file size in bytes for path or -1 if error.

#### **FileType**

Parameters: IN path: **STRING**

Return Value: **FileTypeET**

Description: Get file type code for path.

#### **GetDirSepChar**

Parameters: None

Return Value: **CHAR**

Description: Returns the `'directory separator character'` for the current OS (e.g. `'/'` on UNIX)

#### **IncrFileNameIfExists**

Parameters: IN filename: **STRING**

Return Value: **STRING**

Description: Makes a unique filename from `filename` by incrementing a `'counter'` at the end of the file name. The format of filename is: `"<optional path><name><idx>.<ext>"` where `"<idx>"` is a number (sequence of digits) or empty and the `"."` is the `'extension separator char'` for this OS. When `'filename'` does not exist, it is returned unchanged. When it does exist, the number in `idx` is incremented until the resulting file name does not exist (gets unique file

name). When 'infilename' is not found, 'idx' starts at 1. When 'idx' grows in width and the total length of the filename exceeds **MaxFileNameLength**, ending characters of 'infilename' (<name> part) are removed.

#### **IsLegalFileNameChar**

Parameters: **IN c: CHAR**

Return Value: **BOOLEAN**

Description: Return **TRUE** if the given char can be used in the NAME PART of a filename on the current OS.

#### **MakeDir**

Parameters: **IN directory: STRING**

Return Value: **INTEGER**

Description: Make directory. Returns **OSOK** for ok, **OSERROR** on error.

#### **MakeEmptyFile**

Parameters: **IN filename: STRING**

Return Value: **INTEGER**

Description: Makes an empty file with the given path. If the file already exists, it is overwritten. Returns **OSOK** on ok, **OSERROR** on error.

#### **MakeTmpFile**

Parameters: **IN directory: STRING**

Return Value: **STRING**

Description: Creates a unique tmp file in 'directory' and returns the name. When 'directory' is an empty string, the current directory is used. The file is created immediately (with **MakeEmptyFile**) so that subsequent calls to it will not return the same name.

#### **MapToLegalFileName**

Parameters: **IN namestr: STRING**

Return Value: **STRING**

Description: Maps an arbitrary string **namestr** to a legal file name, deleting all characters from **namestr** that are illegal for a file name on the current operating system.

#### **MatchesFilePattern**

Parameters: **IN str, pat: STRING**

Return Value: **BOOLEAN**

Description: Returns **TRUE** when string **str** matches pattern **pat**. Intended for file name pattern matching. Pattern recognizes wild cards "\*" and "?" with their usual meaning. Combinations of the wild card characters are allowed. The wild cards work as follows:

- Let **ANYCHAR** be one arbitrary character. The match of **pat** against **str** is done in a **FORWARD** comparison.

- A '?' in **pat** matches exactly one **ANYCHAR** in **str** (at the current comparison position).
- A "\*" in **pat** matches a sequence (0 or more) of **ANYCHARS** in **str** EXCEPT the one character that follows the '\*' in **pat**. For instance, **pat** = **"\*."** matches **"abc."** where '\*' matches **"abc"**. But **"\*."** does not match **"abc.d."**, since the first dot in **str** will match the first (and only) dot in **pat**.

**Note:** This is not a regular expression match.

**Note:** This is a file name comparison. On Windows, the match is not case sensitive.

#### **MatchesFilePatternNoCase**

Parameters: **IN str, pat: STRING**

Return Value: **BOOLEAN**

Description: Case insensitive version of **MatchesFilePattern**'.

#### **OpenDir**

Parameters: **IN path: STRING**

Return Value: **DIRHNDL**

Description: Call to start reading the directory '**path**'. Returns a handle to the opened directory that must be passed in to **NextDirEntry** and **CloseDir**. When the return **DIRHNDL** is **NILREC**, the directory could not be opened. When '**path**' is an empty string the current directory is read. Use '**FileType**' and '**MatchesFilePattern**' to filter out only certain entries when reading a directory.

#### **NextDirEntry**

Parameters: **IN dirhndl: DIRHNDL**

Return Value: **STRING**

Description: Returns next directory entry in directory opened with **OpenDir** or an empty string for end or error. When **end**, **OSErrorCode** is 0, otherwise **OSErrorCode** <> 0 (OS dependent error code). See also: '**MatchesFilePattern**'.

#### **RemoveDir**

Parameters: **IN directory: STRING**

Return Value: **INTEGER**

Description: Remove directory. Returns **OSOK** for ok, **OSERROR** for error when the directory was not empty.

#### **RenameFile**

Parameters: **IN old, new: STRING**

Return Value: **INTEGER**

Description: Rename file **old** to **new** (UNIX: **mv**). Might not work on all OSs to rename directories. Returns **OSOK** or **OSERROR**.

#### **RemoveFinalSlash**

Parameters: **IN path: STRING**  
 Return Value: **STRING**  
 Description: Removes all directory separator characters from the end of **path**.

**TestAccess**

Parameters: **IN path: STRING; IN acc: AccessTypeET**  
 Return Value: **BOOLEAN**  
 Description: Test file with name **path** for access rights by current user. The type checked for is given in **acc**. Returns **TRUE** for access, **FALSE** other

**D.6.4.4 Process Management****CheckBGTask**

Parameters: **OUT pid, exitcode, status: INTEGER**  
 Return Value: None.  
 Description: This function can be used to query the operating system as to whether a background task has finished yet, whether it terminated normally or was killed by a signal and what exit code was returned (e.g. from an **ExitToOS()** call). The output parameters are set as follows:

- **pid**: the process id of the terminated process, or -1 for 'no children present' or 0 for 'no background task has finished yet'. When **pid** is neither -1 or 0, i.e. it contains a valid **PID** of a background process that has terminated, the remaining out parameters are set as follows:
- **exitcode**: the exit/return code of the terminated process
- **status**: 0 when child terminated normally (with **Exit()** call); a value  $\neq$  0 indicates abnormal termination (e.g. killed, not enough memory, etc.) in which case the value's interpretation is dependent on the OS.

This call NEVER WAITS for a background task to finish. It always returns immediately. To check for multiple finished background tasks, it should be called in a loop as long as the return value is greater than 0.

**GetPID**

Parameters: None.  
 Return Value: **INTEGER**  
 Description: Gets **PID** of current process or -1 for error. Returns 0 when processes are not supported on the current operating system.

**GetPPID**

Return Value: **INTEGER**  
 Description: Gets **PPID** (parent **PID**) of current process or -1 for error.

**KillBGTask**

Parameters: **IN pid: INTEGER**

Return Value: **INTEGER**  
 Description: Sends a **'kill'** signal to the background task **PID** (previously started with **startBackgroundTask**. Note that some programs can choose to ignore the **kill** signal on some operating systems. Returns **OSOK** for ok, **OSERROR** for error (e.g. wrong PID).

**startBGTask**

Parameters: **IN cmd: STRING; IN minimized: INTEGER**  
 Return Value: **INTEGER**  
 Description: Issues **'cmd'** as a background task and returns the **PID** (process ID/ a process handle) of the newly started task. As in **systemCall**, **'cmd'** is handed to the systems command line interpreter/shell. This call is asynchronous, i.e. it returns immediately after starting the background job. To find out when the background job has finished and what exit code was returned by that task, you must call **'HasBGTaskFinished'** described below. When **'minimized' <> 0**, the application runs minimized (an icon but no window is shown).

**systemCall**

Parameters: **IN cmd: STRING; IN minimized: INTEGER**  
 Return Value: **INTEGER**  
 Description: Performs OS call with **'cmd'** and returns the return code. **'cmd'** is a command string that will be handed to a command line interpreter. This call is synchronous, i.e. it waits until the command returns. When **'minimized' <> 0**, the application runs minimized (an icon but no window is shown).

## D.7 Module Name: RandMod

### D.7.1 Description

Provides random number generation capability.

### D.7.2 Constants

None

### D.7.3 Types

#### D.7.3.1 Object Types

The following is a list of objects which are documented in Appendix E:

`RandomObj`

### D.7.4 Procedures

#### **FetchSeed**

Parameters: `IN SimscriptSeedNumber : INTEGER`

Return Value: `INTEGER`

Description: **FetchSeed** will return the first seed of the specified SIMSCRIPT random number stream. For example,

```
ASK RandStream TO SetSeed(FetchSeed(4)).
```

#### **Random**

Parameters: None

Return Value: `REAL`

Description: Pseudo-random number generator. Returns a sample between 0.0 and 1.0 excluding the end points.

## **D.8 Module Name: ResMod**

### **D.8.1 Description**

Contains descriptions of objects used to track resources used by an application.

### **D.8.2 Constants**

None

### **D.8.3 Types**

None

#### **D.8.3.1 Object Types**

The following is a list of objects which are documented in Appendix E:

- EntryObj**
- PriorityList**
- AllocQueueObj**
- ResourceObj**

## D.9 Module Name: SimMod

### D.9.1 Description

Provides interface to simulation functionality.

### D.9.2 Variables

#### Timescale

Type: **REAL**

Description: Number of real seconds per simulation unit (graphics only).

### D.9.3 Types

#### D.9.3.1 Object Types

The following is a list of objects which are documented in Appendix E:

**ActivityGroup**  
**SimControlObj**  
**TriggerObj**

### D.9.4 Procedures

#### ActivityListDump

Parameters: **IN ProcObj : ANYOBJ**

Return Value: **None**

Description: Dumps the activity list of a particular object.

#### ActivityName

Parameters: **IN activity : ACTID**

Return Value: **STRING**

Description: Given an activity record for a TELL METHOD, returns the name of the method.

#### ActivityOwner

Parameters: **IN activity : ACTID**

Return Value: **ANYOBJ**

Description: Given an activity record for a TELL METHOD, returns the owner object for this method instance.

#### Interrupt

Parameters: **IN object: ANYOBJ**

**IN methName: STRING**

Return Value: **None**

Description: Causes the method named **methName** of the object instance **object** to receive an interrupt message when it returns from its wait. No error

occurs if no active method of this name is found for the indicated object instance. If the method has been scheduled, but has not yet executed it will simply be removed from the pending list.

#### **InterruptAll**

Parameters: **IN object: ANYOBJ**  
 Return Value: None  
 Description: Interrupts all activities scheduled for that object.

#### **InterruptMethod**

Parameters: **IN activity: ACTID**  
 Return Value: None  
 Description: Sends an interrupt message to the specific method instance described by '**activity**.' The activity reference should have been captured when doing the original scheduling of the method. If the method has already completed or the '**activity**' argument is **NILREC**, a runtime error will result.

#### **InterruptWaitingFor**

Parameters: **IN activity: ANYREC**  
 Return Value: None  
 Description: Interrupts the method(s) that are suspended and waiting '**activity**'. The '**activity**' must be a method that was activated by a WAIT FOR statement. This method will set the state of the invoking method to interrupted and cause '**activity**' to be removed from any pending lists.

#### **NumActivities**

Parameters: **IN object: ANYOBJ**  
 Return Value: **INTEGER**  
 Description: Returns number of activities pending for an object.

#### **NumActPending**

Parameters: None  
 Return Value: **INTEGER**  
 Description: Returns the total number of activities pending.

#### **NumObjPending**

Parameters: None  
 Return Value: **INTEGER**  
 Description: Returns the number of objects with activities pending.

**NumWAITFOR**

Parameters: None  
 Return Value: **INTEGER**  
 Description: Returns the number of activities in a **WAIT..FOR** status.

**PendingListDump**

Parameters: **IN DoActList: BOOLEAN**  
 Return Value: None  
 Description: Dumps the entire pending list showing each object on it. If **DoActList** is true, it also dumps each object's activity list.

**PendingListDumpToStream**

Parameters: **IN doActList : BOOLEAN; IN stream : streamObj**  
 Return Value: None  
 Description: Prints a summary of the contents of the simulation pending list to the named file '**stream**'. If '**doActList**' is 'TRUE', it also dumps each object's activity list.

**ResetSimTime**

Parameters: **IN newtime: REAL**  
 Return Value: None  
 Description: Resets simulation time to newtime for multiple runs - may only be invoked before **StartSimulation** begins or after **StartSimulation** ends.

**ScheduledTime**

Parameters: **IN activity : ACTID**  
 Return Value: **REAL**  
 Description: Given an activity record for a TELL METHOD, returns the next time the activity is scheduled to execute.

**SimTime**

Parameters: None  
 Return Value: **REAL**  
 Description: Returns the current simulation time.

**StartSimulation**

Parameters: None  
 Return Value: None  
 Description: This procedure begins the simulation run. No simulation methods will be initiated (actually executed) until **StartSimulation** is begun. At least one method must be scheduled prior to calling **StartSimulation** or control will immediately return to the state-

ment following the `StartSimulation`. Control returns to this statement after all pending method activities have completed.

**StopSimulation**

Parameters: None

Return Value: None

Description: Will empty the pending list of all activities and return control to the statement after `StartSimulation`.

**UseCalendar**

Parameters: `IN flag : BOOLEAN`

Return Value: None

Description: Determines which of two algorithms will be used to order the simulation pending list. The Calendar Queue algorithm is the default data structure for ordering the simulation pending list. It works best for models which may have 10 or more activities concurrently pending. For certain models the older ranking algorithm may prove more optimal. To obtain the older method pass 'FALSE' as the argument. To restore usage of the Calendar Queue pass 'TRUE'.

## D.10 Module Name: StatMod

### D.10.1 Description

Contains definitions for statistical accumulation objects. Also included are predefined statistical types for basic **INTEGER** and **REAL** types.

### D.10.2 Constants

None.

### D.10.3 Types

#### **BINTEGER**

Type: **LMONITORED INTEGER BY IStatObj, ITimedStatObj;**  
 Description: Predefined **INTEGER** type for collectiong statistics.

#### **BREAL**

Type: **LMONITORED REAL BY RStatObj, RTimedStatObj;**  
 Description: Predefined **REAL** type for collecting statistics.

#### **histogram**

Type: **ARRAY INTEGER OF REAL**  
 Description: Provides the type definition used by statistical objects for histograms. Users may declare variables of this type and assign the Histogram field of statistical objects to it, thereby getting access to the elements of the histogram.

#### **SINTEGER**

Type: **LMONITORED INTEGER BY IStatObj;**  
 Description: Predefined **INTEGER** type for collecting statistics.

#### **SREAL**

Type: **LMONITORED REAL BY RStatObj;**  
 Description: Predefined **REAL** type for collecting statistics.

#### **TSINTEGER**

Type: **LMONITORED INTEGER BY ITimedStatObj;**  
 Description: Predefined **INTEGER** type for collecting statistics.

#### **TSREAL**

Type: **LMONITORED REAL BY RTimedStatObj;**  
 Description: Predefined **REAL** type for collecting statistics.

### D.10.3.1 Object Types

The following is a list of objects which are documented in Appendix E:

```
IStatObj  
ITimedStatObj  
RStatObj  
RTimedStatObj  
StatObj  
TimedStatObj
```

## D.11 Module Name: UtilMod

### D.11.1 Description

Provides general purpose utility procedures.

### D.11.2 Constants

None.

### D.11.3 Types

#### MachineType

Type: **enumeration**

Constants: UnknownComp, Sun3, SPARC, Sun386, PC, VAXstation, DECstation, SGI, MacII, Tektronix, DG88000, R6000, NeXT, HP300, HP700, HP800, PCUnix, Motorola, DECALPHA

Description: Constants returned by **GetComputerType** function.

#### OSType

Type: **enumeration**

Constants: UnknownOS, DOS, OS286, OS2, SunOS, VMS, Ultrix, Irix, UTek, DGUnix, HPUnix, AIX, Macintosh, MACH, SCOUNIX, Unix88, Windows, Solaris, AlphaOSF

Description: Constants returned by **GetOSType** function.

### D.11.4 Procedures

#### AdrToHex

Parameters: **IN Adr : ANYREC**  
**OUT Hex : STRING**

Return Value: None

Description: Converts a dynamic record reference into a **STRING** containing the hex representation of the record's address.

#### CallDebugger

Parameters: None

Return Value: **BOOLEAN**

Description: Invokes the MOSIM debugger, returns TRUE if the debugger was invoked.

#### ClockTimeSecs

Parameters: None

Return Value: **INTEGER**

Description: Time in seconds since 1/1/70 .

**ClockRealSecs**

Parameters: None  
 Return Value: **REAL**  
 Description: Time in seconds since 1/1/70 - resolution is fractions of seconds if the OS provides support.

**DateTime**

Parameters: **OUT time : STRING**  
 Return Value: None  
 Description: Provides date and time in following format: Tue Aug 02 17:38:32 1988 .

**ExitToOS**

Parameters: **IN status : INTEGER**  
 Return Value: None  
 Description: Halts execution, passing exit **status** to OS.

**GetCmdLineArg**

Parameters: **IN ArgNumber: INTEGER;**  
**OUT Arg: STRING**  
 Return Value: None  
 Description: Returns command line arguments. Returns a null string if **ArgNumber** is higher than the actual arguments. Returns program name if given **ArgNumber=0**.

**GetComputerType**

Parameters: None  
 Return Value: **MachineType**  
 Description: Returns a constant of type '**MachineType**' which indicates the machine on which the program is running.

**GetNumArgs**

Parameters: None  
 Return Value: **INTEGER**  
 Description: Returns the number of command line arguments.

**GetOSType**

Parameters: None  
 Return Value: **OSType**  
 Description: Returns a constant of type '**OSType**' indicating the type of operating system under which the program is executing.

**RuntimeError**

Parameters: **IN message: STRING**

Return Value: None

Description: Invokes MODSIM run-time error mechanism passing it the message.

## D.12 Module Name: Version

### D.12.1 Description

Provides access to the version number of the MODSIM compiler.

### D.12.2 Constants

None

### D.12.3 Types

None

### D.12.4 Procedures

#### **getVersion**

Parameters: None

Return Value: **STRING**

Description: Returns the version number of the MODSIM compiler, e.g. '1.2'.

#### **getVersionDate**

Parameters: None

Return Vaalue: **STRING**

Description: Returns the build date of the MODSIM compiler, e.g. '10/18/96'.

#### **ObtainVersion**

Parameters: **OUT version : STRING**

Return Value: None

Description: Gives the version of the MODSIM compiler, e.g. 'MODSIM-III Version 1.2'.



## **Appendix E: Objects**

---

# ActivityGroup

Module: **SimMod**

Derived From: **QueueList**

Substitutes: **ANYREC:ACTID**

Description: A specialized group for collecting activities.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>ActivityGroup</b>
<b>Next</b>	None	No	<b>ActivityGroup</b>

## FIELDS and METHODS

ASK Method: **Add**

Parameters: **IN item : ACTID;**

Return Value: None

Description: Adds an activity to the end of the group.

ASK Method: **Next**

Return Value: None

Parameters: **IN member : ACTID;**

Description: Returns next member.

Module: **SysMod**  
Derived From: **RankedList**  
Description: A MODSIM internal.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>nextTime</b>	<b>REAL</b>	Yes	<b>ActivityList</b>
<b>NextActivity</b>	<b>ActivityList</b>	Yes	<b>ActivityList</b>
<b>PrevActivity</b>	<b>ActivityList</b>	Yes	<b>ActivityList</b>

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>SetNext</b>	None	Yes	<b>ActivityList</b>
<b>SetPrev</b>	None	Yes	<b>ActivityList</b>
<b>Add</b>	None	Yes	<b>ActivityList</b>
<b>AddFirst</b>	None	Yes	<b>ActivityList</b>
<b>Remove</b>	<b>activitytype</b>	Yes	<b>ActivityList</b>
<b>RemoveThis</b>	None	Yes	<b>ActivityList</b>

#### FIELDS and METHODS

Internal to operation of the MODSIM system.

## ActivityQueue

Module: **SimMod**  
Derived From: **QueueList**  
**activitytype**  
Description: A MODSIM internal.

FIELDS and METHODS: None

Module: **ResMod**

Derived From: **StatQueueObj**

Substitutes: **EntryObj** for: **ANYOBJ**

Description: A list of **EntryObj**'s containing references to objects that have acquired one or more resources.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>numResources</b>	<b>INTEGER</b>	No	<b>StatGroupObj</b>

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Adjust</b>	None	No	<b>AllocQueueObj</b>
<b>Reset</b>	None	No	<b>AllocQueueObj</b>

#### FIELDS and METHODS

Field: **numResources**

Type: **INTEGER**

Description: Field provided for statistics.

ASK Method: **Adjust**

Parameters: **IN delta : INTEGER;**

Return Value: None

Description: Updates **numResources** field for statistics.

ASK Method: **Reset**

Parameters: None

Return Value: None

Description: Reset monitors associated with **numResources**.

# BasicBTreeList

Module: **ListMod**

Derived From: **ListObj**

Description: A “virtual” object that provides the basic methods required for a btree ordered group. A btree is an efficient data structure for storing ordered sets of data that will have many members.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Order</b>	<b>INTEGER</b>	No	<b>BasicBTreeList</b>
<b>ASK Method</b>			
<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>	
<b>Add</b>	None	No	<b>BasicBTreeList</b>
<b>Find</b>	<b>#ANYREC</b>	No	<b>BasicBTreeList</b>
<b>Key</b>	<b>STRING</b>	No	<b>BasicBTreeList</b>
<b>ObjInit</b>	None	No	<b>BasicBTreeList</b>
<b>SetOrder</b>	None	No	<b>BasicBTreeList</b>

## FIELDS and METHODS

Field: **Order**

Type: **INTEGER**

Description: Contains the maximum number of values that may be stored in a node of the tree. The desired order of a tree depends upon the nature of the objects to be stored. The default order of the btree is 5.

ASK Method: **Add**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Determines the correct insertion point for the passed in record based upon its key value (returned by the **Key** method) and adds the record to the group at that location in the tree. If more than one object may have the same key, the new record will be added after all other equivalently keyed records.

ASK Method: **Find**

Parameters: **IN key: STRING**

Return Value: **#ANYREC**

Description: This method will return the first record it finds that is associated with the passed in string. If no record is found in the group with such a key **NILREC** is returned. If more than one record may have the same key the **Next** method will provide references to them.

ASK Method: **Key**

Parameters: **IN object: #ANYREC**

Return Value: **STRING**

Description: This method will be used by the insertion and location methods of the **btree** to determine the correct location for this object. The user is expected to provide (by overriding) a method appropriate to the records being added to the group.

ASK Method: **ObjInit**

Parameters: **None**

Return Value: **None**

Description: Sets the default order of the **btree** to 5.

ASK Method: **SetOrder**

Parameters: **IN degree: INTEGER**

Return Value: **None**

Description: Allows the order of the **btree** to be changed. The **btree** must be empty when this is done otherwise a runtime error will be generated. In general an odd order works best.

# BasicBTreeObj

Module: **GrpMod**

Derived From: **BasicGroupObj**

Description: A “virtual” object that provides the basic methods required for a btree ordered group based upon a user provided sting key. A btree is an efficient data structure for storing ordered sets of data that will have many members.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Order</b>	<b>INTEGER</b>	No	<b>BasicBTreeObj</b>
<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Find</b>	<b>#ANYOBJ</b>	No	<b>BasicBTreeObj</b>
<b>Key</b>	<b>STRING</b>	No	<b>BasicBTreeObj</b>
<b>ObjInit</b>	None	No	<b>BasicBTreeObj</b>
<b>ObjLoad</b>	None	No	<b>BasicBTreeObj</b>
<b>SetOrder</b>	None	No	<b>BasicBTreeObj</b>

## FIELDS and METHODS

Field: **Order**

Type: **INTEGER**

Description: Contains the maximum number of values that may be stored in a node of the tree. The desired order of a tree depends upon the nature of the objects to be stored.

ASK Method: **ObjInit**

Parameters: None

Return Value: None

Description: Sets the default order of the btree to 5.

ASK Method: **ObjLoad**

Parameters: None

Return Value: None

Description: Internal.

ASK Method: **Key**

Parameters: **IN object: #ANYOBJ**

Return Value: **STRING**

Description: This method will be used by the insertion and location methods of the btree to determine the correct location for this object. The user is expected to provide (by overriding) a method appropriate to the objects being added to the group.

ASK Method: **Find**

Parameters: **IN key: STRING**

Return Value: **#ANYOBJ**

Description: This method will return the first object it finds that is associated with the passed key in string. If no object is found in the group with such a key **NILOBJ** is returned. If more than one object may have the same key the **Next** method will provide references to them.

ASK Method: **SetOrder**

Parameters: **IN degree: INTEGER**

Return Value: None

Description: Allows the order of the btree to be changed. The btree must be empty when this is done, otherwise a runtime error will be generated. In general an odd order works best.

# BasicGroupObj

Module: **GrpMod**

Derived From: **GroupObj**

Description: A “virtual” object that describes the core methods that all group objects have.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	<b>None</b>	No	<b>BasicGroupObj</b>
<b>AddAfter</b>	<b>None</b>	No	<b>BasicGroupObj</b>
<b>AddBefore</b>	<b>None</b>	No	<b>BasicGroupObj</b>
<b>Dump</b>	<b>None</b>	No	<b>BasicGroupObj</b>
<b>Empty</b>	<b>None</b>	No	<b>BasicGroupObj</b>
<b>EmptyAndDispose</b>	<b>None</b>	No	<b>BasicGroupObj</b>
<b>First</b>	<b>#ANYOBJ</b>	No	<b>BasicGroupObj</b>
<b>Includes</b>	<b>BOOLEAN</b>	No	<b>BasicGroupObj</b>
<b>Last</b>	<b>#ANYOBJ</b>	No	<b>BasicGroupObj</b>
<b>Next</b>	<b>#ANYOBJ</b>	No	<b>BasicGroupObj</b>
<b>ObjClone</b>		No	<b>BasicGroupObj</b>
<b>ObjTerminate</b>	<b>None</b>	No	<b>BasicGroupObj</b>
<b>Prev</b>	<b>#ANYOBJ</b>	No	<b>BasicGroupObj</b>
<b>Rank</b>	<b>#ANYOBJ</b>	No	<b>BasicGroupObj</b>
<b>Remove</b>	<b>#ANYOBJ</b>	No	<b>BasicGroupObj</b>
<b>RemoveThis</b>	<b>None</b>	No	<b>BasicGroupObj</b>

## FIELDS and METHODS

ASK Method: **Add**

Parameters: **IN ExistingMember: #ANYOBJ**  
**IN NewMember: #ANYOBJ**

Return Value: None

Description: Adds an object (**NewMember**) to the end of the group.

ASK Method: **AddAfter**

Parameters: **IN ExistingMember: #ANYOBJ**  
**IN NewMember: #ANYOBJ**

Return Value: None

Description: Adds the an object (**NewMember**) to the group after an object that is already a member of the group.

ASK Method: **AddBefore**

Parameters: **IN ExistingMember: #ANYOBJ**  
**IN NewMember: #ANYOBJ**

Return Value: None

Description: Adds an object (**NewMember**) to the group before an object that is already a member of the group.

ASK Method: **Dump**

Parameters: None

Return Value: None

Description: Prints information about a group object including references (hex addresses) to its members to **stdout**.

ASK Method: **First**

Parameters: None

Return Value: **#ANYOBJ**

Description: Returns a reference to the first object in the group. If the group is empty it returns **NILOBJ**.

ASK Method: **Includes**

Parameters: **IN candidate: #ANYOBJ**

Return Value: **BOOLEAN**

Description: Determines membership of an object in the group. If it is a member **TRUE** is returned, otherwise **FALSE**.

ASK Method: **Last**

Parameters: None

Return Value: **#ANYOBJ**

Description: Returns a reference to the last object in the group. If the group is empty it returns **NILOBJ**.

ASK Method: **Next**

Parameters: **IN candidate: #ANYOBJ**

Description: A reference to an object that is a member of the group.

Return Value: **#ANYOBJ**

Description: Returns a reference to the object that immediately follows the passed in object within the group. If the passed in object is the last member of the group **NILOBJ** is returned.

ASK Method: **ObjClone**

Parameters: None

Return Value: **#ANYOBJ**

Description: Per MODSIM language.

ASK Method: **ObjTerminate**

Parameters: None

Return Value: None

Description: Checks that the group is empty before disposal. If not, a runtime error is generated. Groups must contain no members when they are disposed.

## BasicGroupObj (cont.)

ASK Method: **Prev**

Parameters: **IN candidate: #ANYOBJ**

Description: A reference to an object that is a member of the group.

Return Value: **#ANYOBJ**

Description: Returns a reference to the object that immediately precedes the passed in object within the group. If the passed in object is the first member of the group **NILOBJ** is returned.

ASK Method: **Remove**

Parameters: None

Return Value: **#ANYOBJ**

Description: Returns a reference to the first object in a group after removing that object from the group.

ASK Method: **RemoveThis**

Parameters: **IN member: #ANYOBJ**

Description: A reference to an object that is a member of the group.

Return Value: None

Description: Removes a specific object from the group.

Module: **ListMod**  
 Derived From: **ListObj**  
 Description: A “virtual” object that describes the core methods that all list objects have.

<u>ASKMethod</u>	<u>ReturnType</u>	<u>Private</u>	<u>DefinedBy</u>
<b>AddAfter</b>	None	No	<b>BasicListObj</b>
<b>AddBefore</b>	None	No	<b>BasicListObj</b>
<b>Dump</b>	None	No	<b>BasicListObj</b>
<b>First</b>	<b>#ANYREC</b>	No	<b>BasicListObj</b>
<b>Includes</b>	<b>BOOLEAN</b>	No	<b>BasicListObj</b>
<b>Last</b>	<b>#ANYREC</b>	No	<b>BasicListObj</b>
<b>Next</b>	<b>#ANYREC</b>	No	<b>BasicListObj</b>
<b>ObjTerminate</b>	None	No	<b>BasicListObj</b>
<b>Prev</b>	<b>#ANYREC</b>	No	<b>BasicListObj</b>
<b>Remove</b>	<b>#ANYREC</b>	No	<b>BasicListObj</b>
<b>RemoveThis</b>	None	No	<b>BasicListObj</b>

#### FIELDS and METHODS

ASK Method: **AddAfter**

Parameters: **IN ExistingMember: #ANYREC**  
**IN NewMember: #ANYREC**

Return Value: None

Description: Adds a record (**NewMember**) to the group after a record that is already a member of the group.

ASK Method: **AddBefore**

Parameters: **IN ExistingMember: #ANYREC**  
**IN NewMember: #ANYREC**

Return Value: None

Description: Adds a record (**NewMember**) to the group before a record that is already a member of the group.

ASK Method: **Dump**

Parameters: None

Return Value: None

Description: Prints information about a group object including references (hex addresses) to its members to **stdout**.

## BasicListObj (cont.)

ASK Method: **First**

Parameters: None

Return Value: **#ANYREC**

Description: Returns a reference to the first record in the group. If the group is empty it returns **NILREC**.

ASK Method: **Includes**

Parameters: **IN candidate: #ANYREC**

Return Value: **BOOLEAN**

Description: Determines membership of a record in the group. If it is a member **TRUE** is returned, otherwise **FALSE**.

ASK Method: **Last**

Parameters: None

Return Value: **#ANYREC**

Description: Returns a reference to the last record in the group. If the group is empty it returns **NILOBJ**.

ASK Method: **Next**

Parameters: **IN candidate: #ANYREC**

Return Value: **#ANYREC**

Description: A reference to a record that is a member of the group.

Description: Returns a reference to the record that immediately follows the passed in record within the group. If the passed in record is the last member of the group **NILOBJ** is returned.

ASK Method: **ObjTerminate**

Parameters: None

Return Value: None

Description: Checks that the group is empty before disposal. If not, a runtime error is generated. Groups must contain no members when they are disposed.

ASK Method: **Prev**

Parameters: **IN candidate: #ANYREC**

Description: A reference to a record that is a member of the group.

Return Value: **#ANYREC**

Description: Returns a reference to the record that immediately precedes the passed in record within the group. If the passed in record is the first member of the group **NILOBJ** is returned.

ASK Method: **Remove**

Parameters: None

Return Value: **#ANYREC**

Description: Returns a reference to the first record in a group after removing that record from the group.

ASK Method: **RemoveThis**

Parameters: **IN member: #ANYREC**

Description: A reference to an record that is a member of the group.

Return Value: None

Description: Removes a specific record from the group.

# BasicQueueList

Module: **ListMod**

Derived From: **ListObj**

Description: A “virtual” object for holding lists of records.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>BasicQueueList</b>

## FIELDS and METHODS

ASK Method: **Add**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Inserts the passed in record at the end of the list.

# BasicRankedList

Module: **ListMod**

Derived From: **ListObj**

Description: A “virtual” object for holding sorted lists of records.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>BasicRankedList</b>
<b>AddFirst</b>	None	No	<b>BasicRankedList</b>
<b>Rank</b>	<b>INTEGER</b>	No	<b>BasicRankedList</b>

## FIELDS and METHODS

ASK Method: **Add**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Adds an object to the group by determining its rank relative to members already in the group. New member records having a rank equal to objects already members of the group will be inserted after all such records.

ASK Method: **AddFirst**

Parameters: **IN: #ANYREC**

Return Value: None

Description: Inserts an record at the head of the group regardless of its rank. Caution should be used in invoking this method as it can disturb the ranked nature of the group.

ASK Method: **Rank**

Parameters: **IN object1: #ANYREC**

**IN object2: #ANYREC**

Return Value: **INTEGER**

Description: This method is provided as a “stub” so that the user may derive their own group object from a **RankedList** (see below) and override the Rank method to specify the desired ordering of the records. The return values should be as follows:

```
record1 < record2 => -1
record1 = record2 => 0
record1 > record2 => 1
```

# BasicRankedObj

Module: **GrpMod**

Derived From: **GroupObj**

Description: A “virtual” object that allows ranked groups to be initially built quickly without ranking.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>AddFirst</b>	None	No	<b>BasicRankedObj</b>

## FIELDS and METHODS

ASK Method: **AddFirst**

Parameters: IN **NewMember**: #ANYOBJ

Return Value: None

Description: Adds before first.

Module: **ListMod**

Derived From: **BasicBTreeList**  
**StatListObj**

Description: Same functionality as **BasicBTreeList** plus accumulates statistics on number of records in list.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>DefinedBy</u>
<b>Order</b>	<b>INTEGER</b>	No	<b>BasicBTreeList</b>
<b>firstRoster</b>	<b>ANYREC</b>	Yes	<b>ListObj</b>
<b>lastRoster</b>	<b>ANYREC</b>	Yes	<b>ListObj</b>
<b>number</b>	<b>StatINTEGER</b>	No	<b>StatListObj</b>
<b>numberIn</b>	<b>INTEGER</b>	No	<b>ListObj</b>
<b>root</b>	<b>ANYOBJ</b>	Yes	<b>BasicBTreeList</b>

<u>ASKMethod</u>	<u>ReturnType</u>	<u>Private</u>	<u>DefinedBy</u>
<b>Add</b>	None	No	<b>BStatBTreeList</b>
<b>Find</b>	<b>#ANYREC</b>	No	<b>BasicBTreeObj</b>
<b>Key</b>	<b>STRING</b>	No	<b>BasicBTreeObj</b>
<b>ObjInit</b>	None	No	<b>BasicBTreeObj</b>
<b>SetOrder</b>	None	No	<b>BasicBTreeObj</b>

#### FIELDS and METHODS

ASK Method: **Add**

Return Value: None

Parameters:

**IN NewMember: #ANYREC**

Description: See **Add** method of **BasicBTreeObj**.

## BasicStackList

Module: **ListMod**

Derived From: **ListObj**

Description: A virtual object for holding stacks of records.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>BasicStackList</b>

### FIELDS and METHODS

ASK Method: **Add**

Return Value: None

Parameters: IN **NewMember**: #ANYREC

Description: Add record first in list.

Module: **ListMod**

Derived From: **BasicBTreeList**  
**StatListObj**

Description: Provides statistics gathering of BTree records.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>BStatBTreeList</b>

#### FIELDS and METHODS

ASK Method: **Add**

Parameters: IN **NewMember**: **#ANYREC**

Return Value: None

Description: Add record to BTree and update level.

# BStatGroupObj

Module: **GrpMod**

Derived From: **ExpandedBasicGroupObj**  
**StatGroupObj**

Description: A “virtual” object that provides all the basic functionality of group objects plus the basic methods required for statistical accumulations on groups.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>AddAfter</b>	None	No	<b>BStatGroupObj</b>
<b>AddBefore</b>	None	No	<b>BStatGroupObj</b>
<b>Count</b>	<b>INTEGER</b>	No	<b>BStatGroupObj</b>
<b>Maximum</b>	<b>INTEGER</b>	No	<b>BStatGroupObj</b>
<b>Mean</b>	<b>REAL</b>	No	<b>BStatGroupObj</b>
<b>Minimum</b>	<b>INTEGER</b>	No	<b>BStatGroupObj</b>
<b>Remove</b>	<b>#ANYOBJ</b>	No	<b>BStatGroupObj</b>
<b>RemoveThis</b>	None	No	<b>BStatGroupObj</b>
<b>Reset</b>	None	No	<b>BStatGroupObj</b>
<b>SetHistogram</b>	None	No	<b>BStatGroupObj</b>
<b>StdDev</b>	<b>REAL</b>	No	<b>BStatGroupObj</b>
<b>Variance</b>	<b>REAL</b>	No	<b>BStatGroupObj</b>
<b>WtdMean</b>	<b>REAL</b>	No	<b>BStatGroupObj</b>
<b>WtdStdDev</b>	<b>REAL</b>	No	<b>BStatGroupObj</b>
<b>WtdVariance</b>	<b>REAL</b>	No	<b>BStatGroupObj</b>

## FIELDS and METHODS

ASK Method: **AddBefore**

Parameters: **IN ExistingMember: #ANYOBJ**  
**IN NewMember: #ANYOBJ**

Return Value: None

Description: Inserts the '**NewMember**' immediately before the '**ExistingMember**' within the group.

ASK Method: **AddAfter**

Parameters: **IN ExistingMember: #ANYOBJ**  
**IN NewMember: #ANYOBJ**

Return Value: None

Description: Inserts the '**NewMember**' immediately after the '**ExistingMember**' within the group.

ASK Method: **Count**

Parameters: None

Return Value: **INTEGER**

Description: Returns the number of times the **numberIn** field has been modified.

ASK Method: **Maximum**

Parameters: None

Return Value: **INTEGER**

Description: Returns the maximum value that has been in the **numberIn** field (i.e., the maximum number of objects ever in the group).

ASK Method: **Mean**

Parameters: None

Return Value: **REAL**

Description: The average number of objects in the group.

ASK Method: **Minimum**

Parameters: None

Return Value: **INTEGER**

Description: Returns the minimum value that has been in the **numberIn** field (i.e., the minimum number of objects ever in the group - always 0).

ASK Method: **Remove**

Parameters: None

Return Value: **#ANYOBJ**

Description: Returns a reference to the first object in a group after removing that object from the group.

ASK Method: **RemoveThis**

Parameters: **IN member**

Type: **#ANYOBJ**

Description: A reference to an object that is a member of the group.

Return Value: None

Description: Removes a specific object from the group.

ASK Method: **Reset**

Parameters: None

Return Value: None

Description: Clears the statistical accumulations to zero. Useful for multiple repetitions of a simulation.

ASK Method: **SetHistogram**

Parameters: **IN low: INTEGER**

**IN high: INTEGER**

**IN interval: INTEGER**

Return Value: None

Description: Allows user to set up a histogram for accumulation on the **numberIn** field of the group.

## BStatGroupObj (cont.)

ASK METHOD: **StdDev**

Parameters: None

Return Value: **REAL**

Description: Returns the standard deviation for the mean of the **numberIn**.

ASK METHOD: **Variance**

Parameters: None

Return Value: **REAL**

Description: Returns the variance for the mean of the **numberIn**.

ASK METHOD: **wtdMean**

Parameters: None

Return Value: **REAL**

Description: Returns the average number of objects in the group, weighted with respect to time.

ASK METHOD: **wtdStdDev**

Parameters: None

Return Value: **REAL**

Description: Returns the standard deviation of the average number of objects in the group, weighted with respect to time.

ASK METHOD: **wtdVariance**

Parameters: None

Return Value: **REAL**

Description: Returns the variance of the average number of objects in the group, weighted with respect to time.

Module: **ListMod**

Derived From: **BasicListObj**  
**StatListObj**

Description: A “virtual” object that provides all the basic functionality of list objects plus the basic methods required for statistical accumulations on lists.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>AddAfter</b>	None	No	<b>BStatListObj</b>
<b>AddBefore</b>	None	No	<b>BStatListObj</b>
<b>Count</b>	<b>INTEGER</b>	No	<b>BStatListObj</b>
<b>Remove</b>	<b>#ANYREC</b>	No	<b>BStatListObj</b>
<b>RemoveThis</b>	None	No	<b>BStatListObj</b>
<b>Reset</b>	None	No	<b>BStatListObj</b>
<b>SetHistogram</b>	None	No	<b>BStatListObj</b>
<b>StdDev</b>	<b>REAL</b>	No	<b>BStatListObj</b>
<b>Variance</b>	<b>REAL</b>	No	<b>BStatListObj</b>
<b>WtdMean</b>	<b>REAL</b>	No	<b>BStatListObj</b>
<b>WtdStdDev</b>	<b>REAL</b>	No	<b>BStatListObj</b>
<b>WtdVariance</b>	<b>REAL</b>	No	<b>BStatListObj</b>

**FIELDS and METHODS**

ASK Method: **AddAfter**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Add a record after an existing record and update level.

ASK Method: **AddBefore**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Add a record before an existing record and update level.

ASK Method: **Count**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Number of observations since last reset.

ASK Method: **Maximum**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Maximum level observed since last reset.

ASK Method: **Mean**

Parameters: **IN NewMember: #ANYREC**

## BStatListObj (cont)

Return Value: None

Description: Mean observation since last reset.

ASK Method: **Minimum**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Minimum level observed since last reset.

ASK Method: **Remove**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: .

ASK Method: **RemoveThis**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Remove specific record from group and update level.

ASK Method: **Reset**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Reset statistics.

ASK Method: **SetHistogram**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Set histogram.

ASK Method: **StdDev**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Standard deviation since last reset.

ASK Method: **WtdMean**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Time-wighted mean since last reset.

ASK Method: **WtdVariance**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Time-wighted variance since last reset.

ASK Method: **Variance**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Variance since last reset.

# BStatQueueList

Module: **ListMod**

Derived From: **BasicQueueList**  
**StatListObj**

Description: Same functionality as a **BasicQueueList** plus accumulates statistics on number of records in list.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>BStatQueueList</b>

## FIELDS and METHODS

ASK Method: **Add**

Return Value: None

Parameters: **IN NewMember: #ANYREC**

Description: See **Add** method of **BasicQueueList**.

Module: **ListMod**

Derived From: **BasicStackObj**  
**StatListObj**

Description: This object has the same functionality as a **BasicStackList** object.  
The **BStatStackList** also accumulates statistics on a number of records  
in the list.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
Add	None	No	<b>BStatStackList</b>

#### FIELDS and METHODS

ASK Method: **Add**

Parameters: **IN NewMember: #ANYOBJ**

Return Value: None

Description: Inserts '**NewMember**' at the end of the group.

# BStatQueueList

Module: **ListMod**

Derived From: **BasicQueueList**  
**StatListObj**

Description: Provides statistics gathering using ranked groups of records.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>BStatQueueList</b>

## FIELDS and METHODS

ASK Method: **Add**

Parameters: **IN NewMember: #ANYREC**

Return Value: None

Description: Inserts '**NewMember**' at the end of the group.

Module: **ListMod**

Derived From: **BasicRankedList**  
**StatListObj**

Description: Same functionality as **BasicRankedList** plus accumulates statistics on number of records in list.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>BStatRankedList</b>
<b>AddFirst</b>	None	No	<b>BStatRankedList</b>

#### FIELDS and METHODS

ASK Method: **Add**

Return Value: None

Parameters: **IN NewMember: #ANYREC**

Description: See **Add** method of **BasicRankedList**.

ASK Method: **AddFirst**

Return Value: None

Parameters: **IN NewMember: #ANYREC**

Description: See **AddFirst** method of **BasicRankedList**.

## BStatStackList

Module: **ListMod**

Derived From: **BasicStackList**  
**StatListObj**

Description: Same functionality as **BasicStackList** plus accumulates statistics on number of records

in list.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>BStatStackList</b>

### FIELDS and METHODS

ASK Method: **Add**

Return Value: None

Parameters: IN **NewMember**: **ANYREC**

Description: See **Add** method of **BasicStackObj**.

Module: **ListMod**

Derived From: **BasicListObj**

**BasicBTreeList**

Description: A composite object that provides an ordered insertion of records into a group based upon a key value for each record added. If the Key method is not overridden insertion will be FIFO as in a **QueueList**. **Btrees** are an efficient data structure for ordered trees that will have many members and do a lot of insertion and deletion.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>AddAfter</b>	None	No	<b>BTreeList</b>
<b>AddBefore</b>	None	No	<b>BTreeList</b>
<b>ObjTerminate</b>	None	No	<b>BTreeList</b>
<b>Remove</b>	<b>#ANYREC</b>	No	<b>BTreeList</b>
<b>RemoveThis</b>	None	No	<b>BTreeList</b>

#### FIELDS and METHODS

ASK Method: **AddAfter**

Parameters: **IN ExistingMember : ANYOBJ;**  
**IN NewMember : ANYOBJ;**

Return Type: None

Description: Must not be overridden with a BTree.

ASK Method: **AddBefore**

Parameters: **IN ExistingMember : ANYOBJ;**  
**IN NewMember : ANYOBJ;**

Return Type: None

Description: Must not be overridden with a BTree.

ASK Method: **ObjTerminate**

Parameters: None

Return Type: None

Description: Cleanup internal BTree structures.

ASK Method: **Remove**

Parameters: None

Return Type: None

Description: Remove first record from group.

## BTreeList (cont)

ASK Method: **RemoveThis**

Parameters: **IN Member : ANYOBJ;**

Return Type: None

Description: Remove the specific record from the group.

Module: **GrpMod**

Derived From: **BasicBTreeObj**

Description: A composite object that provides an ordered insertion of objects into a group based upon a key value for each object added. If the Key method is not overridden insertion will be FIFO as in a **QueueObj**. BTrees are an efficient data structure for ordered trees that will have many members and do a lot of insertion and deletion.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>AddAfter</b>	None	No	<b>BasicBTreeObj</b>
<b>AddBefore</b>	None	No	<b>BasicBTreeObj</b>
<b>ObjTerminate</b>	None	No	<b>BasicBTreeObj</b>
<b>Remove</b>	<b>#ANYOBJ</b>	No	<b>BasicBTreeObj</b>
<b>RemoveThis</b>	None	No	<b>BasicBTreeObj</b>

#### FIELDS and METHODS

ASK Method: **AddAfter**

Parameters: **IN ExistingMember : ANYOBJ;**  
**IN NewMember : ANYOBJ;**

Return Type: None

Description: **AddAfter** cannot be used with a BTree.

ASK Method: **AddBefore**

Parameters: **IN ExistingMember : ANYOBJ;**  
**IN NewMember : ANYOBJ;**

Return Type: None

Description: **AddBefore** cannot be used with a BTree.

ASK Method: **ObjTerminate**

Parameters: None

Return Type: None

Description: Verification and overhead.

ASK Method: **Remove**

Parameters: None

Return Type: None

Description: Removes First, i.e., the element with the smallest key.

ASK Method: **RemoveThis**

Parameters: **IN Member :**

Return Type: None

Description: Removes the specific member.

# EntryObj

Module: **ResMod**

Derived From: None

Description: This object is a record of an object's use of a resource.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Object</b>	<b>ANYOBJ</b>	No	<b>EntryObj</b>
<b>Trigger</b>	<b>TriggerObj</b>	No	<b>EntryObj</b>
<b>Number</b>	<b>INTEGER</b>	No	<b>EntryObj</b>
<b>Priority</b>	<b>REAL</b>	No	<b>EntryObj</b>
<b>Timer</b>	<b>ANYOBJ</b>	No	<b>EntryObj</b>
<b>State</b>	<b>INTEGER</b>	No	<b>EntryObj</b>

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Initialize</b>	None	No	<b>EntryObj</b>
<b>SetNumberIn</b>	None	No	<b>EntryObj</b>
<b>SetPriorityTo</b>	None	No	<b>EntryObj</b>
<b>SetState</b>	None	No	<b>EntryObj</b>
<b>SetTimer</b>	None	No	<b>EntryObj</b>
<b>SetTriggerTo</b>	None	No	<b>EntryObj</b>

## FIELDS and METHODS

Field: **Object**

Type: **ANYOBJ**

Description: Object that own/requests the resource.

Field: **Trigger**

Type: **TriggerObj**

Description: Internal trigger.

Field: **Number**

Type: **INTEGER**

Description: Number of units of the resource.

Field: **Priority**

Type: **REAL**

Description: Priority of the request.

Field: **Timer**

Type: **ANYOBJ**

Description: Internal.

Field: **State**

Type: **INTEGER**

Description: Internal.

ASK Method: **Initialize**

Return Value: None

Parameters: **IN Obj: ANYOBJ**  
**IN num: INTEGER**  
**IN Trig: TriggerObj**  
**IN next: INTEGER**  
**IN pr: REAL**

Description: Initializes all fields of the object.

ASK Method: **SetNumberTo**

Return Value: None

Parameters: **IN number : INTEGER**

Description: Sets the **Number** field.

ASK Method: **SetPriorityTo**

Return Value: None

Parameters: **IN pr : REAL**

Description: Sets the **Priority** field.

ASK Method: **setState**

Return Value: None

Parameters: **IN state : INTEGER**

Description: Sets the internal state field.

ASK Method: **setTimer**

Return Value: None

Parameters: **IN timer : ANYOBJ**

Description: Sets the **Timer** field.

ASK Method: **setTriggerTo**

Return Value: None

Parameters: **IN Trig : TriggerObj**

Description: Sets the **Trigger** field.

# ExpandedBasicGroupObj

Module: **GrpMod**

Derived From: **BasicGroupObj**

**ExpandedBasicGroupObj**

Description: Adds support for FIFO, LIFO and ranked groups.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>SetGroupOrder</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>UpdateDelay</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>UpdateEntryLevel</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>UpdateLevel</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>UpdateExitLevel</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>UpdateNumEntries</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>UpdateNumExits</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>SetDelayStats</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>ResetStats</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>Add</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>GetRosterCard</b>	None	No	<b>ExpandedBasicGroupObj</b>
<b>DelRosterCard</b>	None	No	<b>ExpandedBasicGroupObj</b>

## FIELDS and METHODS

ASK Method: **Add**

Parameters: **IN NewMember : #ANYOBJ**

Type: **INTEGER**

Return Value: None

Description: Add a member to the group.

ASK Method: **DelRosterCard**

Parameters: **IN card : ANYREC;**

**IN member : #ANYOBJ**

Return Value: None

Description: Internal method.

ASK Method: **GetRosterCard**

Parameters: **IN NewMember : #ANYOBJ**

Return Value: None

Description: Internal method.

ASK Method: **ResetStats**

Parameters: None

Return Value: None

Description: Sets the **Timer** field.

ASK Method: **SetGroupOrder**

Parameters: **IN disc : GroupOrderType**

Return Value: None

Description: Set the **eGroupOrder** field.

ASK Method: **SetDelayStats**

Parameters: **IN flag : BOOLEAN**

Return Value: None

Description: Set the **bDelayStats** field.

ASK Method: **SetLevelStats**

Parameters: **IN flag : BOOLEAN**

Return Value: None

Description: Set the **bLevelStats** field.

ASK Method: **UpdateDelay**

Parameters: **IN obj : #ANYOBJ;**  
**IN dt : REAL**

Return Value: None

Description: Update the **Delay** field.

ASK Method: **UpdateLevel**

Parameters: **IN rDelta : REAL**

Return Value: None

Description: Set the **Level** field.

ASK Method: **UpdateExitLevel**

Parameters: **IN obj : #ANYOBJ**

Return Value: None

Description: Update the **Level** field on exit from the group.

ASK Method: **UpdateNumEntries**

Parameters: **IN obj : #ANYOBJ**

Return Value: None

Description: Update the number of entries.

ASK Method: **UpdateNumExits**

Parameters: **IN obj : #ANYOBJ**

Return Value: None

Description: Set the number of exits.

# ExpandedGroupObj

Module: **GrpMod**

Derived From: **GroupObj**

Description: Adds fields used in the **ExpandedBasicGroupObj**.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>bDelayStats</b>	<b>BOOLEAN</b>	No	<b>ExpandedGroupObj</b>
<b>bLevelStats</b>	<b>BOOLEAN</b>	No	<b>ExpandedGroupObj</b>
<b>eGroupOrder</b>	<b>STRING</b>	No	<b>ExpandedGroupObj</b>
<b>rLevel</b>	<b>INTEGER</b>	No	<b>ExpandedGroupObj</b>
<b>rNumEntries</b>	<b>ANYREC</b>	No	<b>ExpandedGroupObj</b>
<b>rNumExits</b>			<b>ExpandedGroupObj</b>
<b>tDelay</b>			<b>ExpandedGroupObj</b>

## FIELDS and METHODS

Field: **bDelayStats**  
Type: **BOOLEAN**  
Description: If **TRUE**, collect delay stats.

Field: **bLevelStats**  
Type: **BOOLEAN**  
Description: If **TRUE**, collect level stats.

Field: **eGroupOrder**  
Type: **GroupOrder**  
Description: Determines add behavior: FIFO, LIFO, Ranked.

Field: **rLevel**  
Type: **LMONITORED REAL**  
Description: Membership level for monitoring.

Field: **rNumEntries**  
Type: **REAL**  
Description: Number of times member is added to group.

Field: **rNumExits**  
Type: **REAL**  
Description: Number of times member is removed from group.

Field: **tDelay**  
Type: **LMONITORED REAL**  
Description: Accumulates **SIMTIME** objects that are members of the group for the purpose of monitoring.

Module: **GrpMod**

Derived From: None

Description: A “virtual” object that describes the fields all group objects have.

Field	Type	Private	DefinedBy
<b>firstRoster</b>	<b>ANYREC</b>	Yes	<b>GroupObj</b>
<b>lastRoster</b>	<b>ANYREC</b>	Yes	<b>GroupObj</b>
<b>numberIn</b>	<b>INTEGER</b>	No	<b>GroupObj</b>

ASK Method	Return Type	Private	Defined By
<b>GetRosterCard</b>	None	No	<b>GroupObj</b>
<b>DelRosterCard</b>	None	No	<b>GroupObj</b>

## FIELDS and METHODS

Field: **firstRoster**

Type: **ANYREC**

Description: An internal (to the group object) record that contains a reference to the first member of the group.

Field: **lastRoster**

Type: **ANYREC**

Description: An internal (to the group object) record that contains a reference to the last member of the group.

Field: **numberIn**

Type: **INTEGER**

Description: A counter that has the current number of objects in the group.

ASK Method: **GetRosterCard**

Parameters: None

Return Value: **INTEGER**

Description: Get a roster card for the new member.

ASK Method: **DelRosterCard**

Parameters: None

Return Value: **INTEGER**

Description: Delete a roster card.

# IStatObj

Module: **StatMod**

Derived From: **StatObj**

Description: Statistical monitor for **INTEGER** type.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Maximum</b>	<b>INTEGER</b>	No	<b>IStatObj</b>
<b>Minimum</b>	<b>INTEGER</b>	No	<b>IStatObj</b>

LMONITOR Method Defined By:

**access**            **IStatObj**

RMONITOR Method Defined By:

**raccess**            **IStatObj**

## FIELDS and METHODS

Field:            **Maximum**

Type:            **INTEGER**

Description: Maximum observation since last reset.

Fields:           **Minimum**

Type:            **INTEGER**

Description: Minimum observation since last reset.

**LMONITOR Method: access**

Description: Updates statistics based on observation. Increments the number of observations by 1. Updates minimum and maximum values if appropriate. Updates the sum by the value of the observation. Updates the sum squared by the squared value of the observation. Adds the value to the histogram if a histogram is previously created by **SetHistogram**.

**RMONITOR Method: raccess**

Description: Override method to add functionality to derived object.

Module: **StatMod**

Derived From: **TimedStatObj**

Description: Time-weighted statistical monitor for **INTEGER** type.

Field	Type	Private	Defined By
<b>Maximum</b>	<b>INTEGER</b>	No	<b>ITimedStatObj</b>
<b>Minimum</b>	<b>INTEGER</b>	No	<b>ITimedStatObj</b>
<b>value</b>	<b>INTEGER</b>	No	<b>ITimedStatObj</b>

ASK Method	Return Type	Private	Defined By
<b>Reset</b>	None	No	<b>ITimedStatObj</b>
<b>TAdjust</b>	None	No	<b>ITimedStatObj</b>

#### LMONITOR Method Defined By

**access** **ITimedStatObj**

#### RMONITOR Method Defined By

**raccess** **ITimedStatObj**

#### FIELDS and METHODS

Field: **Maximum:**

Type: **INTEGER;**

Description: Maximum observation since last reset.

Field: **Minimum:**

Type: **INTEGER;**

Description: Minimum observation since last reset.

Field: **Value:**

Type: **INTEGER;**

Description: The value being monitored.

LMONITOR Method: **access;**

Description: Updates statistics based on observation. Increments the number of observations by 1. Updates minimum and maximum values if appropriate. Adjusts time dependent values.

RMONITOR Method: **raccess;**

Description: Provides access to object if monitored statistic is accessed. Override method to add functionality to derived object.

ASK Method: **Reset**

Parameters: None

## ITimedStatObj (cont)

Return Value: None

Description: Resets **Minimum** and **Maximum** and sets current time.

ASK Method: **TAdjust**

Parameters: None

Return Value: None

Description: Updates time dependent values.

Module: **ListMod**

Derived From: None

Description: A “virtual” object that describes the fields all list objects have. All list objects are designed to hold RECORDs.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>numberIn</b>	<b>INTEGER</b>	No	<b>ListObj</b>

#### FIELDS and METHODS

Field: **numberIn**

Type: **INTEGER**

Description: A counter that has the current number of objects in the group.

# PriorityList

Module: **ResMod**

Derived From: **StatRankedObj**

Substitutes: **EntryObj** for: **ANYOBJ**

Description: Keeps a list, ranked by priority, of all objects waiting to receive a resource.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>numResources</b>	<b>INTEGER</b>	No	<b>PriorityList</b>

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>PriorityList</b>
<b>Adjust</b>	None	No	<b>PriorityList</b>
<b>Rank</b>	<b>INTEGER</b>	No	<b>PriorityList</b>
<b>Reset</b>	None	No	<b>PriorityList</b>

## FIELDS and METHODS

Field: **numResources**

Type: **INTEGER**

Description: **LRMONITORED INTEGER** by **IStatObj** and **ITimedStatObj**.

ASK Method: **Add**

Parameters: **IN NewMember : EntryObj**

Return Value: **REAL**;

Description: Inserts **NewMember** in proper ranking order.

ASK Method: **Adjust**

Parameters: **IN NewSeed**

Return Value: None

Description: Updates **numResources** field for statistics.

ASK Method: **Rank**

Parameters: **IN a,b : EntryObj**

Return Value: None

Description: Relative order of **a, b** based on priority.

ASK Method: **Reset**

Parameters: None

Return Value: None

Description: Resets monitors associated with **numberResources**.

Module: **ListMod**

Derived From: **BasicListObj**

**BasicQueueList**

Description: A composite object that provides a grouping mechanism based upon FIFO (first-in-first-out) insertion and removal.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>ObjLoad</b>	None	No	<b>QueueList</b>

#### FIELDS and METHODS

ASK Method: **ObjLoad**

Parameters: None

Return Value: **REAL**

Description: Load **QueueList** from a persistent data base.

# QueueObj

Module: **GrpMod**

Derived From: **BasicGroupObj**

Description: A composite object that provides a grouping mechanism based upon FIFO (first-in-first-out) insertion and removal.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>QueueObj</b>

## FIELDS and METHODS

ASK Method: **Add**

Parameters: **IN NewMember : ANYOBJ**

Return Value: **REAL**

Description: Add the new member first.

Module: **RandMod**

Derived From: None

Description: Provides a variety of statistical distributions for random number generation.

Field	Type	Private	Defined By
<b>originalSeed</b>	<b>INTEGER</b>	No	<b>RandomObj</b>
<b>currentSeed</b>	<b>INTEGER</b>	No	<b>RandomObj</b>
<b>antithetic</b>	<b>BOOLEAN</b>	No	<b>RandomObj</b>

ASK Method	Return Type	Private	Defined By
<b>Sample</b>	<b>REAL</b>	No	<b>RandomObj</b>
<b>SetSeed</b>	None	No	<b>RandomObj</b>
<b>Reset</b>	None	No	<b>RandomObj</b>
<b>UniformReal</b>	<b>REAL</b>	No	<b>RandomObj</b>
<b>UniformInt</b>	<b>INTEGER</b>	No	<b>RandomObj</b>
<b>Exponential</b>	<b>REAL</b>	No	<b>RandomObj</b>
<b>Normal</b>	<b>REAL</b>	No	<b>RandomObj</b>
<b>Gamma</b>	<b>REAL</b>	No	<b>RandomObj</b>
<b>Beta</b>	<b>REAL</b>	No	<b>RandomObj</b>
<b>Triangular</b>	<b>REAL</b>	No	<b>RandomObj</b>
<b>SetAntithetic</b>	None	No	<b>RandomObj</b>
<b>ObjInit</b>	None	No	<b>RandomObj</b>
<b>Dump</b>	None	No	<b>RandomObj</b>
<b>LogNormal</b>	None	No	<b>RandomObj</b>
<b>ObjInit</b>	None	No	<b>RandomObj</b>
<b>Erlang</b>	None	No	<b>RandomObj</b>
<b>Weibull</b>	None	No	<b>RandomObj</b>
<b>Poisson</b>	None	No	<b>RandomObj</b>
<b>Binomial</b>	None	No	<b>RandomObj</b>

## FIELDS and METHODS

Field: **antithetic**  
 Type: **BOOLEAN**  
 Description: Generate antithetic variates.

Field: **currentSeed**  
 Type: **INTEGER**  
 Description: Current seed - changes on every random draw.

Field: **originalSeed**  
 Type: **INTEGER**  
 Description: Argument in last **SetSeed** message.

## RandomObj (cont.)

ASK Method: **Binomial**

Parameters: None

Return Value: None

Description: Generate a random sample from the Binomial distribution. The binomial distribution represents the integer number of successes in "n" independent trials, each having the probability of success "p". The values of both "n" (number of trials) and "p" (probability) must be greater than 0.

ASK Method: **Beta**

Parameters: **IN alpha1: REAL**

**IN alpha2: REAL**

Return Value: REAL

Description: Returns a random sample from the beta distribution related to the gamma function where the result is restricted to the unit interval. Given arguments of **alpha1** and **alpha2** must be greater than 0;

```
alpha1, alpha2 > 0; Beta(alpha1,alpha2) =  
Gamma(alpha1,alpha1) / (Gamma(alpha1,alpha1) +  
Gamma(alpha2,alpha2))
```

ASK Method: **Dump**

Parameters: None

Return Value: None

Description:

ASK Method: **Erlang**

Parameters: None

Return Value: None

Description: The Erlang distribution is a special case of "Gamma" which results when "alpha" is an integer. If 'K = 1' this function is the same as the exponential distribution. Generate a random sample from the Erlangian distribution.

ASK Method: **Exponential**

Parameters: **IN mean: REAL**

Return Value: **REAL**

Description: Returns a random sample from the exponential distribution, mean > 0.

ASK Method: **Gamma**

Parameters: **IN mean: REAL**

**IN alpha: REAL**

Return Value: **REAL**

**Description:** Returns a random sample from the gamma distribution. This distribution has smaller variance and more control in parameter selection than the Exponential method, and can therefore be used to more realistically represent observable data. **mean, alpha > 0; mean = alpha \* beta** in the standard representation of this distribution.

**ASK Method:** **LogNormal**

**Parameters:** None

**Return Value:** None

**Description:** Generates a random sample from the Log Normal distribution. The log normal distribution is often used to characterize skewed data. The given "mean" must be greater than 0.

**ASK Method:** **Normal**

**Parameters:** **IN mean: REAL**

**IN sigma: REAL**

**Return Value:** **REAL**

**Description:** Returns a random sample from the normal distribution. This distribution generates the Gaussian bell-shaped curve. **sigma > 0.**

**ASK Method:** **ObjInit**

**Parameters:** None

**Return Value:** None

**Description:**

**ASK Method:** **Poisson**

**Parameters:** None

**Return Value:** None

**Description:** Generates a random sample from the Poisson distribution. Poisson distributions are often used to model the number of occurrences of some event in a given period of time. The value of **mu** must be greater than 0.

**ASK Method:** **Reset**

**Parameters:** None

**Return Value:** None

**Description:** Resets to the original seed last specified by **SetSeed.**

**ASK Method:** **Sample**

**Parameters:** None

**Return Value:** **REAL;**

**Description:** Returns sample S where:  $0.0 < S < 1.0$

**ASK Method:** **SetAntithetic**

**Parameters:** **IN onOff: BOOLEAN**

**Return Value:** None

## RandomObj (cont.)

Description: Toggles sampling with antithetic variates, default is **FALSE**.

ASK Method: **SetSeed**

Parameters: **IN NewSeed**

Type: **INTEGER**

Return Value: None

Description: All **RandomObjs** start with a default of **FetchSeed(1)**. A new seed can be generated at any time.

ASK Method: **Triangular**

Parameters: **IN min: REAL**

**IN mode: REAL**

**IN max: REAL**

Return Value: **REAL**

Description: Returns a random sample from the triangular distribution, **min < mean < max**.

ASK Method: **UniformInt**

Parameters: **IN lo: INTEGER**

**IN hi: INTEGER**

Return Value: **INTEGER**

Description: Returns a random sample from the uniform distribution in **[lo,hi]**.

ASK Method: **UniformReal**

Parameters: **IN lo: REAL**

**IN hi: REAL**

Return Value: **REAL**

Description: Returns a random sample from the uniform distribution in **[lo,hi]**.

ASK Method: **Weibull**

Parameters: None

Return Value: None

Description: Generates a random sample from the Weibull distribution. The Weibull function can be used to generalize distribution function implementation. By selecting the values of the parameters, several families can be represented. For example, if **shape = 1** the Weibull function is the same as **Exponential** with **mean = scale**. The **shape** and **scale** arguments must be greater than 0.

Module: **ListMod**

Derived From: **BasicListObj**

**BasicRankedList**

Description: A composite object that allows the user to specify a relative order between records being added to the group. Records will be added to the group based upon this ordering. If the Rank method is not overridden the insertion order of the group is undefined.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>ObjLoad</b>	None	No	<b>RankedList</b>

#### FIELDS and METHODS

ASK Method: **ObjLoad**

Return Value: None

Parameters: None

Description: Establishes ranked list when read from a persistent data base.

# RankedObj

Module: **GrpMod**

Derived From: **BasicGroupObj**

**BasicRankedObj**

Description: A composite object that allows the user to specify a relative order between objects being added to the group. Objects will be added to the group based upon this ordering. If the **Rank** method is not overridden the insertion order of the group is undefined.

<u>ASKMethod</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>RankedObj</b>

## FIELDS and METHODS

ASK Method: **Add**

Return Value: None

Parameters **IN NewMember : ANYOBJ**

Description: Performs ordered insertion.

Module: **ResMod**

Derived From:None

Description: This object type provides a blocking resource acquisition mechanism. This mechanism is particularly useful for simulation scenarios. Object instances requesting one or more resources from the created (instantiated) pool will be granted acquisition as the resources are or become available. The requesting method will block (**WAIT FOR**) at the point of request until the request can be fulfilled or it is interrupted.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
AllocationList	AllocQueueObj	No	ResourceObj
MaxResources	INTEGER	No	ResourceObj
PendingList	PriorityList	No	ResourceObj
PendingResources	INTEGER	No	ResourceObj
Resources	INTEGER	No	ResourceObj
<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
AllocCount	INTEGER	No	ResourceObj
AllocMaximum	INTEGER	No	ResourceObj
AllocMean	REAL	No	ResourceObj
AllocMinimum	INTEGER	No	ResourceObj
AllocStdDev	REAL	No	ResourceObj
AllocVariance	REAL	No	ResourceObj
AllocWtdMean	REAL	No	ResourceObj
AllocWtdStdDev	REAL	No	ResourceObj
AllocWtdVariance	REAL	No	ResourceObj
Allocate	None	Yes	ResourceObj
Cancel	None	No	ResourceObj
Create	None	No	ResourceObj
Find	#ANYOBJ	Yes	ResourceObj
IncrementResourcesBy	None	No	ResourceObj
NumberAllocatedTo	INTEGER	No	ResourceObj
ObjInit;	None	No	ResourceObj
ObjTerminate;	None	No	ResourceObj
PendWtdMean	REAL	No	ResourceObj
PendWtdStdDev	REAL	No	ResourceObj
PendWtdVariance	REAL	No	ResourceObj
PendingCount	INTEGER	No	ResourceObj
PendingMaximum	INTEGER	No	ResourceObj
PendingMean	REAL	No	ResourceObj
PendingMinimum	INTEGER	No	ResourceObj
PendingStdDev	REAL	No	ResourceObj
PendingVariance	REAL	No	ResourceObj
ReportAvailability	INTEGER	No	ResourceObj

## ResourceObj (cont)

ASK Method	Return Type	Private	Defined By
ReportNumberPending	INTEGER	No	ResourceObj
Reset	None	No	ResourceObj
ResetAllocationStats	None	No	ResourceObj
ResetPendingStats	None	No	ResourceObj
SetAllocHistogram	None	No	ResourceObj
SetAllocationStats	None	No	ResourceObj
SetPendHistogram	None	No	ResourceObj
SetPendStats	None	No	ResourceObj
TakeBack	None	No	ResourceObj
Transfer	None	No	ResourceObj

TELL Method	Return Type	Private	DefinedBy
DecrementResourcesBy	None	No	ResourceObj

WAITFOR Method	Return Type	Private	DefinedBy
GetResource	None	No	ResourceObj
Give	None	No	ResourceObj
PriorityGive	None	No	ResourceObj
TimedGive	None	No	ResourceObj

### FIELDS and METHODS

Field: **AllocationList**  
 Type: **AllocQueueObj**  
 Description: A group of **EntryObjs** describing currently allocated resources.

Field: **PendingList**  
 Type: **PriorityList**  
 Description: A group, ordered by priority (highest to least), of **EntryObjs** describing resource requests.

Field: **MaxResources**  
 Type: **INTEGER**  
 Description: Total number of resources potentially available from the **ResourceObj** instance.

Field: **Resources**  
 Type: **INTEGER**  
 Description: The number of currently available resources for the instance.

Field: **PendingResources**  
 Type: **INTEGER**  
 Description: Number of resources requested.

ASK Method: **ObjInit**

Return Value: None

Parameters: None

Description: Creates all internal data structures.

ASK Method: **ObjTerminate**

Return Value: None

Parameters: None

Description: Empties and disposes internal data structures.

ASK Method: **ReportAvailability**

Parameters: None

Return Value: **INTEGER**

Description: How many could be obtained immediately.

ASK Method: **ReportNumberPending**

Parameters: None

Return Value: **INTEGER**

Description: Number requested that have not yet been provided.

ASK Method: **NumberAllocatedTo**

Parameters: **IN Object: #ANYOBJ**

Return Value: **INTEGER**

Description: The number of resources that have been allocated to object.

ASK Method: **Create**

Parameters: **IN number: INTEGER**

Return Value: None

Description: Initialize resource to have number of resources.

ASK Method: **IncrementResourcesBy**

Parameters: **IN incBy: INTEGER**

Return Value: None

Description: Increase the number of resources in the total resource pool.

TELL Method: **DecrementResourcesBy**

Parameters: **IN decBy: INTEGER**

Return Value: None

Description: Decrease the number of resources in the total resource pool.

## ResourceObj (cont)

WAITFOR Method: **Give**

Parameters: **IN Me: #ANYOBJ**  
**IN numberDesired: INTEGER**

Return Value: None

Description: Requests acquisition of '**numberDesired**' resources and allocates them to object '**Me**'. Calling method is blocked until resource(s) become available on a first come first served basis.

WAITFOR Method: **TimedGive**

Parameters: **IN Me: #ANYOBJ**  
**IN numberDesired: INTEGER**  
**IN timePeriod: REAL**

Return Value: None

Description: Same as **Give** above except if resources are not acquired within **timePeriod** simulation units the calling method will be resumed with an **INTERRUPT**. Calling code should have an **ON INTERRUPT** clause to handle this case.

WAITFOR Method: **PriorityGive**

Parameters: **IN Me: #ANYOBJ**  
**IN numberDesired: INTEGER**  
**IN priority: REAL**

Return Value: None

Description: Same as **Give** above except those requests that must be queued will be queued based upon the given priority. The higher the priority argument the more forward in the pending list the request will be placed.

WAITFOR Method: **GetResource**

Parameters: **IN Me: #ANYOBJ**  
**IN numberDesired: INTEGER**  
**IN timePeriod: REAL**  
**IN priority: REAL**

Return Value: None

Description: A combination of **TimedGive** and **PriorityGive**. Requesting method will '**timeout**' if resource is not allocated within **timePeriod** simulation units and calling method will be resumed with an **INTERRUPT** condition (an **ON INTERRUPT** clause should be provided), queued requests will be allocated based upon given priority.

ASK Method: **TakeBack**

Parameters: **IN FromMe: #ANYOBJ**  
**IN numberReturned: INTEGER**

Return Value: None

Description: This returns previously allocated resource(s) that were acquired by object **FromMe**. If the object did not have the resource to begin with a runtime error will result.

ASK Method: **Transfer**

Parameters: **IN From, To: #ANYOBJ**  
**IN numberTrans: INTEGER**

Return Value: None

Description: Transfers “ownership” of resource from **From** object to **To** object. User must provide code to notify both objects of condition. This method is provided to facilitate preemption allocation.

ASK Method: **Cancel**

Parameters: **IN Object: #ANYOBJ**  
**IN numberToCancel: INTEGER**

Return Value: None

Description: Cancels all or portion of request for resource(s) that is currently on pending list - user must provide code to notify object of cancellation.

ASK Method: **Reset**

Parameters: None

Return Value: None

Description: Resets both Allocation and Pending statistics.

ASK Method: **ResetAllocationStats**

Parameters: None

Return Value: None

Description: Resets statistics for the **AllocationList**.

ASK Method: **SetAllocationStats**

Parameters: **IN on**

Type: **BOOLEAN**

Return Value: None

Description: Statistics gathering may be turned on (**TRUE**) or off - default is off.

ASK Method: **SetAllocHistogram**

Parameters: **IN low: INTEGER**  
**IN high: INTEGER**  
**IN interval: INTEGER**

Return Value: None

Description: Set up resource allocation histogram bounds.

ASK Method: **AllocMaximum**

Parameters: None

Return Value: **INTEGER**

## ResourceObj (cont)

Description: The maximum number allocated at any time up to the present simulation time.

ASK Method: **AllocMinimum**

Parameters: None

Return Value: **INTEGER**

Description: The minimum number allocated at any time up to the present simulation time.

ASK Method: **AllocCount**

Parameters: None

Return Value: **INTEGER**

Description: Returns the number of times the number of units of the resource was allocated at the present simulation time.

ASK Method: **AllocMean**

Parameters: None

Return Value: **REAL**

Description: The mean number of objects on the allocation list.

ASK Method: **AllocVariance**

Parameters: None

Return Value: **REAL**

Description: The variance of the number of objects allocated.

ASK Method: **AllocStdDev**

Parameters: None

Return Value: **REAL**

Description: The standard deviation of the number of objects on the allocation list.

ASK Method: **AllocWtdMean**

Parameters: None

Return Value: **REAL**

Description: The mean, weighted with respect to time, of objects on the allocation list.

ASK Method: **AllocWtdVariance**

Parameters: None

Return Value: **REAL**

Description: The weighted variance of the number of units of the resource with respect to time, of objects on the allocation list.

ASK Method: **AllocWtdStdDev**

Parameters: None

Return Value: **REAL**

Description: The weighted standard deviation with respect to time, of objects on the allocation list.

ASK Method: **ResetPendingStats**

Parameters: None

Return Value: **REAL**

Description: Reset the pending stats.

ASK Method: **SetPendingStats**

Parameters: None

Return Value: **REAL**

Description: Set up resource pending histogram bounds.

ASK Method: **SetPendHistogram**

Parameters: None

Return Value: **REAL**

Description: Set up resource pending histogram bounds.

ASK Method: **PendingMaximum**

Parameters: None

Return Value: **REAL**

Description: The maximum number of objects requested but not yet granted.

ASK Method: **PendingMinimum**

Parameters: None

Return Value: **REAL**

Description: The minimum number of objects requested but not yet granted.

ASK Method: **PendingCount**

Parameters: None

Return Value: **REAL**

Description: The number of times the object pending was updated.

ASK Method: **PendingMean**

Parameters: None

Return Value: **REAL**

Description: The mean of the number of objects on the pending list.

ASK Method: **PendingVariance**

Parameters: None

Return Value: **REAL**

Description: The variance of the number of objects requested but not yet granted.

ASK Method: **PendingStdDev**

Parameters: None

## ResourceObj (cont)

Return Value: **REAL**

Description: The standard deviation of the number of objects requested but not yet granted.

ASK Method: **PendWtdMean**

Parameters: None

Return Value: **REAL**

Description: The average, weighted with respect to time, of objects on the pending list.

ASK Method: **PendWtdVariance**

Parameters: None

Return Value: **REAL**

Description: The weighted variance of the number of objects on the pending list.

ASK Method: **PendWtdStdDev**

Parameters: None

Return Value: **REAL**

Description: The weighted standard deviation of the number of objects on the pending list.

ASK Method: **Find**

Parameters: **IN Obj: #ANYOBJ**

Return Value: **#ANYOBJ**

Description: For private use by the **ResourceObj**, locates a particular object on the pending list for transfer or cancellation purposes.

ASK Method: **Allocate**

Parameters: **IN Me: #ANYOBJ**

**IN number: INTEGER**

**IN priority: REAL**

Return Value: None

Description: For private use by the **ResourceObj**, actually executes granting resource request and all attendant bookkeeping.

Module: **StatMod**  
 Derived From: **StatObj**  
 Description: Statistical monitor for **REAL** type.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Maximum</b>	<b>REAL</b>	No	<b>RStatObj</b>
<b>Minimum</b>	<b>REAL</b>	No	<b>RStatObj</b>

<u>LMONITOR Method</u>	<u>Defined By</u>
<b>access</b>	<b>RStatObj</b>

<u>RMONITOR Method</u>	<u>Defined By</u>
<b>raccess</b>	<b>RStatObj</b>

#### FIELDS AND METHODS

Field: **Maximum**  
 Type: **REAL**  
 Description: Maximum observation since last reset.

Fields: **Minimum**  
 Type: **REAL**  
 Description: Minimum observation since last reset.

**LMONITOR Method: access**  
 Description: Provides access to object if monitored statistics is accessed.

**RMONITOR Method: raccess**  
 Description: Updates statistics based on observation.

# RTimedStatObj

Module: **StatMod**

Derived From: **TimedStatObj**

Description: Weighted statistical monitor for **REAL** type.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Maximum</b>	<b>REAL</b>	No	<b>RTimedStatObj</b>
<b>Minimum</b>	<b>REAL</b>	No	<b>RTimedStatObj</b>
<b>value</b>	<b>INTEGER</b>	No	<b>RTimedStatObj</b>

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Reset</b>	None	No	<b>RTimedStatObj</b>
<b>TAdjust</b>	None	No	<b>RTimedStatObj</b>

## LMONITOR Method Defined By

**access** **RTimedStatObj**

## RMONITOR Method Defined By

**raccess** **RTimedStatObj**

## FIELDS AND METHODS

Field: **Maximum**

Type: **REAL**

Description: Maximum observation since last reset.

Fields: **Minimum**

Type: **REAL**

Description: Minimum observation since last reset.

**LMONITOR** Method: **access**

Description: Updates statistics based on observation.

**RMONITOR** Method: **raccess**

Description: Provides access to object if monitored statistics is accessed.

ASK Method: **TAdjust**

Parameters: None

Return Value: None

Description: Updates time dependent values.

ASK Method: **Reset**

Parameters: None

Return Value: None

Description: Resets Minimum, Maximum, and sets **resetTime**.

Module: **SimMod**  
 Derived From: None  
 Description: The simulation control object lets the developer resolve time ties.

This object provides mechanisms for fine-tuning the execution of activities within a simulation. To take advantage of its capabilities (tie-breaking and time advance notification) the user must derive an object type from this class and override the desired behaviors. In addition, either or both, of the 'Set' methods must be called to notify the simulation controller of the desired behavior.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>TimeAdvance</b>	<b>REAL</b>	No	<b>SimControlObj</b>
<b>ChooseNext</b>	<b>ACTID</b>	No	<b>SimControlObj</b>
<b>ActivityTrace</b>	<b>ACTID</b>	No	<b>SimControlObj</b>
<b>SetTieBreaking</b>	None	No	<b>SimControlObj</b>
<b>SetTimeAdvance</b>	None	No	<b>SimControlObj</b>
<b>SetActivityTrace</b>	None	No	<b>SimControlObj</b>

## FIELDS AND METHODS

ASK Method: **ActivityTrace**

Parameters: **IN activity : ACTID**

Return Value: **ACTID**

Description: Trace simulation activities. Method is activated right before the activity is activated or reactivated and just before it is suspended for a WAIT.

ASK Method: **ChooseNext**

Parameters: **IN group : ActivityGroup**

Return Value: **ACTID**

Description: Choose the next event in the case of a time tie. The method is invoked when there are two or more activities scheduled for the identical simulation time. The method must return the activity which should be activated next.

ASK Method: **SetActivityTrace**

Parameters: **IN flag : BOOLEAN**

Return Value: None

Description: If TRUE, then **ActivityTrace** is called for each new activity.

ASK Method: **SetTieBreaking**

Parameters: **IN flag : BOOLEAN**

Return Value: None

Description: If TRUE, then **ChooseNext** is called to resolve time ties.

## SimControlObj (cont)

ASK Method: **SetTimeAdvance**

Parameters: **IN flag : BOOLEAN**

Return Value: **ACTID**

Description: If TRUE, then **TimeAdvance** is called when the simulation clock is to advance.

ASK Method: **TimeAdvance**

Parameters: **IN newTime : REAL**

Return Value: **REAL**

Description: Notification of advance of the simulation clock. This method is invoked when the simulation time is about to be advanced. Any desired work may be performed from this method including scheduling more activities. The method must return either the current simulation time which will allow any newly scheduled activities to be performed or the new proposed simulation time which has been passed in as an argument.

Module: **GrpMod**  
 Derived From: **ExpandedBasicGroupObj**  
                   **RankedObj**  
 Description: LIFO, FIFO and ranked groups with delay and level monitoring.

Field	Type	Private	Defined By
<b>oDelayMonitor</b>			
<b>oLevelMonitor</b>			

ASK Method	Return Type	Private	Defined By
<b>AddDelayMonitor</b>	<b>REAL</b>	No	<b>SimQueueObj</b>
<b>AddLevelMonitor</b>	<b>ACTID</b>	No	<b>SimQueueObj</b>
<b>RemoveDelayMonitor</b>	<b>ACTID</b>	No	<b>SimQueueObj</b>
<b>RemoveLevelMonitor</b>	None	No	<b>SimQueueObj</b>
<b>ObjInit</b>	None	No	<b>SimQueueObj</b>
<b>ResetStats</b>	None	No	<b>SimQueueObj</b>
<b>ObjTerminate</b>	None	No	<b>SimQueueObj</b>
<b>Add</b>	None	No	<b>SimQueueObj</b>
<b>GetRosterCard</b>	None	No	<b>SimQueueObj</b>
<b>DelRosterCard</b>	None	No	<b>SimQueueObj</b>

**FIELDS AND METHODS**

Field: **oDelayMonitor**  
 Type: **RStatObj**  
 Description: Statistics.

Field: **oLevelMonitor**  
 Type: **RTimedStatObj**  
 Description: Weighted statistics.

ASK Method: **AddDelayMonitor**  
 Parameters: **IN mon : RStatObj**  
 Return Value: None  
 Description: Adds monitor to **tdelay** field.

ASK Method: **AddLevelMonitor**  
 Parameters: **IN mon : RTimedStatObj**  
 Return Value: None  
 Description: Adds monitor to **rLevel** field.

## SimQueueObj (cont)

ASK Method: **RemoveDelayMonitor**

Parameters: None

Return Value: None

Description: Removes monitor from **tDelay** field.

ASK Method: **RemoveLevelMonitor**

Parameters: None

Return Value: None

Description: Removes monitor from **rLevel** field.

ASK Method: **ObjInit**

Parameters: None

Return Value: None

Description: Default version does nothing.

ASK Method: **ResetStats**

Parameters: None

Return Value: None

Description: Resets **numEntries**, **numExits** and monitor objects.

ASK Method: **ObjTerminate**

Parameters: None

Return Value: None

Description: Verification and overhead.

ASK Method: **Add**

Parameters: **IN NewMember : ANYOBJ**

Return Value: **ACTID**

Description: Supports LIFO, FIFO and ranked insertions.

ASK Method: **GetRosterCard**

Parameters: **IN NewMember : ANYOBJ**

Return Value: None

Description: Internal

ASK Method: **DelRosterCard**

Parameters: **IN member : ANYOBJ**

Return Value: None

Description: Internal

Module: **ListMod**

Derived From: **BasicListObj**

**BasicStackList**

Description: A composite object that provides a grouping mechanism based upon LIFO (last-in-first-out) insertion and removal.

<u>ASK Method</u>	<u>ReturnType</u>	<u>Private</u>	<u>Defined By</u>
<b>ObjLoad</b>	None	No	<b>StackList</b>

#### FIELDS and METHODS

ASK Method: **ObjLoad**

Parameters: None

Return Value: None

Description: Load **StackList** from a persistent data base.

# StackObj

Module: **GrpMod**

Derived From: **BasicGroupObj**

Description: A composite object that provides a grouping mechanism based upon LIFO (last-in-first-out) insertion and removal.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>StackObj</b>

## FIELDS and METHODS

ASK Method: **Add**

Parameters: **NewMember**

Return Value: **REAL**

Description: Add the new member last.

Module: **ListMod**

Derived From: **BStatListObj**

**BStatBTreeList**

Description: Same functionality as **BasicBTreeList** plus accumulates statistics on number of records in list.

# StatBTreeObj

Module: **GrpMod**

Derived From: **BStatGroupObj**

Description: Same functionality as a **BTreeObj** plus statistical accumulation on number of objects kept in group.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>StatBTreeObj</b>

## FIELDS and METHODS

ASK Method: **Add**

Parameters: **NewMember**

Return Value: **REAL**

Description: Performs balanced insertions, accumulates stats.

Module: **GrpMod**

Derived From:None

Description: Provides basic fields required by group objects that accumulate statistics.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>number</b>	<b>StatINTEGER</b>	No	<b>StatGroupObj</b>

#### FIELDS and METHODS

Field: **number**

Type: **StatINTEGER**

Description: A monitored integer field that parallels the **numberIn** field of group objects. It is this field upon which statistics are kept.

## StatListObj

Module: **ListMod**

Derived From:None

Description: Provides basic fields required by list objects that accumulate statistics.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>DefinedBy</u>
<b>number</b>	<b>StatINTEGER</b>	No	<b>StatListObj</b>

### FIELDS and METHODS

Field: **number**

Type: **StatINTEGER**

Description: A monitored integer field that parallels the **numberIn** field of group objects. It is this field upon which statistics are kept.

Module: **StatMod**

Derived From: None

Description: Provides basic functionality and fields for Statistical accumulation.

Field	Type	Private	Defined By
<b>Count</b>	<b>INTEGER</b>	No	<b>StatObj</b>
<b>Sum</b>	<b>REAL</b>	No	<b>StatObj</b>
<b>SumOfSquares</b>	<b>REAL</b>	No	<b>StatObj</b>
<b>high</b>	<b>INTEGER</b>	No	<b>StatObj</b>
<b>interval</b>	<b>INTEGER</b>	No	<b>StatObj</b>
<b>low</b>	<b>INTEGER</b>	No	<b>StatObj</b>

ASK Method	Return Type	Private	Defined By
<b>ObjTerminate</b>	None	No	<b>StatObj</b>
<b>Mean</b>	<b>REAL</b>	No	<b>StatObj</b>
<b>MeanSquare</b>	<b>REAL</b>	No	<b>StatObj</b>
<b>Reset</b>	None	No	<b>StatObj</b>
<b>GetHistogram</b>	None	No	<b>StatObj</b>
<b>SetHistogram</b>	None	No	<b>StatObj</b>
<b>StdDev</b>	<b>REAL</b>	No	<b>StatObj</b>
<b>Variance</b>	<b>REAL</b>	No	<b>StatObj</b>

#### FIELDS AND METHODS

Field: **Count**

Type: **INTEGER**

Description: Number of observations.

ASK Method: **Mean**

Parameters: None

Return Value: **REAL**

Description: Compute the mean of observations.

ASK Method: **MeanSquare**

Parameters: None

Return Value: **REAL**

Description: Compute the mean square of observations.

Field: **Sum**

Type: **REAL**

Description: Sum of observations.

## StatObj (cont.)

Field: **SumOfSquares**  
Type: **REAL**  
Description: Sum of squares of observations

Field: **high**  
Type: **INTEGER**  
Description: High bound of histogram.

Field: **interval**  
Type: **INTEGER**  
Description: Number of intervals.

Field: **low**  
Type: **INTEGER**  
Description: Low bound of histogram.

ASK Method: **SetHistogram**  
Parameters: **IN Low: INTEGER**  
**IN High: INTEGER**  
**IN Interval: INTEGER**

Return Value: None

Description: Set up parameters for histogram collection: bin for all values lower than **Low** and higher than **High** will automatically be allocated to the 0th element and **High - Low DIV Interval + 1** element respectively.

ASK Method: **StdDev**  
Parameters: None  
Return Value: **REAL**  
Description: Compute the standard deviation of the observations.

ASK Method: **Variance**  
Parameters: None  
Return Value: **REAL**  
Description: Compute the variance of the observations.

ASK Method: **Reset**  
Parameters: None  
Return Value: **REAL**  
Description: Reset statistics.

ASK Method: **GetHistogram**  
Parameters: None  
Return Value: None  
Description: Get the monitor's histogram pointer.

ASK Method: **ObjTerminate**

Parameters: None

Return Value: None

Description: Disposes histogram if it exists.

## StatQueueList

Module: **ListMod**

Derived From: **BStatListObj**

**BStatQueueList**

Description: Same functionality as **BasicQueueList** plus accumulates statistics on number of records in list.

Module: **GrpMod**

Derived From: **BStatGroupObj**  
**BStatQueueObj**

Description: Same functionality as a **QueueObj** plus statistical data provided from the number of objects in the queue.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>StatQueueObj</b>

## FIELDS and METHODS

ASK Method: **Add**

Return Value: None

Parameters: **NewMember**

Description: Add new member last.

## StatRankedList

Module: **ListMod**

Derived From: **BStatListObj**

**BStatRankedList**

Description: Same functionality as **BasicRankedList** plus accumulates statistics on number of records in list.

Module: **GrpMod**

Derived From: **BStatGroupObj**

**BStatRankedObj**

Description: Same functionality as **RankedObj** plus statistical accumulation on number of objects kept in group.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>StatRankedObj</b>
<b>AddFirst</b>	None	No	<b>StatRankedObj</b>
<b>GetRosterCard</b>	None	No	<b>StatRankedObj</b>
<b>DelRosterCard</b>	None	No	<b>StatRankedObj</b>
<b>ObjInit</b>	None	No	<b>StatRankedObj</b>

#### FIELDS and METHODS

ASK Method: **Add**

Return Value: None

Parameters: **NewMember**

Description: Add new member first.

ASK Method: **AddFirst**

Return Value: None

Parameters: **NewMember**

Description: Add at beginning of membership list.

ASK Method: **DelRosterCard**

Return Value: None

Parameters: **rec = ANYREC**  
**obj : ANYOBJ**

Description: Internal routine.

ASK Method: **GetRosterCard**

Return Value: None

Parameters: **NewMember**

Description: Internal routine.

ASK Method: **ObjInit**

Return Value: None

Parameters: None

Description: Sets field required for invoking ranked **Add** behavior.

## StatStackList

Module: **ListMod**

Derived From: **BStatListObj**

**BStatStackList**

Description: Same functionality as **BasicStackList** plus accumulates statistics on number of records in list.

Module: **GrpMod**

Derived From: **BStatGroupObj**

Description: Same functionality as a **stackObj** plus statistical accumulation on number of objects in group.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Add</b>	None	No	<b>StatStackObj</b>
<b>ObjInit</b>	None	No	<b>StatGroupObj</b>

#### FIELDS and METHODS

ASK Method: **Add**

Return Value: None

Parameters: None

Description: Add new member first.

ASK Method: **ObjInit**

Return Value: None

Parameters: None

Description: Sets field required for invoking LIFO behavior .

# StreamObj

Module: **IOMod**

DerivedFrom: None

Description: Object type provides the basic file (stream) input-output capabilities.

<u>Field</u>	<u>Type</u>	<u>Private</u>	<u>Defined By</u>
<b>eof</b>	<b>BOOLEAN</b>	No	<b>StreamObj</b>
<b>fileName</b>	<b>STRING</b>	No	<b>StreamObj</b>
<b>handleIN</b>	<b>ANYREC</b>	Yes	<b>StreamObj</b>
<b>handleOUT</b>	<b>ANYREC</b>	Yes	<b>StreamObj</b>
<b>haltOnErr</b>	<b>BOOLEAN</b>	No	<b>StreamObj</b>
<b>ioResult</b>	<b>INTEGER</b>	No	<b>StreamObj</b>
<b>isBinary</b>	<b>BOOLEAN</b>	No	<b>StreamObj</b>

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Close</b>	None	No	<b>StreamObj</b>
<b>Delete</b>	None	No	<b>StreamObj</b>
<b>Dump</b>	None	No	<b>StreamObj</b>
<b>GetPosition</b>	None	Yes	<b>StreamObj</b>
<b>IsOpen</b>	None	No	<b>StreamObj</b>
<b>ObjInit</b>	None	No	<b>StreamObj</b>
<b>ObjTerminate</b>	None	No	<b>StreamObj</b>
<b>Open</b>	None	No	<b>StreamObj</b>
<b>Position</b>	None	Yes	<b>StreamObj</b>
<b>ReadBlock</b>	None	No	<b>StreamObj</b>
<b>ReadChar</b>	None	No	<b>StreamObj</b>
<b>ReadInt</b>	None	No	<b>StreamObj</b>
<b>ReadLine</b>	None	No	<b>StreamObj</b>
<b>ReadReal</b>	None	No	<b>StreamObj</b>
<b>ReadString</b>	None	No	<b>StreamObj</b>
<b>SetHaltOnErr</b>	None	No	<b>StreamObj</b>
<b>WriteBlock</b>	None	No	<b>StreamObj</b>
<b>WriteChar</b>	None	No	<b>StreamObj</b>
<b>WriteExp</b>	None	No	<b>StreamObj</b>
<b>WriteHex</b>	None	No	<b>StreamObj</b>
<b>WriteInt</b>	None	No	<b>StreamObj</b>
<b>WriteLn</b>	None	No	<b>StreamObj</b>
<b>WriteReal</b>	None	No	<b>StreamObj</b>
<b>WriteString</b>	None	No	<b>StreamObj</b>

## FIELDS and METHODS

Field: **eof**  
Type: **BOOLEAN**  
Description: Set to **TRUE** when the last item in the file is read.

Field: **filename**  
Type: **STRING**  
Description: Name of the file.

Field: **haltOnErr**  
Type: **BOOLEAN**  
Description: If **TRUE**, generate a runtime error when an inout/output error is detected.

Field: **handleIN**  
Type: **ANYREC**  
Description: Internal and implementation specific.

Field: **handleOUT**  
Type: **ANYREC**  
Description: Internal and implementation specific.

Field: **ioResult**  
Type: **INTEGER;**  
Description: The result of the last IO request.

Field: **isBinary**  
Type: **BOOLEAN**  
Description: True when file last opened in Binary mode.

ASK Method: **Close**  
Return Value: None  
Parameters: None  
Description: Concludes access the opened file.

ASK Method: **Delete**  
Return Value: None  
Parameters: None  
Description: Removes the opened file from the disk storage device.

ASK Method: **Dump**  
Return Value: None  
Parameters: None  
Description: Prints to **stdout** basic information regarding the object instance.

## StreamObj (cont.)

ASK Method: **GetPosition**

Return Value: **INTEGER**

Parameters: None

Description: Returns the position (in bytes) from the beginning of the file to the current location of the file pointer. The file pointer is advanced by reads, writes or explicit positioning to the file.

ASK Method: **IsOpen**

Return Value: None

Parameters: None

Description: Has a file been opened.

ASK Method: **ObjInit**

Return Value: None

Parameters: None

ASK Method: **ObjTerminate**

Return Value: None

Parameters: None

Description: Ensures that the file is closed before disposing of the object.

ASK Method: **Open**

Return Value: None

Parameters: **IN FileName: STRING**  
**IN IOdirection: FileUseType**

Description: Sets up "communication" between user and disk file named 'FileName.'  
Files may be opened for input, output or both depending on the constant 'IOdirection.'

ASK Method: **Position**

Return Value: None

Parameters: **IN moveTo: INTEGER**

Description: Moves the file pointer **moveTo** bytes from the beginning of the file. Useful for random accessing files.

ASK Method: **ReadBlock**

Return Value: None

Parameters: **IN buffer: ANYREC**  
**IN size: INTEGER**  
**IN blocknum: INTEGER**

Description: Reads size bytes from the location size \* blocknum in the file into the record 'buffer.'

ASK Method: **ReadChar**

Return Value: None

Parameters: **OUT ch: CHAR**

Description: Reads a single character from the currently opened file.

ASK Method: **ReadInt**

Return Value: None

Parameters: **OUT n: INTEGER**

Description: Reads a single integer from the currently opened file.

ASK Method: **ReadLine**

Return Value: None

Parameters: **OUT str: STRING**

Description: Reads from current position to end of line NOT including the newline. If line is longer than **str**, as much as will fit is read, and remainder of line is truncated

ASK Method: **ReadReal**

Return Value: None

Parameters: **OUT x: REAL**

Description: Read a real number from the file.

ASK Method: **ReadString**

Return Value: None

Parameters: **OUT str: STRING**

Description: Reads up to the next space or tab.

ASK Method: **SetHaltOnErr**

Return Value: None

Parameters: None

Description: Set **haltOnErr** field.

ASK Method: **writeBlock**

Return Value: None

Parameters: **IN buffer: ANYREC**

**IN size: INTEGER**

**IN blocknum: INTEGER**

Description: Writes size bytes from the beginning of the record 'buffer' to the file starting at file position size \* blocknum bytes from the beginning of the file.

ASK Method: **writeChar**

Return Value: None

Parameters: **IN ch: CHAR**

Description: Writes a single character to the opened file.

ASK Method: **writeExp**

## StreamObj (cont.)

Return Value: None

Parameters: **IN num: REAL**  
**IN fieldwidth: INTEGER**  
**IN precision: INTEGER**

Description: Writes a real number, in exponential form ( 3.14 e 10) to the currently opened file.

ASK Method: **writeHex**

Return Value: None

Parameters: **IN num: INTEGER**  
**IN fieldwidth: INTEGER**

Description: Writes the integer value **num** to the currently opened file as a hexadecimal number.

ASK Method: **writeInt**

Return Value: None

Parameters: **IN num: INTEGER**  
**IN fieldwidth: INTEGER**

Description: Writes the integer value **num** to the currently opened file.

ASK Method: **writeLn**

Return Value: None

Parameters: None

Description: Writes a single newline to the currently opened file.

ASK Method: **writeReal**

Return Value: None

Parameters: **IN num: REAL**  
**IN fieldwidth: INTEGER**  
**IN precision: INTEGER**

Description: Writes a single real number in standard form (i.e., 123.889) to the file. '**precision**' argument indicates the number of places to the right of the decimal to write.

ASK Method: **writeString**

Return Value: None

Parameters: **IN str: STRING**

Description: Writes a single string to the opened file.

Module: **StatMod**

Derived From: **StatObj**

Description: Adds fields and methods for statistical accumulation weighted with respect to simulation time.

Field	Type	Private	Defined By
<b>LastTime</b>	<b>REAL</b>	No	<b>TimedStatObj</b>
<b>FirstTime</b>	<b>REAL</b>	No	<b>TimedStatObj</b>

ASK Method	Return Type	Private	Defined By
<b>GetHistogram</b>	<b>REAL</b>	No	<b>TimedStatObj</b>
<b>Mean</b>	<b>REAL</b>	No	<b>TimedStatObj</b>
<b>MeanSquare</b>	<b>REAL</b>	No	<b>TimedStatObj</b>
<b>Reset</b>	None	No	<b>TimedStatObj</b>
<b>UpdateHistogram</b>	None	Yes	<b>TimedStatObj</b>

## FIELDS and METHODS

Field: **LastTime**

Type: **REAL**

Description: Simulation time of last observation.

Field: **FirstTime**

Type: **REAL**

Description: Simulation time of first observation.

ASK Method: **Mean**

Return Value: **REAL**

Description: Compute the time-weighted mean.

ASK Method: **MeanSquare**

Return Value: **REAL**

Description: Returns the time-weighted mean square.

ASK Method: **Reset**

Return Value: None

Description: Reset statistics.

ASK Method: **UpdateHistogram**

Return Value: None

Parameters: **IN value: REAL**

Description: Use value to update a cell in the histogram array.

## TimedStatObj (cont)

ASK Method: **GetHistogram**

Return Value: None

Parameters: IN value: **REAL**

Description: Get the monitor's histogram pointer.

Module: **SimMod**

Derived From: None

Description: Provides asynchronous rendezvous capability for TELL methods.

<u>ASK Method</u>	<u>Return Type</u>	<u>Private</u>	<u>DefinedBy</u>
<b>Dump</b>	None	No	<b>TriggerObj</b>
<b>InterruptTrigger</b>	None	No	<b>TriggerObj</b>
<b>NumWaiting</b>	None	No	<b>TriggerObj</b>
<b>ObjInit</b>	None	No	<b>TriggerObj</b>
<b>ObjTerminate</b>	None	No	<b>TriggerObj</b>
<b>Release</b>	None	No	<b>TriggerObj</b>

<u>TELL Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Trigger</b>	None	No	<b>TriggerObj</b>

<u>WAITFOR Method</u>	<u>Return Type</u>	<u>Private</u>	<u>Defined By</u>
<b>Fire</b>	None	No	<b>TriggerObj</b>

## FIELDS and METHODS

ASK Method: **Dump**

Return Value: None

Parameters: None

Description: Print out relevant information regarding state.

WAITFOR Method: **Fire;**

Return Value: None

Parameters: None

Description: Enables suspension of calling TELL method.

ASK Method: **InterruptTrigger**

Return Value: None

Parameters: None

Description: Interrupt all activities pending a **TriggerObj**.

ASK Method: **NumWaiting**

Return Value: **INTEGER**

Parameters: None

Description: Returns number of activities waiting to be triggered.

ASK Method: **ObjInit**

Return Value: None

Parameters: None

Description: Initializes internal data structures.

## TriggerObj (cont)

ASK Method: **ObjTerminate**

Return Value: None

Parameters: None

Description: Cleans up internal data structures .

ASK Method: **Release**

Return Value: None

Parameters: None

Description: Synchronous release of suspended activities.

TELL Method: **Trigger**

Return Value: None

Parameters: None

Description: Asynchronous release of suspended activities.

# Index

---

- .(dot)
  - symbol in identifier..... 32
- .dot notation..... 56
- .mod
  - file extension..... 94, 99
- A
- accumulative errors..... 21
- accuracy of representation..... 21
- ACTID..... 151, 154-155
- ACTID variable ..... 155
- actions..... 101
- activities..... 4, 141
  - concurrent..... 141
  - interrupt or stop..... 153, 155
  - order of ..... 151, 155
  - scheduled..... 142
- activities scheduled for object..... 153, 155
- activity..... 144, 153, 155, 183
  - synchronous..... 148
- activity record..... 153, 155
- ActivityList..... 153, 155
- ActivityName procedure..... 151, 155
- ActivityOwner procedure..... 151, 155
- ActivityTrace..... 153
- Actual parameter..... 84
- Add..... 158-159
- AddAfter..... 159
- AddAfter or AddBefore methods..... 159
- AddBefore..... 159
- allocate..... 47
- allocating objects..... 109
- allocation
  - dynamic ..... 57
- ancestor..... 126, 183
- AND ..... 32, 187
- animation..... 179
- anonymous type..... 63
- ANSI..... 22
- antithetic variate ..... 163
- ANYOBJ..... 56, 71, 108-114, 136, 157, 160
  - reference to fields..... 109, 114
- ANYREC..... 55, 71, 108, 111, 114
- approximation errors..... 21, 74
- arithmetic operator..... 30
- ARRAY..... 43-56, 66, 188
  - allocation ..... 59, 60
  - anonymous type..... 63
  - bounds..... 61
  - disposal ..... 60
  - dynamic..... 45
  - dynamic sizing..... 57
  - in Boolean expression..... 59
  - initialization ..... 60
  - operations..... 59
  - ragged ..... 60, 61
  - type declaration ..... 57
- array bounds..... 57
- array elements
  - referencing ..... 59
- array of fields..... 45
- ARRAY OF INTEGER..... 44
- ARRAY OF STRING..... 44
- AS..... 90, 188
  - resolve IMPORT conflict..... 90
- ASCII character set ..... 22
- ASK..... 108, 114, 117, 118, 124, 188
  - method within object..... 117
- ASK METHOD..... 104-105, 114-116
  - proper or funtion type..... 116
- ASK statement..... 115
- ASK statement
  - effect on program execution..... 115
- assignment ..... 48, 55, 64
- assignment operator..... 30
- assignment statement..... 73
- asynchronous call..... 116
- automatic compilation..... 5
- automatic initialization..... 38, 55
- B
- base type..... 56, 125, 126, 183
- base types..... 132, 136
- BEGIN ..... 11, 189
- begin end
  - Pascal ..... 70
- behavior ..... 183
- Beta distribution..... 163
- binary..... 27
- binary number system..... 20
- binary operator..... 31, 32
- binding
  - dynamic..... 183
- Binomial distribution..... 163
- BINTEGGER..... 165, 245
- block ..... 2, 9, 13, 14, 189
- blocking mechanism
  - resource..... 167

BOOLEAN..... 19, 25, 31, 40, 47  
 Boolean expression..... 40, 59, 74-81  
 Boolean literal ..... 29  
 BOOLEAN type..... 24  
 bounds..... 56, 57  
     in array dimensions..... 57  
     upper and lower in array..... 58  
 bracket..... 17  
 BREAL..... 165, 245  
 btree ..... 157  
 BTreeObj..... 158  
 built-in procedure..... 32, 205  
 built-in simple data type..... 19  
 BY..... 189  
     loop increment..... 78

C

C language interface..... 95, 99  
 C/C++ interface  
     STRING data type..... 97  
     STRING data type passed in..... 97  
 C/C++ source code  
     naming conventions..... 95  
 CALL..... 38, 85, 189  
 Cancel method..... 168  
 case label..... 76  
 case sensitive..... 12  
 CASE statement..... 76  
     type of expression in..... 76  
 CHAR..... 19, 22, 25, 40, 47  
 CHAR type..... 22, 23  
 character..... 22  
 character literal..... 28  
 Chinese character set..... 22  
 ChooseNext method  
     SimControlObj..... 151, 155  
 class..... 107, 114, 190  
     meta..... 107, 114  
 Class variables..... 107, 114  
 CLONE ..... 50, 110, 114, 205  
 Close ..... 175  
 code management..... 5  
 code sample..... 9  
 comment..... 17  
 comments  
     nested ..... 17  
 comparison of fields..... 160  
 comparisons  
     value..... 31  
 compilation  
     modular ..... 1  
     separate..... 89  
 compilation..... 5  
 component..... 183

concatenation..... 40  
 concurrent..... 143  
 concurrent activities..... 4, 141  
 conflicting field names  
     in inheritance ..... 136  
 conflicting methods..... 183  
 conformant type..... 23  
 CONST..... 36, 190  
 constant declaration ..... 35  
 Control character ..... 28  
 control structure..... 70  
 conversion  
     type..... 71  
 copy..... 50  
     dynamic data type..... 50  
 cyclic relationships..... 93, 99

D

data  
     orphaned ..... 51  
 data element..... 43, 56  
 data hiding ..... 137  
 data sharing ..... 137  
 data structure ..... 43  
     aggregate..... 52  
     ARRAY ..... 43  
     dynamic ..... 46, 184  
     fixed..... 46, 63  
     OBJECT..... 43  
     RECORD..... 43, 52  
     without identifier..... 48  
 data structures  
     building complex..... 62  
 data type..... 20  
     BOOLEAN..... 19, 24-25, 31  
     built-in simple..... 19  
     CHAR..... 19, 22, 23  
     copy of dynamic..... 50  
     definition of..... 20  
     dynamic ..... 84, 183  
     dynamic vs. fixed..... 54  
     enumeration ..... 19  
     fixed..... 47, 184  
     FIXED RECORD..... 64  
     INTEGER ..... 19, 21  
     MODSIM vs. C/C++..... 95  
     procedure..... 38  
     REAL..... 19, 21  
     simple ..... 19, 186  
     STRING..... 19, 23  
     structured ..... 19, 43, 186  
     subrange..... 19, 25  
     user defined..... 43  
     user-defined simple..... 19

- data types
    - create new ..... 36
    - dynamic ..... 47
    - replaceable ..... 111, 114
  - deallocate ..... 47
  - debugger ..... 5
  - DEC ..... 24, 206
  - decimal ..... 27
  - decimal notation ..... 27
  - declaration ..... 10, 35, 62, 104, 114
    - constant ..... 35, 36
    - method ..... 104, 114
    - object ..... 102, 104, 114
    - of method ..... 115
    - of procedures, object methods ..... 93, 99
    - procedure ..... 35, 38
    - syntax ..... 37
    - TYPE ..... 36
    - VAR ..... 35
    - variable ..... 37
  - declaring an object type ..... 102
  - DecrementResourcesBy ..... 169
  - DEFINITION module ..... 89-99, 104-105, 114
    - with IMPLEMENTATION module ..... 92, 99
  - delayed method call ..... 116
  - Delete ..... 175
  - DeleteFile ..... 176, 245
  - delimiter ..... 11, 15
    - comment ..... 17
  - derived object ..... 129, 136
  - derived type ..... 125, 126, 183
  - descendant ..... 126
  - describe the nature of the data ..... 35
  - development environment ..... 5
  - dimension
    - array ..... 56
    - index ..... 56
  - discrete-event simulation ..... 4, 141, 142
  - DISPOSE ..... 47, 51-55, 60, 101-114, 183, 206
    - of ANYOBJ ..... 110, 114
  - distribution ..... 163
  - DIV ..... 30, 31, 190
  - double precision ..... 21
  - DOWNTO ..... 190
    - loop increment ..... 78
  - DResMod.mod ..... 167
  - DStatMod.mod ..... 165
  - duplicate symbols ..... 98, 99
  - DURATION ..... 145, 191
  - DURATION in WAIT statment ..... 144, 145
  - dynamic array ..... 45
  - dynamic binding ..... 183
  - dynamic data structure ..... 46, 63
  - dynamic data structures ..... 157
  - dynamic data type ..... 47, 50, 54-55, 183
    - object ..... 101
  - dynamic memory management ..... 47
  - dynamic string ..... 23
- E
- elapse simulation time ..... 141
  - element ..... 43
  - ELSE ..... 74, 191
  - ELSIF ..... 74, 191
  - encapsulation ..... 1, 101, 137, 183
  - END WAIT ..... 144, 145
  - enumerated type ..... 24, 91
  - enumerated types
    - in OUTPUT statment ..... 173
  - enumeration ..... 19, 25, 47, 183
    - anonymous type ..... 63
  - eof ..... 177
  - equal sign ..... 30
  - Erlang distribution ..... 163
    - runtime ..... 109, 114, 251
  - errors
    - approximation ..... 74
  - evaluating expressions ..... 39
  - evaluation
    - BOOLEAN ..... 32
  - event ..... 142
  - example
    - OUTPUT and INPUT ..... 173
    - RandomObj ..... 163
  - execution order ..... 151, 155
  - ExistsFile ..... 176
  - EXIT ..... 77, 191
  - EXIT statement ..... 79, 81
  - Exponent ..... 27
  - Exponential distribution ..... 163
  - exponential notation ..... 27
  - expression ..... 35, 39, 40
    - BOOLEAN ..... 32, 40, 59
    - mathematical ..... 40
  - expressions
    - BOOLEAN ..... 32
    - evaluating ..... 39
- F
- FALSE ..... 24, 29, 192
  - FetchSeed ..... 165, 245
  - field ..... 4, 43-45, 52, 101-124, 183
    - PRIVATE ..... 107, 114, 137
  - field name ..... 52
  - fields
    - inherited ..... 125
  - FIFO ..... 157, 158
    - circumventing ..... 159

file	94, 99
does it exist	176
naming convention	94, 99
repositioning within	175
file close	175
file delete	175, 176
file end flag	177
file I/O	
random access	64
file name extension	94, 99
file open	175
file read flag	177
file size	177
filename	95, 99
constraints	95, 99
FileSize	177, 245
Fire	150, 155
First	158
FIXED ARRAY	46, 63, 66, 83, 84
assignment	64
copy	64
each dimension	65
IN or INOUT	65
ragged not allowed	66
FIXED ARRAY type	64
fixed data structure	46, 63
fixed data type	47, 48, 54-55, 63, 184
FIXED RECORD	46, 63, 66
declared as explicit type	64
FIXED RECORD type	64
FIXED RECORDS	63
flag	24
floating point number	21
FOR in WAIT statment	144, 145
FOR statement	78, 79
FOREACH	79, 192
with empty group	80
Formal parameter	84
formal parameter qualifier	84
format string	174
formatstring	174
FORWARD	87, 193
free formatted I/O	173
function	83
MODSIM	32
function method	104, 114, 184
Function procedure	83, 86, 184
function result type	86

G

Gamma distribution	163
generic type	55, 56
GetResource	167
GetResource method	168

GetResource request method	168
Give	167, 168
Give method	168
glossary	183
GOTO statement	73
Graphic Editor	179
Graphics	179
group	111, 114, 184
LIFO and FIFO	157
number times has changed	160
queue	157
QueueObj	158
ranked	157
removing from	159
stack	157
statistical	160
group based on user defined Rank	158
group ordering	157
groups	157
GrpMod module	157, 160

H

hanging reference	52
help system	5
hexadecimal	27
hexadecimal address of object	173
Hierarchical type	102
hierarchy	101
object	126
HIGH	61, 206
histogram	245
for ResourceObj	170

I

I/O	173
Icon	179
identifier	11, 20, 24, 47, 106, 114
length	98-99
IF	16, 193
IF statement	69, 73, 76
multiple choice	73
IMPLEMENTATION module	89-99, 104-114
import	89, 94, 99, 105, 114, 193
conflict	90
enumerated type	91
IMPORT statement	90
IN	65, 84, 147, 194
IN parameter	64, 147
INC	24, 206
Includes	158
increment of loop variable	79
IncrementResourcesBy	169
independent compilation	1
index	43, 44, 56

array..... 44  
 inherit ..... 128  
 Inheritance ..... 101, 125, 184  
   base type..... 125  
   conflicting field names..... 136  
   derived type..... 125  
   multiple..... 125, 132, 136  
 inherited..... 125, 130, 135, 136, 194  
   class variables and methods..... 107, 114  
   ObjInit..... 131, 136  
   order of method execution..... 131, 136  
 inherited behaviors..... 130, 136  
 inherited call..... 131, 136  
   qualified..... 185  
 initialization..... 38, 135, 136  
   automatic ..... 38  
   BOOLEAN..... 38  
   CHAR..... 38  
   enumeration variable..... 38  
   INTEGER..... 38  
   ordinal types..... 38  
   REAL..... 38  
   STRING..... 38  
 initialize..... 60  
 initialize modules..... 89  
 INOUT..... 84, 97-99, 116, 147, 194, 245  
 INPUT ..... 3, 83, 173, 207, 245  
 input/output  
   IOMod module..... 173  
   StreamObj..... 173  
 instance..... 143, 184  
 INTEGER.. 2, 3, 10, 14-31, 38-50, 59-66, 96-99  
 integer literal..... 26  
 INTEGER type..... 21  
 interrupt..... 141, 153-155, 167-168, 194, 245  
 InterruptTrigger method  
   TriggerObj..... 150, 155  
 invoke..... 184  
 invoking a method..... 115  
 invoking a procedure..... 85  
 IOMod module..... 173  
 ioResult..... 177  
 ISO 646 character set..... 22  
 iterate through a group..... 157

**J**

Japanese character set ..... 22

**K**

kanji..... 22  
 katakana..... 22  
 key  
   STRING..... 158  
 Key method..... 158

**L**

Last..... 158  
 layout..... 9  
 length of identifier..... 13, 98-99  
 Lexical components..... 9  
 library module..... 5  
 library modules..... 89  
 LIFO ..... 157-158  
 link  
   limits ..... 98-99  
 link records ..... 46  
 linked list ..... 46, 54  
 list  
   linked ..... 54  
 literal..... 11, 26  
   boolean..... 29  
   character..... 28  
   decimal..... 27  
   enumerated type..... 29  
   hexadecimal..... 27  
   integer..... 26  
   real ..... 27  
   string ..... 28  
 literal constant..... 20  
 locality ..... 9  
 logical operator..... 32  
 LogNormal distribution..... 163  
 long integer..... 21  
 loop increment  
   BY..... 78  
   DOWNTO..... 78  
   TO..... 78  
 loop increments..... 78  
 loop iteration..... 80  
   REVERSED..... 80  
 loop statement..... 77, 81  
 loop variable..... 78  
   increment ..... 79  
 LOW ..... 61, 207

**M**

MAIN MODULE..... 9, 89, 92, 104, 114  
 MAX..... 12, 22, 24, 207  
 MAXOF..... 83, 207  
 MaxResources..... 169  
 Mean..... 160  
 member ..... 184  
 memory  
   running out of..... 51  
 memory leak..... 51  
 memory locations..... 55  
 memory management..... 47  
   dynamic..... 47

message..... 115, 184  
 Message passing..... 101  
*meta* class ..... 107, 114  
 method..... 1, 101, 102, 115, 183-184, 196  
     asynchronous..... 145, 186  
     class..... 107, 114  
     declaration ..... 104, 114  
     function ..... 104, 114, 184  
     INHERITED..... 130, 136  
     invoking..... 115  
     not returning a result..... 104, 114  
     OVERRIDE..... 107, 114, 125, 129, 136  
     parameters ..... 104, 114  
     pause to wait for condition..... 150, 155  
     PRIVATE..... 107, 114, 137  
     proper ..... 104, 114, 145, 185  
     returning a result..... 104, 114  
     TELL..... 186  
     time elapsing..... 186  
 METHOD declaration..... 102, 104, 114, 115  
 METHOD heading..... 102, 114  
 MIN..... 22, 24, 207  
 MOD..... 30-31, 196  
 ModInit ..... 89, 94, 99  
     order of execution..... 94, 99  
 modular structure..... 89  
 module..... 1, 5, 89, 196  
 modulus operator..... 30, 31  
 MONITORED INTEGER field..... 161  
 MONITORING..... 120, 124  
 Multi-dimension array..... 58  
 multiple choice IF statement..... 73  
 multiple inheritance..... 125, 132, 136  
 Multiple Process Activities..... 151

N

naming convention..... 94, 99  
 naming conventions  
     C source code..... 95, 99  
 nested comment..... 17  
 NEW ..... 47, 51-60, 101-110, 114, 183, 196, 208  
 new capabilities..... 101  
 new data types..... 36  
 newline character..... 173  
 Next..... 158, 161  
 NILARRAY..... 60, 105, 114, 196  
 NILOBJ..... 56, 105-106, 114, 159, 161, 196  
 NILREC ..... 52, 55-56, 105, 114, 197  
 NONMODSIM..... 95-99, 197  
 Normal distribution..... 163  
 NOT ..... 32, 197  
 notation ..... 56  
 null string..... 38  
 number

    random..... 163  
 number field..... 161  
 number times group has changed..... 160  
 numerical accuracy..... 21

O

ObjClone..... 110, 114  
     override..... 110, 114  
 object..... 1, 4, 43-51, 59, 62, 101, 184, 197  
     activity ..... 144  
     array of..... 101  
     CLONE..... 110, 114  
     declaration ..... 104, 114  
     declaration/initialization..... 105, 114  
     derived..... 129, 136  
     DISPOSE..... 110, 114  
     dynamic creation..... 105, 114  
     Encapsulation of data & code..... 101  
     field..... 101  
     field with reference variable..... 107, 114  
     fields ..... 118, 124  
     hierarchy ..... 101  
     in groups..... 157  
     inheritance ..... 125  
     inserting first in group..... 159  
     inserting in group..... 158, 159  
     instance..... 102, 106, 114  
     interaction..... 143  
     METHOD declaration..... 102, 104, 114  
     METHOD heading..... 102, 114  
     modification of fields..... 118, 124  
     NEW..... 106, 114  
     new data type capabilities..... 101  
     ObjClone..... 110, 114  
     ObjInit..... 110, 114, 124  
     ObjInit in multiple inheritance ..... 135, 136  
     ObjTerminate..... 110, 114  
     Polymorphism..... 101, 130, 136  
     properties ..... 101  
     PROTO ..... 157  
     reference as SELF ..... 118  
     removing from group..... 159  
     routines/methods..... 101  
     statistical data collection..... 160  
     statistical monitor..... 165  
     type declaration..... 102, 105, 114  
 object declaration..... 104, 114  
 object hierarchy..... 126  
 object instance..... 102, 109, 114  
     concurrent..... 143  
 OBJECT type..... 101, 183  
 object type declaration..... 102, 104, 114  
 objects queueing..... 157  
 ObjInit ..... 110, 114, 131, 136

- of RandomObj..... 163
- override..... 131, 136
- ObjInit method..... 135, 136
- ObjPrint..... 173
- ObjTerminate..... 110, 114
- ON INTERRUPT..... 144, 148
- ON INTERRUPT clause..... 153, 155, 168
- ON INTERRUPT in WAIT statement.. 144, 145
- Open file..... 175
- operand..... 31, 175, 183
- operation..... 183
- operations
  - array..... 59
- operator..... 11, 29, 30
  - arithmetic..... 30
  - assignment..... 30
  - binary..... 30, 31, 32
  - logical..... 32
  - modulus ..... 30-31
  - precedence..... 39
  - relational..... 31
  - unary..... 30-32
- operator precedence..... 39
- OR..... 32, 197
- order ..... 151, 155
- ordering ..... 10
  - of groups..... 157
- ordinal type..... 19, 25, 56, 184
- orphaned data..... 51
- OTHERWISE..... 76, 198
- OUT..... 84, 116, 147, 198
- OUT parameter ..... 97, 99
- OUTPUT..... 3, 83, 173, 174, 209, 245
- OUTPUT procedure..... 85
- OUTPUTing Objects..... 173
- OVERRIDE..... 107, 114, 125, 129, 136, 198
- override ObjInit..... 131, 136

## P

- parameter
  - actual ..... 84
  - copy of..... 84
  - formal ..... 84
  - formal qualified..... 84
  - method..... 104, 114
  - passing to C..... 97, 99
- parameter list ..... 83
  - empty..... 88
- parameters
  - OUT or INOUT..... 85
- parentheses ..... 39
- pass by reference..... 84, 185
- pass by value..... 185
- pause to wait for condition..... 150, 155

- pending list..... 142, 153, 155, 245
- PendingResources..... 169
- pi..... 90, 190, 245
- pointer..... 48
  - lost ..... 51
- Poisson distribution..... 163
- polymorphism..... 1, 4, 101, 130, 136
- precedence..... 35
- precedence rule..... 39
- Presentation Graphics..... 179
- Prev..... 158, 161
- PRINT..... 174, 198, 209
- priority of resource request..... 168
- PriorityGive..... 167-168
- PriorityGive method..... 168
- PRIVATE..... 107, 114, 137, 198
- private property..... 185
- procedure..... 55, 83, 86, 198
  - executed before program..... 94, 99
  - function ..... 83, 86, 184
  - ModInit ..... 94, 99
  - MODSIM ..... 32
  - proper..... 83
  - recursive..... 83
  - used before defined..... 87
- procedure block..... 86
- procedure declaration..... 35, 38
- PROCEDURE heading..... 86
- procedure type..... 38
- procedures..... 85, 245
  - built-in..... 85
  - user defined..... 38, 85
- Process..... 2, 4, 142-44, 185
- program execution
  - asynchronous..... 116-117
- program structure ..... 9
- project management..... 5
- proper method..... 104, 114, 185
- proper methods..... 145
- Proper procedure..... 83, 185
- properties ..... 101, 185
- PROTO OBJECT..... 114, 167
  - replaceable types..... 113-114
- pseudo-random number..... 163
- public property..... 185

## Q

- queue..... 157
- QueueObj..... 158, 162

## R

- ragged array ..... 60, 61
- RandMod module..... 163
- Random..... 165, 245

random number .....	163	resource return .....	169
pseudo.....	163	ResourceObj.....	167-170
reproducible.....	164	SetAllocStats.....	170
random number generator		ResourceObj field MaxResources.....	169
non object-oriented.....	165	ResourceObj field PendingResources.....	169
period of.....	164	ResourceObj field Resources.....	169
Random variable.....	163	ResourceObj SetPendStats.....	170
RandomObj.....	163	ResourceObj statistics.....	167-170
Rank method.....	158-160	ResourceObj TakeBack method.....	169
ranked .....	157	resources	
RankedObj.....	158-159	changing set of.....	169
read character from console.....	177, 245	pending requests.....	169
ReadChar.....	176	resources available.....	169
ReadInt.....	176	Resources field.....	169
ReadKey.....	177, 245	resources requested.....	167
ReadLine .....	176	RETURN.....	86, 199
ReadReal.....	176	RETURN statement.....	81, 86
ReadString.....	176	REVERSED.....	80, 199
REAL.....	19-21, 30, 38, 40, 47, 59, 63, 74	round off errors.....	21
real number .....	20-21, 27	routine .....	83, 184, 185
REAL type.....	21	runtime error	
REAL values		resource request.....	167
exact.....	74	resource return .....	169
RECORD.....	43-52, 59, 66, 185, 199	transfer of resource ownership.....	169
dynamic.....	46		
RECORD type declaration.....	52, 102	S	
recursive procedure.....	83	sample MODSIM code.....	9
recursive TERMINATE.....	154, 155	sample of MODSIM code.....	2
reference		scalar type.....	19, 186
hanging.....	52	scheduled activities.....	142
pass by.....	84, 185	scientific notation.....	27
reference type.....	105, 114, 185	scope.....	9, 13-14, 102, 105, 114, 137
reference value.....	106, 114	scope of identifier.....	15
reference variable. 48-55, 84, 102-114, 138, 185		scope of reference.....	105, 114
relational operator.....	31	seed.....	163, 245
Remove.....	158	SELF.....	118, 186, 200
Remove method.....	159	semicolon.....	16
RemoveThis.....	158	Separate compilation.....	1, 5
RemoveThis method.....	158	separator .....	11, 16
rename.....	90	sequence of execution.....	73
REPEAT statement.....	78	SetAllocHistogram.....	170
Replaceable types.....	111-112, 114	SetAllocStats.....	170
Reserved word.....	12, 187	SetPendingHistogram.....	170
resource .....	167	SetPendStats	
Create method.....	169	of ResourceObj.....	170
histogram set up.....	170	SetSeed.....	165
preemption.....	168	SetTieBreaking method	
release.....	167	SimControlObj.....	151, 155
revoking request for.....	169	SetTimeAdvance	
transfer ownership .....	169	SimControlObj.....	153, 155
resource acquisition.....	167	shared variable.....	186
resource blocking mechanism.....	167	short-circuit evaluation.....	32, 40
resource request		SimControlObj.....	151, 155
priority of.....	168	SIMDRAW.....	5

- simple data type..... 19, 43, 186
  - SIMSCRIPT II.5..... 4, 164
  - SimTime()..... 141, 155, 251
  - simulation time..... 4, 141-144, 251
    - elapsing..... 141-147, 167, 183
    - notification of advance..... 151, 155
    - passage of..... 142
    - passing..... 116
    - unit conversions..... 141
    - units of..... 141
    - update notification..... 153, 155
  - SINTEGER..... 165, 245
  - source code..... 95, 99
  - source file..... 94, 99
  - SPRINT..... 174, 201, 210
  - SREAL..... 165, 245
  - stack..... 157
  - StackObj..... 158, 159
  - StackObj type..... 159
  - standard input..... 173
  - standard output..... 173
  - standard procedure..... 12
  - state..... 101
  - Statement..... 69
  - statement sequence..... 70
  - statistical data..... 160
  - statistical distribution..... 163
  - statistical groups..... 161
  - statistical monitor objects..... 165
  - statistics
    - on ResourceObj..... 167, 170
    - pre-defined types of variables..... 165
  - StatQueueObj..... 160, 339
  - StatRankedObj..... 160
  - StatStackObj..... 160
  - StdDev..... 160
  - stream
    - random number..... 163
  - StreamObj..... 173, 175, 338
  - StreamObj for input/output..... 173
  - STRING..... 19, 22, 40, 47
    - characteristics of..... 23
    - generation of format..... 174
  - STRING key..... 158
  - String literal..... 28
  - STRING type..... 23, 97, 99
  - strongly typed..... 3, 9, 30-31, 186
  - structured data type..... 19, 43, 186
  - subblock..... 87
  - subprogram..... 83
  - subrange..... 19, 25, 56, 58, 70
  - subrange type..... 25
  - sub-routine..... 185
  - subroutine..... 83
  - substitute..... 113-114
  - substitution..... 111-114
  - switch..... 24
  - symbols..... 15
- T
- TakeBack method..... 167, 169
  - TELL..... 117, 141-146, 201
    - method within object..... 117
  - TELL METHOD.... 104, 114-116, 154-155, 186
  - TERMINATE..... 149, 154-155, 201
  - TERMINATE statement..... 81
  - time..... 141
    - units of..... 141
  - TimeAdvance method
    - SimControlObj..... 153, 155
  - TimedGive method..... 167, 168
    - for resource request..... 168
  - time-elapsing method..... 186
  - TO..... 117, 201
    - loop increment..... 78
  - token..... 9, 11, 16
  - Transfer..... 168
  - Transfer method..... 167
  - Triangular distribution..... 163
  - Trigger..... 150, 155
  - Trigger Object..... 150, 155
  - TriggerObj..... 150, 155
  - TRUE..... 24, 29, 202
  - TSINTEGER..... 165, 245
  - TSREAL..... 165, 245
  - type..... 2, 20, 36, 62, 95, 99, 202
    - ordinal..... 184
    - reference..... 185
    - scalar..... 186
    - underlying..... 186
  - type casting..... 72
  - type checking..... 55, 109, 114
    - circumvent with ANYOBJ..... 109, 114
  - type compatibility..... 70
  - type conversion..... 3, 71
  - TYPE declaration..... 36
    - anonymous..... 63
    - array..... 57
    - PROTO OBJECT..... 111, 114
    - RECORD..... 52, 102
- U
- unary operator..... 32
  - undefined..... 52
  - underlying type..... 126, 186
  - UniformInt distribution..... 163
  - UniformReal distribution..... 163
  - units of time..... 141

UNTIL.....78, 202  
 update simulation time..... 153, 155  
 user-defined simple data type..... 19  
 user-defined type..... 24

V

VAL .....24, 211  
 pass by..... 185  
 value comparison..... 31  
 VAR.....62, 202  
 variable.....20, 105, 114  
   class.....107, 114  
   global in module.....105, 114  
   local to method.....105, 114  
   reference.....48, 52, 84, 102, 105, 114, 185  
   refers to DISPOSEd data structure..... 51  
   shared ..... 186  
 Variable declaration..... 37  
 Variance ..... 161  
 visibility..... 9

W

WAIT ..... 4, 144, 154-155, 186, 202  
 WAIT DURATION..... 145

WAIT FOR..... 148-155, 167  
   TERMINATE..... 149, 155  
 WAIT FOR statement..... 116  
 WAIT statement..... 81  
   syntax..... 144  
 WAIT statements  
   multiple..... 154-155  
 WAITFOR..... 141-144, 203  
 WAITFOR METHOD.. 114-116, 147, 155, 186  
 Weibull distribution..... 163  
 WHEN..... 76, 203  
 WHILE statement..... 77  
 whole number..... 21  
 Write line..... 176  
 WriteChar ..... 176  
 WriteHex ..... 176  
 WriteInt ..... 176  
 WriteLn ..... 176  
 WriteReal..... 176  
 WriteString ..... 176  
 WtdMean..... 161  
 WtdStdDev..... 161  
 WtdVariance..... 161

