SIMSCRIPT III®

Users Manual

CACI Products Company

If there are questions regarding the use or availability of this product, please contact CACI at any of the following addresses:

For product Information contact:

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

CACI Worldwide Headquarters
1100 North Glebe Road
Arlington, Virginia 22201
Telephone (703) 841-7800
www.caci.com

For technical support contact:

Manager of Technical Support
CACI Products Company
1455 Frazee Road #700
San Diego, CA 92108

simscript@caci.com

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

SIMSCRIPT III® and SIMSCRIPT II.5® are a registered trademarks and service mark of CACI Products Company.

TABLE OF CONTENTS

FIGURES

PREFACE

This document contains information on the use of CACI's SIMSCRIPT III compiler for developing simulation models. Development can be done either using SIMSCRIPT III Development Studio (Simstudio) or Command-line interface.

CACI publishes Manuals that describe the SIMSCRIPT III Programming Language, SIMSCRIPT III Graphics and SIMSCRIPT III SimStudio. All documentation is available on SIMSCRIPT WEB site http://www.simscript.com

- *SIMSCRIPT III User's Manual* – this manual - a detailed description of the SIMSCRIPT III development environment: usage of SIMSCRIPT III Compiler and the symbolic debugger from the SIMSCRIPT Development studio - Simstudio, and from the Command-line interface.

- *SIMSCRIPT III Programming Manual* - A short description of the programming language and a set of programming examples.

- *SIMSCRIPT III Reference Manual* - A complete description of the SIMSCRIPT III programming language constructs in alphabetic order. Graphics constructs are described in SIMSCRIPT III Graphics Manual.

- *SIMSCRIPT III Graphics Manual* — A detailed description of the presentation graphics and animation environment for SIMSCRIPT III

Series of Manuals and text books for SIMSCRIPT II.5 language and SIMSCRIPT II.5 Simulation Graphics, Development environment, Data Base connectivity, Combined Discrete-Continuous Simulation, etc

- *SIMSCRIPT II.5 Simulation Graphics User's Manual* — A detailed description of the presentation graphics and animation environment for SIMSCRIPT II.5

- *SIMSCRIPT II.5 Data Base Connectivity (SDBC) User's Manual* — A description of the SIMSCRIPT II.5 API for Data Base connectivity using ODBC

- *SIMSCRIPT II.5 Operating System Interface* — A description of the SIMSCRIPT II.5 APIs for Operating System Services

- *Introduction to Combined Discrete-Continuous Simulation using SIMSCRIPT II.5* — A description of SIMSCRIPT II.5 unique capability to model combined discrete-continuous simulations.

- *SIMSCRIPT II.5 Programming Language* — A description of the programming techniques used in SIMSCRIPT II.5.

- *SIMSCRIPT II.5 Reference Handbook* — A complete description of the

SIMSCRIPT II.5 programming language, without graphics constructs.

• *Introduction to Simulation using SIMSCRIPT II.5* — A book: An introduction to simulation with several simple SIMSCRIPT II.5 examples.

• *Building Simulation Models with SIMSCRIPT II.5* —A book: An introduction to building simulation models with SIMSCRIPT II.5 with examples.

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

Free Trial Offer

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

Training Courses

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

San Diego, California
Washington, D.C.

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

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

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

Introduction

As an aid to making important decisions, the use of computer simulation has grown at an astonishing rate since its introduction. Simulation is now used in manufacturing, military, nuclear applications, models relating to urban growth, hydroelectric planning, transportation systems, election redistricting, cancer and tuberculosis studies, hospital planning, communications, and multi-computer networks. SIMSCRIPT III is suitable for building high-fidelity simulation models especially very large models. SIMSCRIPT III modularity and object-orientated concepts simplify development, maintenance and code reuse.

SIMSCRIPT III is a language designed specifically for simulation. It is the most efficient and effective program development technique for simulation, due to the following properties:

• **Portability**. SIMSCRIPT III development environment, which includes Development Studio, language compiler and Graphical systems are available on the various computer systems. This facilitates the development of general-purpose models and simulation applications that can be moved easily from one site to another and from one organization to another.

• **Appropriate Constructs.** SIMSCRIPT III provides constructs designed especially for simulation (e.g., classes, objects, object attributes, processes, resources, events, entities, and sets). These constructs make it easier to formulate a simulation model. Implementation of the simulation program is also quicker because these powerful tools do not have to be invented anew.

• **Self-Documenting Language.** Applications developed using the SIMSCRIPT III language is characteristically easy to read and understand. The language encourages this because it is oriented toward the kinds of problems being solved rather than the machines being used as tools. The very high-level language features of SIMSCRIPT III were designed to make it possible to manage a complex simulation models.

• **Error Detection.** SIMSCRIPT III performs a number of error checks that help to assure that a simulation model is running correctly. Powerful inline symbolic debugger speeds up run-time analysis of model behavior.

When an error in a run is detected, model enters SIMSCRIPT III symbolic debugger, which allows program status investigation, which includes the names and values of variables, system status, and other valuable information. This reduces the time spent in developing and testing programs.

• **Statistical Tools.** Along with the mathematical and statistical functions most often used in simulation (exponential functions, random number generators, and so on), SIMSCRIPT III includes the **accumulate** and **tally** statements that allow the model builder to collect statistics on key variables in his model.

• **Simulation Graphics.** Brings interactive animated and display graphics to the

SIMSCRIPT III models. Graphical objects can be easily added to the program providing automatic animation and information display. Input/ Output dialog boxes, menu bars, pallets can easily be added to the model providing elegant and functional Graphical User Interfaces.

• **Data Base Connectivity.** Provides SIMSCRIPT III Application Program Interfaces (API's) to the major databases available on the market: Microsoft Access, SQL Server Oracle, IBM DB2 and IBM Informix.

• **Operating System Interface.** Provides SIMSCRIPT III Application Program Interfaces (API's) to Operating System Services facilitating portable models across all SIMSCRIPT III supported computer platforms.

• **Open System.** SIMSCRIPT III provides possibility to call non-simscript routines/functions from a SIMSCRIPT model. This facilitates usage of libraries written in C/C++ , Java or FORTRAN from SIMSCRIPT models.

• **Complete Methodology.** The SIMSCRIPT III approach to simulation model development provides the complete set of capabilities needed to develop a simulation model. A simulation model developed in the SIMSCRIPT III programming language is readable by the analyst familiar with the system under study.

• **Support.** CACI provides SIMSCRIPT III software, documentation, training and technical support. Model development services are also available from CACI.

1. Developing Simulation Models with Simstudio

Developing a SIMSCRIPT III model typically involves the following steps:

1. Preparing one or more SIMSCRIPT III source files using a text editor.
2. Preparing graphical elements: Icons, Graphs, Dialog boxes, Menubars, etc
3. Building the model (creating the executable file), checking for compilation or linking errors
4. Editing and re-building the model, as needed, until there are no errors.
5. Executing the model
6. Debugging the model. In case of errors during execution, the model should be built with the debugging option, and executed with the interactive SIMSCRIPT III symbolic debugger, to examine the state of the model and find the cause of the error.

This development process can be done in the following two ways:

1. Using SIMSCRIPT III Development Studio – Simstudio or
2. Using Command-line interface from cmd window.

Simstudio is an easy to use, user friendly integrated programming development environment. It is the Graphical User Interface (GUI) to the SIMSCRIPT III compiler, syntax color coded text editor, graphical editors, automatic project builder and help system. In Simstudio, editing source files, compiling and linking model executable is controlled automatically for optimal efficiency. Simstudio provides the most commonly used compiler switches and link options. It will be explained in detail in Chapter 1.1

Command-line interface can be used from a “cmd” window on MS Windows or a shell terminal window on Linux. It is very convenient for users who need more control over compilation and link phases and like to make use of make files and scripts. You can use ed4sim SIMSCRIPT editor or your own favorite text editor such as “edit”, “vi”, etc. to create SIMSCRIPT source files. To create graphical (either display or user input) elements for your model, use the Simstudio graphical editors. Here you can make icons either by hand or imported from JPG files. These can then be used for presenting dynamic objects or a realistic background. CACI provides a set of commands for compiling and linking graphical and non-graphical models like: simc, simld, simgld etc. These commands are explained in full detail in Chapter 2. It also contains description of all available compiler switches.

1.1 Simstudio Overview

SIMSCRIPT III Development Studio helps you to organize your model as a **project** which can be built automatically using menu options.

When you start a new model development you have to create a new project, add source files, add graphical elements and define how you want your project to be built. After that, you can build and execute your model.

For a new project you will define the name of your project and directory where it will be located. In the project directory a **project_name.sp** file will be created to hold model information. Three subdirectories will be created: **sources, executable** and **temp**.

> **sources** – will hold all the source files of your project. You can keep all source files in one directory or organize them as a hierarchical structure of subdirectories. If model is organized as a set of modules (subsystems), please see recommended source code organization section 1.3.3
>
> **executable**– will hold model executable **project_name.exe** and **graphics.sg2** file which holds graphical elements used during model execution. Input data files necessary for model execution should also be placed in this directory.
>
> **temp**– will hold object files necessary for building the model and other temporary files. The contents of this directory are not important to developers.

This project directory structure helps you during development and deployment of your model. The subdirectory "sources" contains the current version of the model source code, while "executable" contains all files necessary to run the model.

Simstudio consists of a Menubar, Toolbar and three windows. The Project window is on the left, Editor window on the right and Status window at the bottom.

Menu bar options: File, Project, Options, Window and Help, facilitate creating a new project, opening an existing project setting project options and building and executing the model.

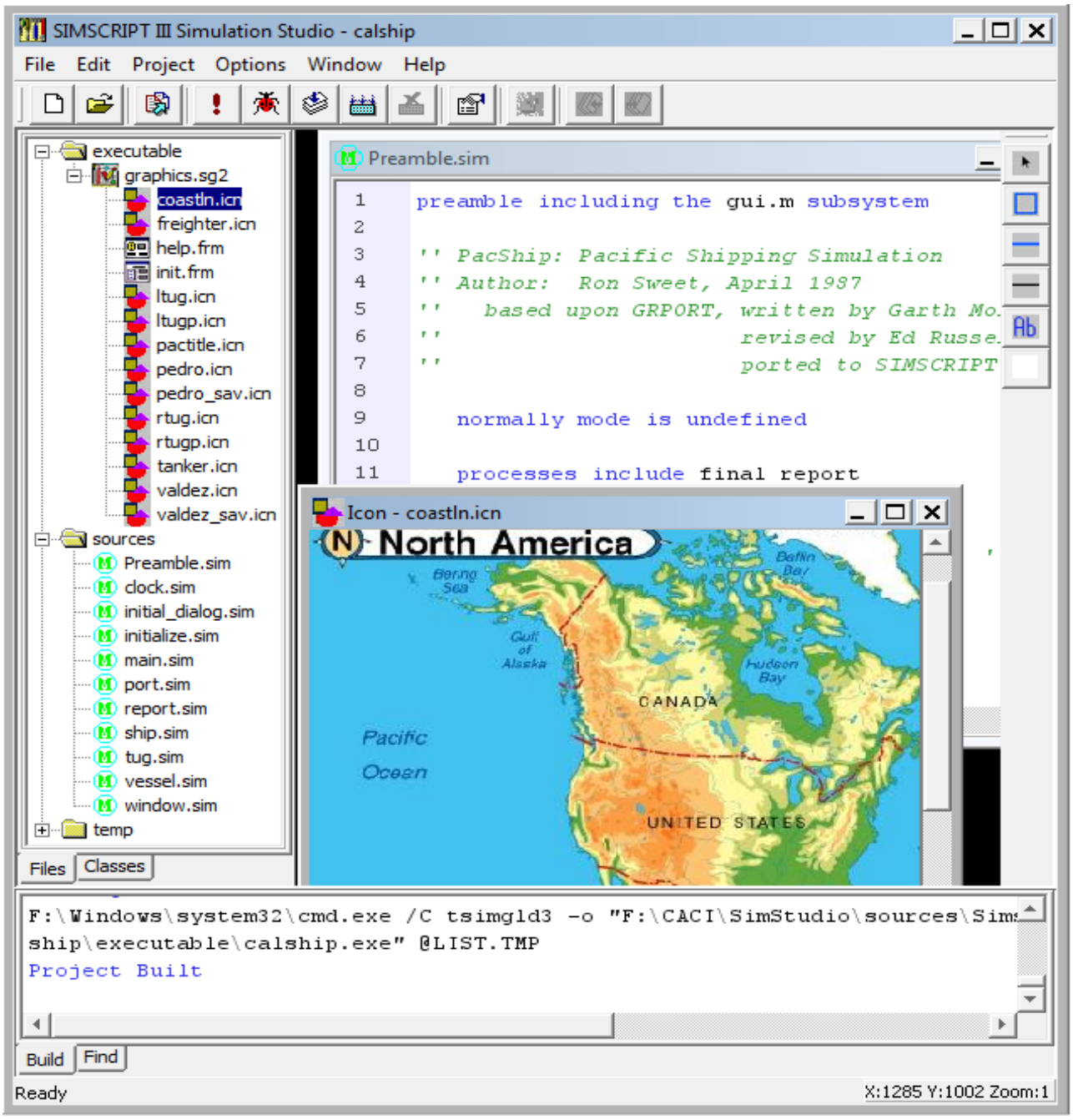

Figure 1-1: Simstudio with source and graphics windows

The project window displays the project tree with current project subdirectories: sources, executable and temp. The editor window contains windows for text and graphical editing. The status window displays messages during project build and execution.

The project tree is composed of source code files with the extension '.sim' in the directory sources. The graphics.sg2 file contains the following graphical elements: icons with extension '.icn', forms with extension '.frm' , and graphs with extension 'grf'. "graphics.sg2" can be found in the directory "executable".

Simstudio incorporates the SIMSCRIPT III syntax color coded text editor for creating/editing source files and graphical editors for creating/editing: icons, graphs, dialogs, menus and palettes. When you open a text file with extension ".sim", all necessary text editing menus and tool bars will appear. The same applies to graphical editors.

The following sections will describe how to create projects, add source code and graphical elements, and build and execute the model.

1.2 Creating a New Project

To create a new project, use the **Project->New** menu option. The dialog box **Create New Project** will appear.

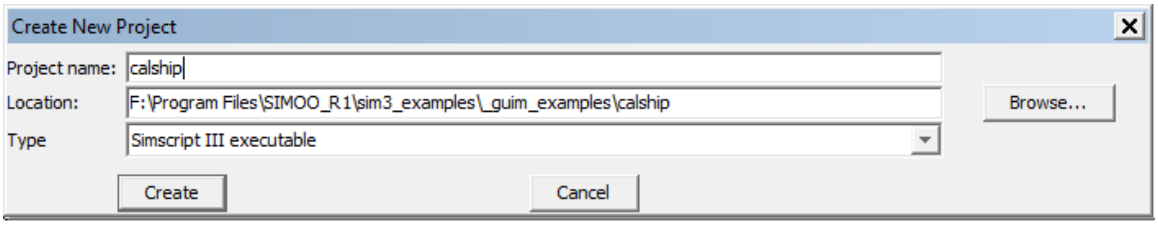

Figure 1-2: Create new project dialog.

Type in a name for the project. You can click on the **Browse…** button to help select a directory for the project. Select the type of project to create in the drop-down box.

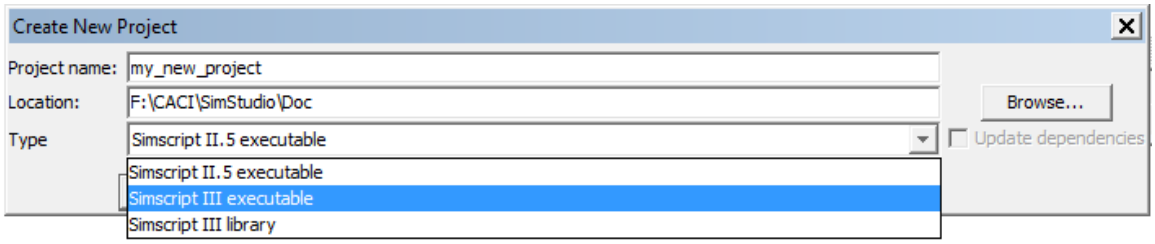

Figure 1-3: The project type dropdown box.

Currently you can create the following types of projects:

Simscript II.5 executable – For supporting legacy programs.
Simscript III executable – Create an object oriented SIMSCRIPT III model.
Simscript III library – Create a runtime library that can be linked into another model.

The new project will be created along with the following project directories: **executable**, **sources** and **temp.** These will appear in the project window. An empty **graphics.sg2** file will be created in the executable directory to hold graphical elements. A ".sp" file named after the project and will be created in the project directory to hold project information.

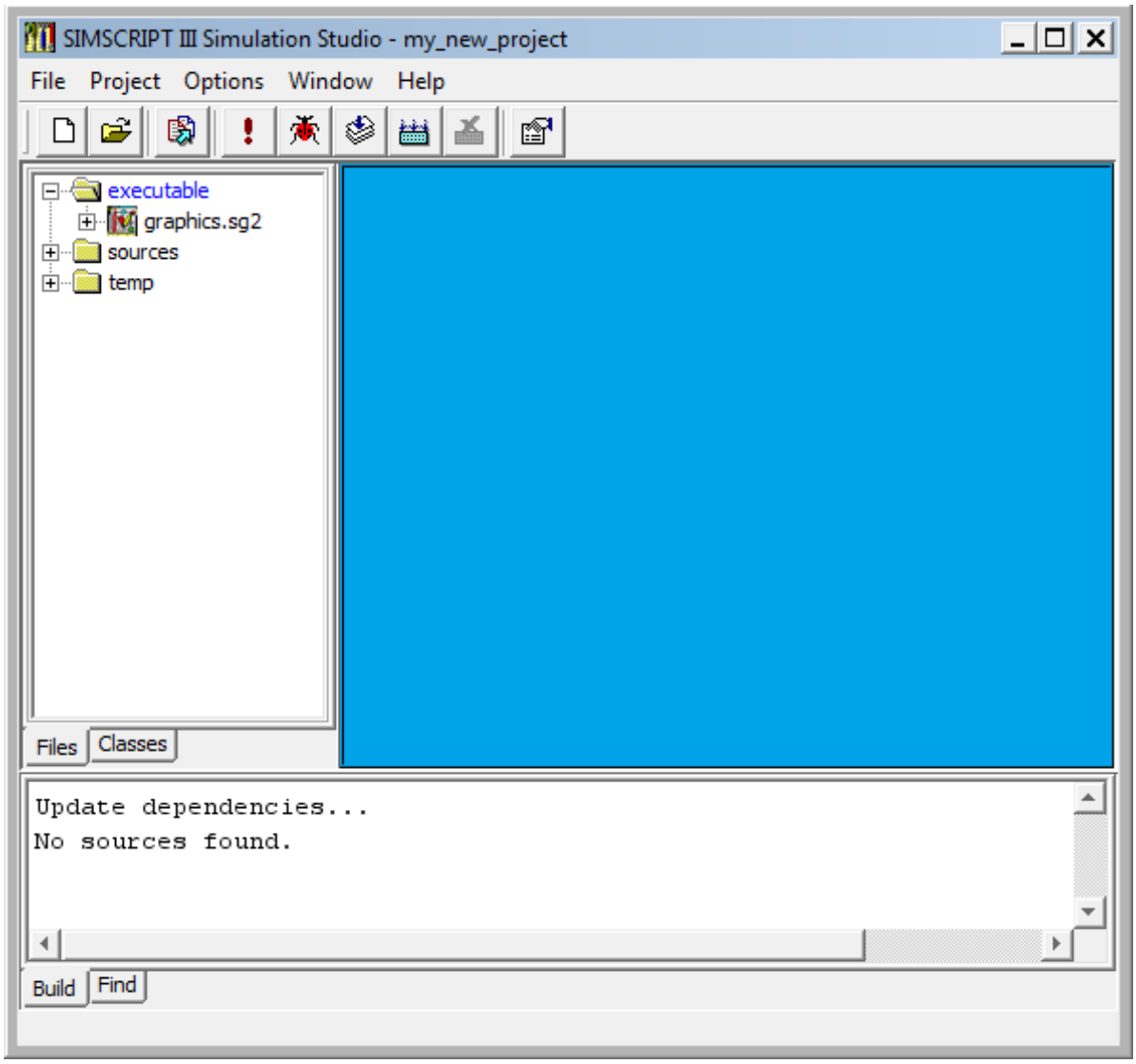

Figure 1-4: Project tree

1.3 Adding Source Code to a Project

Source code for projects is stored by default in the directory sources. You can create a new text '.sim' file, add individual directories and files or add the whole subdirectory with multiple sub-directories to your project.

1.3.1 Creating a New File with the Text Editor

Use **File->New** to open an untitled text window. Type in the text and **use File->Save As** to save it in the directory **sources**. The new file will appear in the project tree in the project window and will be saved on the disk.

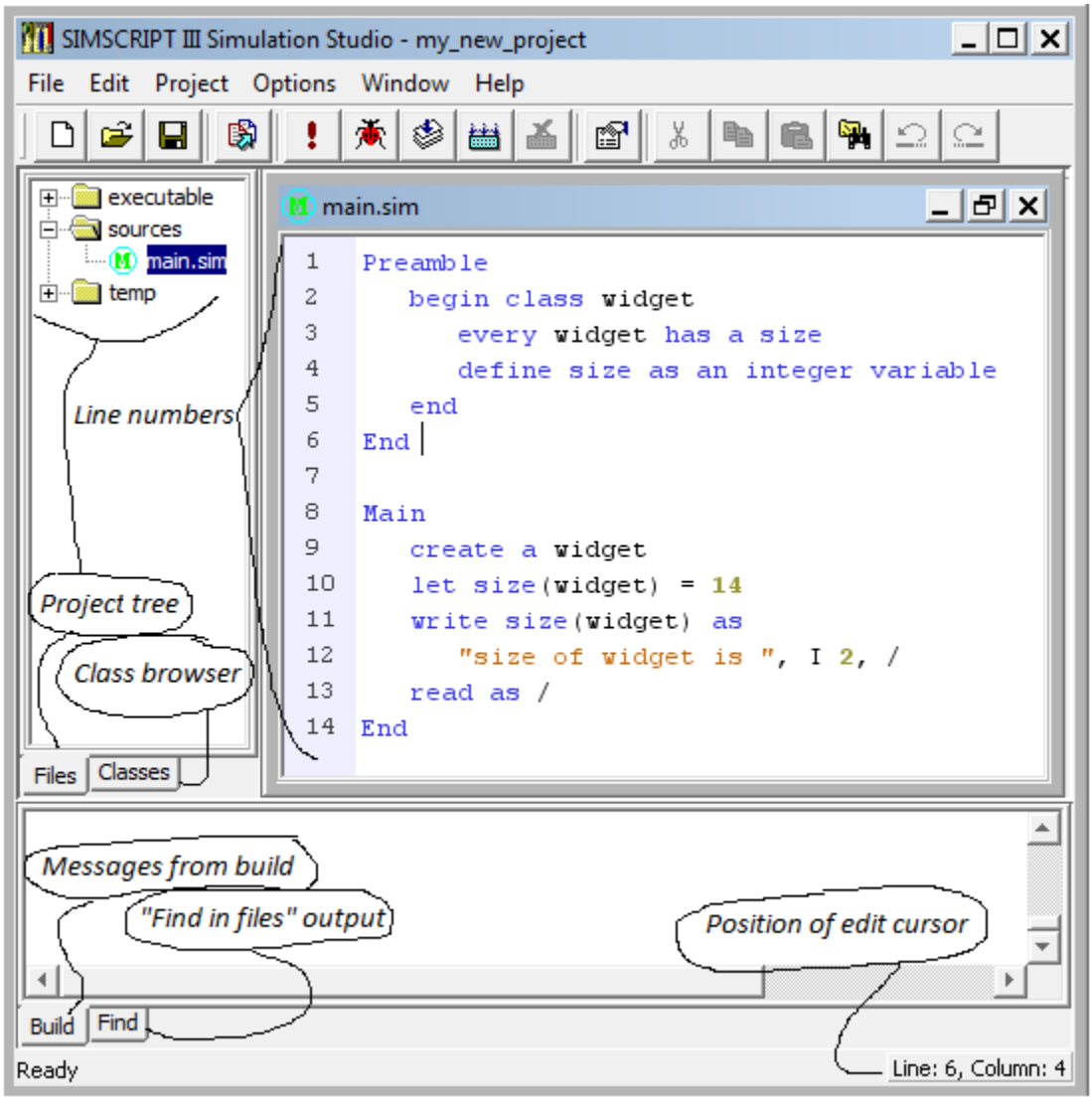

Figure 1-5: Creating a new source file

To open this file again, double-click on its name in the project tree. When you open a text file for editing the menu option **Edit** and the **text editor toolbar** will contain all necessary options for text editing.

You can open or delete a file from the project tree. Right mouse click on the source file name in the project tree. This will open a pop-up menu with the options **open** or **delete**. You can open the file for editing or you can delete it from the project and the disk.

1.3.2 Adding a Directory or a File Using Project Window

To add a new directory to the directory **sources**, right-click on its name in the tree. It will bring a pop-up menu with options: **find in directory, add files**, **new folder** and **delete**. Chose **new folder**.

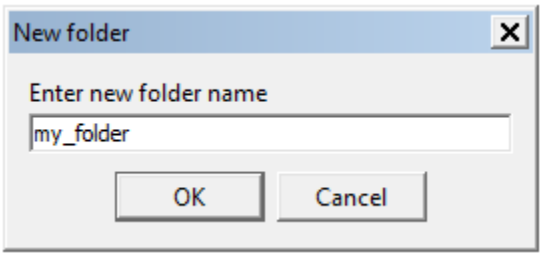

Figure 1-6: New folder dialog

Enter the new folder name in the dialog box and click OK. The new folder will be created on the disk and will appear in the project tree.

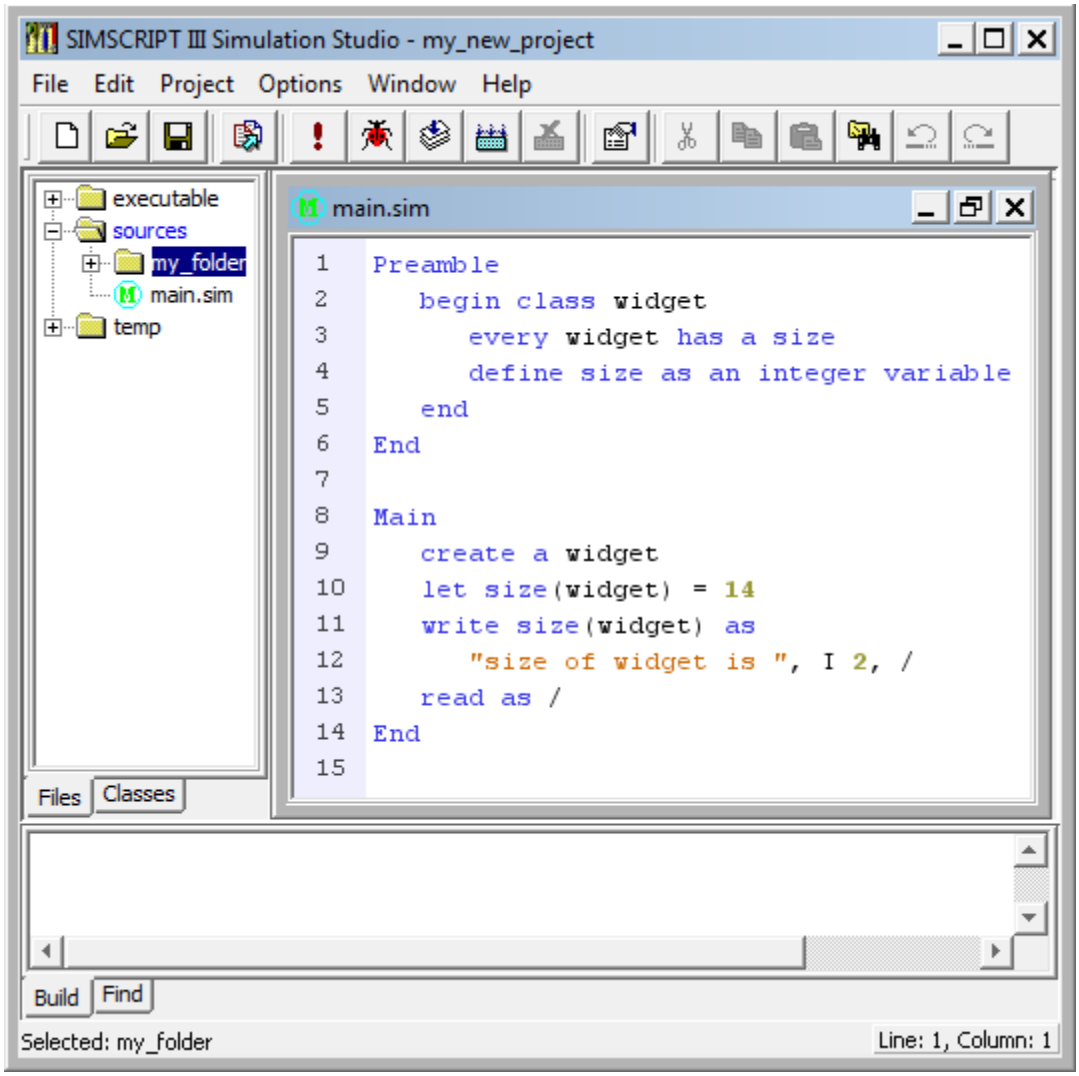

Figure 1-7: Creating a new folder in the project tree

You can right mouse click on this new folder to delete it from the disk and the project tree.

When you chose **add files** from the pop-up menu the browsing dialog box will appear. This allows you to add any file to your project. The added file will be copied to the selected directory and will appear in the project tree.

1.3.3 Adding Multiple Directories and Files

To add multiple source files that are organized in hierarchical multiple subdirectories, copy the whole directory structure with to the project sources directory. Use **Project->Update Project Tree** to include all directories and files for the project tree.

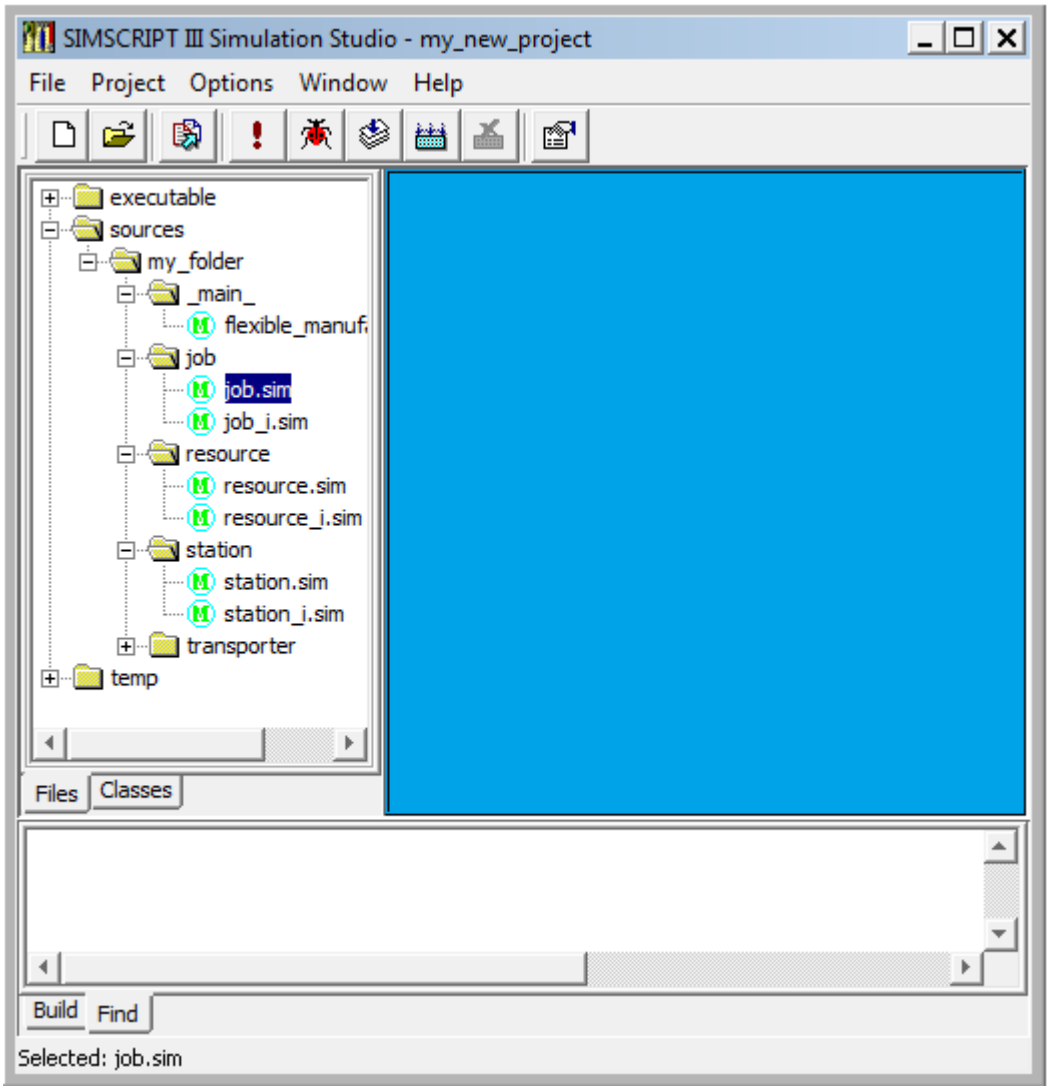

Figure 1-8: Importing source code files and directories.

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

Subsystems should also be placed in two files: *subsysname*.sim and *subsysname_i*.sim. The file subsysname.sim should contain the public preamble of the subsystem and

subsysname_i.sim should contain the private preamble (if any) and the implementation of the subsystem. The private preamble must appear first followed by the methods of the subsystem. For example, subsystem **resource** would be placed in **resource.sim** and **resource_i.sim** files.

All model files have to be placed in project directory **sources,** or a sub-directory of **sources.**

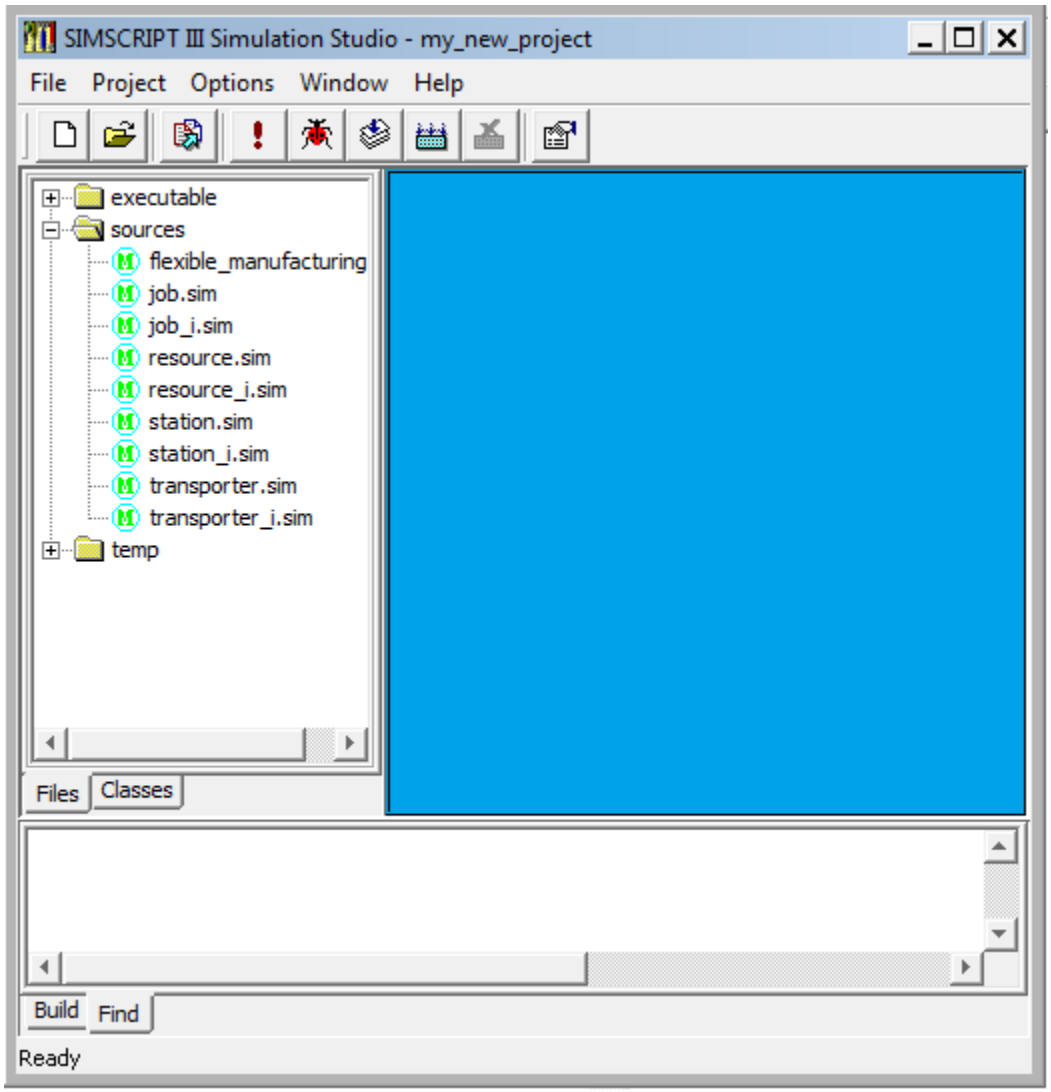

Figure 1-9: Project tree with SIMSCRIPT III source code with subsystems

1.3.4 Adding Graphical Elements to a Project

Graphical elements for your model are located in the **graphics.sg2** in the directory called **executable.** An empty graphics.sg2 container will be created with every new project.

Right mouse click on graphics.sg2 in the project window. This brings up a pop-up menu with the following options: **new**, **import** and **save**. If you click on **new**, a dialog box will be presented allowing you to name the new graphical element and to chose its type: Icon, Dialog Box, Simple message box, Menu bar, Palette, 2D chart, Pie chart, Analog clock, Digital clock, Dial, Level Meter, Digital display and Text display.

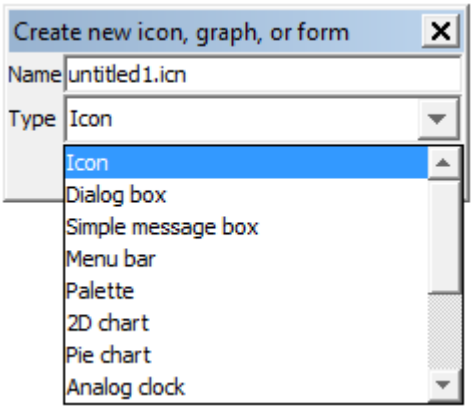

Figure 1-10: Create a new icon, graph or form.

After you define a type and click **Create**, a new graphical element icon will appear in the graphics.sg2 project window. This opens the graphics window in the Editor window along with the toolbar for the corresponding Graphical editor.

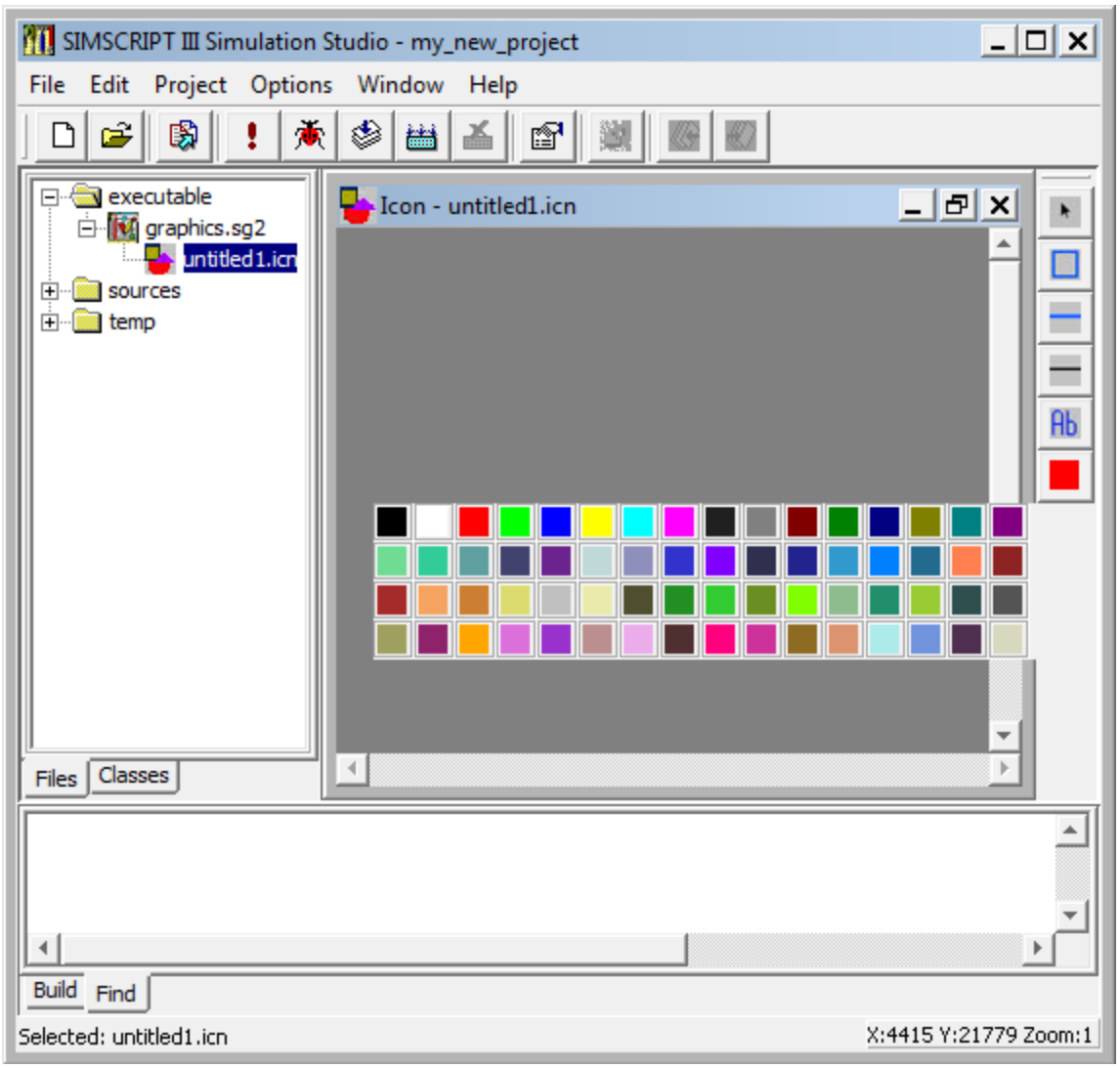

Figure 1-11: Adding a new Icon in Simstudio

A detailed explanation on how to create and use graphical elements in SIMSCRIPT III models can be found in the **SIMSCRIPT III Graphics Manual**.

1.4 Opening an Existing Project

To open existing projects use the **Project->Open** menu. The dialog box **Open Project** will appear allowing you to locate a project file.

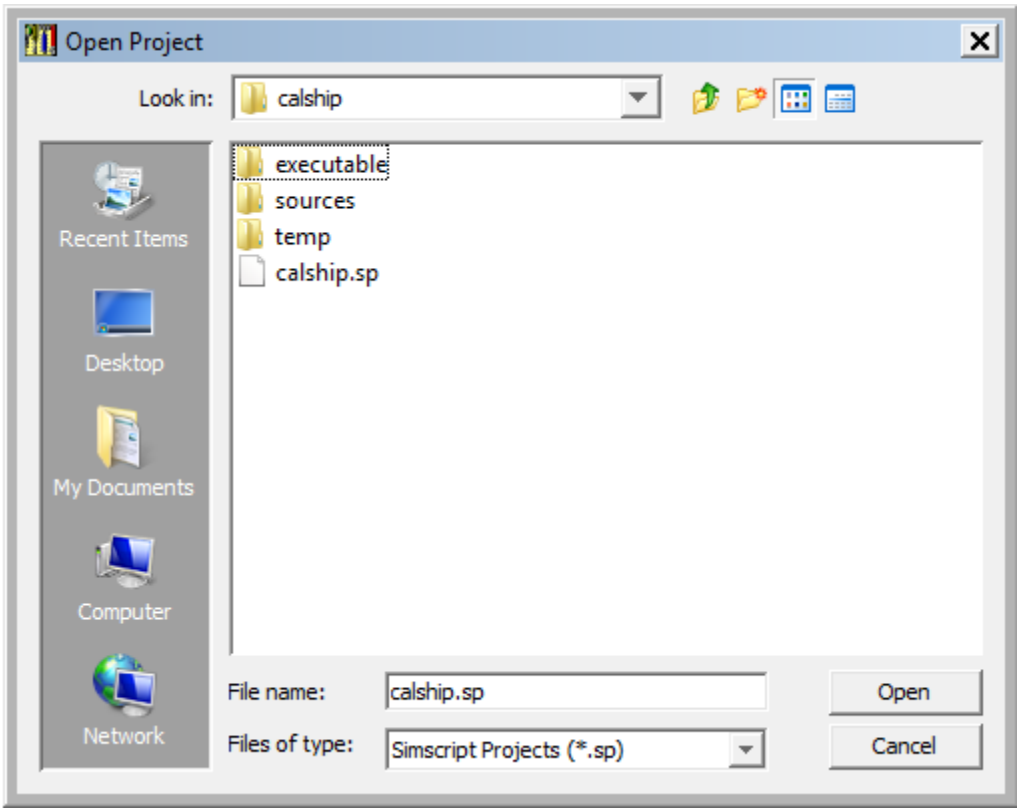

Figure 1-12: Open an existing project.

Select **project_name.sp** and click **Open.** The selected project will be opened for development.

1.5 Building a Project

Building a project can be done in two ways: using menu options: **Project->Build** or **Project-> Rebuild All**.

If you use **Project-> Rebuild All**, all source files will be compiled and linked. When you use **Project->Build** only the modules changed after the previous build will be recompiled, and the model will be re-linked.

Options for building a model are found by clicking on the **Options->Project** menu. It will bring up the **Project options** dialog**.**

If you are building an existing SIMSCRIPT II.5 project, in some cases you can use the compatibility switch so that compiler suppresses warnings about deprecated functionality, and your existing model should behave as before. But, usage of this switch will also exclude the new more efficient time scheduling mechanism and an improved algorithm for handling ranked sets.

The model can be built either for **release** or for **debugging**.

You can define compiler options for **release** mode to optimize code generation and to include run-time checking. For **debugging** mode you can define warning messages to be suppressed or displayed and run-time checking to be performed. You can also request various compiler listings to be generated.

Your model can be linked **With Graphics** libraries or **Without Graphics** libraries. Models that import from the **gui.m**, or **3d.m** modules must be linked **with graphics**. It can also be linked **statically** or **dynamically**. Static link will link all necessary modules in the executable, while dynamic link will link with the dynamic link libraries. Dynamic link is faster and convenient during model development. However, executables should be linked statically before begin moved to another computer for execution.

The name of the executable is by-default its **project_name,** but you can change it by typing the desired name into the **Binary** text box

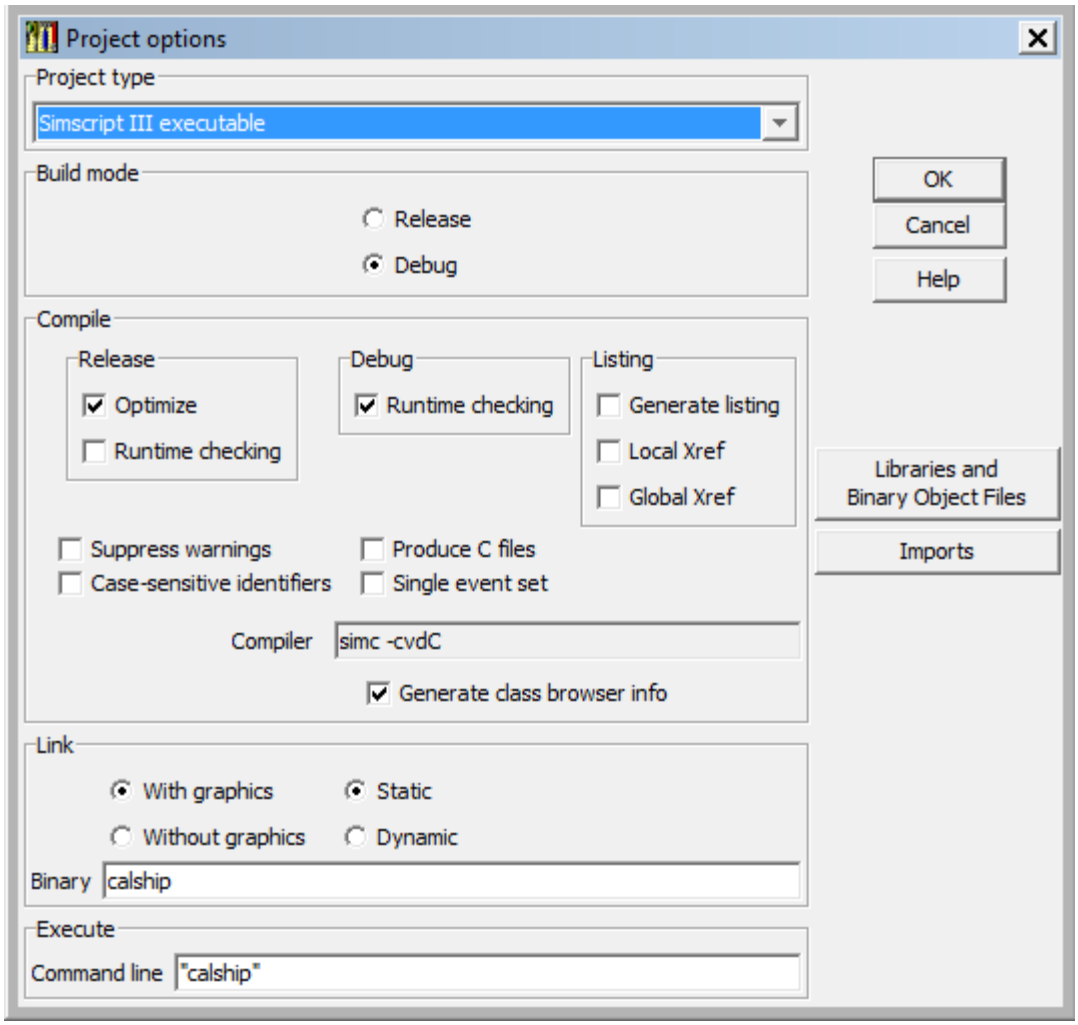

Figure 1-13: The Project Options dialog

1.5.1 Building a Project for Debugging

During the development of your model, you may want to build your project for debugging. Select **Options->Project** to open the Project options dialog box. Check build mode **Debug**. This will cause debugging facilities to be incorporated in your model.

You can also define if you would like compiler warnings to be presented or to be suppressed. During the debug phase, it is advisable that run-time checking be enabled – this will turn on entity/object attribute access checking and array index checking at runtime. A run-time error will be generated in for invalid accesses. These features will speed-up the testing phase.

To run your model with the debugger use **Project->Debug**. This will allow you to execute the model step-by-step and to observe model variables.

A model built for debugging can also be executed with **Project->Execute**. Project will run normally but in case of run-time error, control will be transferred to the debugger and you will have full debugging capabilities.

1.5.2 Building a Project for Release

Once a model has been debugged it can be rebuild in **Release** mode. Use **Options->Project** to bring up the Project options dialog box and click the **Release** radio button (under "build mode")**.** Choosing Optimization can increase execution speed**.** Turning off Runtime checking can increase speed even more. However, as seen in the following table, optimization has the following influence on the ability to locate the source of a crash or runtime error**.**

Build mode	**Optimization**	**Debugging notes**
Release	On	No traceback or debugging in event of crash.
Release	Off	Traceback available, but no line numbers or debugging.
Debug	Off	Full Traceback and debugging.

A model built in Release mode should be run with **Project->Execute.**

1.5.3 Compiler Listings

Checking the appropriate **Listing** boxes in **Project** Options will tell the SIMSCRIPT III compiler to generate various compiler listings: a compiler listing with or without local cross-reference, or a compiler listing with global cross-reference. All compiler listings will appear in the status window and will be placed in the **project_name.lis** file in the **temp** directory.

1.6 Using the Class Browser

Simscript III users of SimStudio can click on the **classes** tab at the bottom of the project tree view to show a tree representing all subsystems, classes, methods and routines in the model. This is a useful way to see a "summery" of your model, or to investigate a model that is not well known. The class browser also allows you to edit a particular implementation or definition of a particular construct by double–clicking on its name in the tree.

At the "root" level of the tree structure that composes the class browser there will be an item for each subsystem in the model.

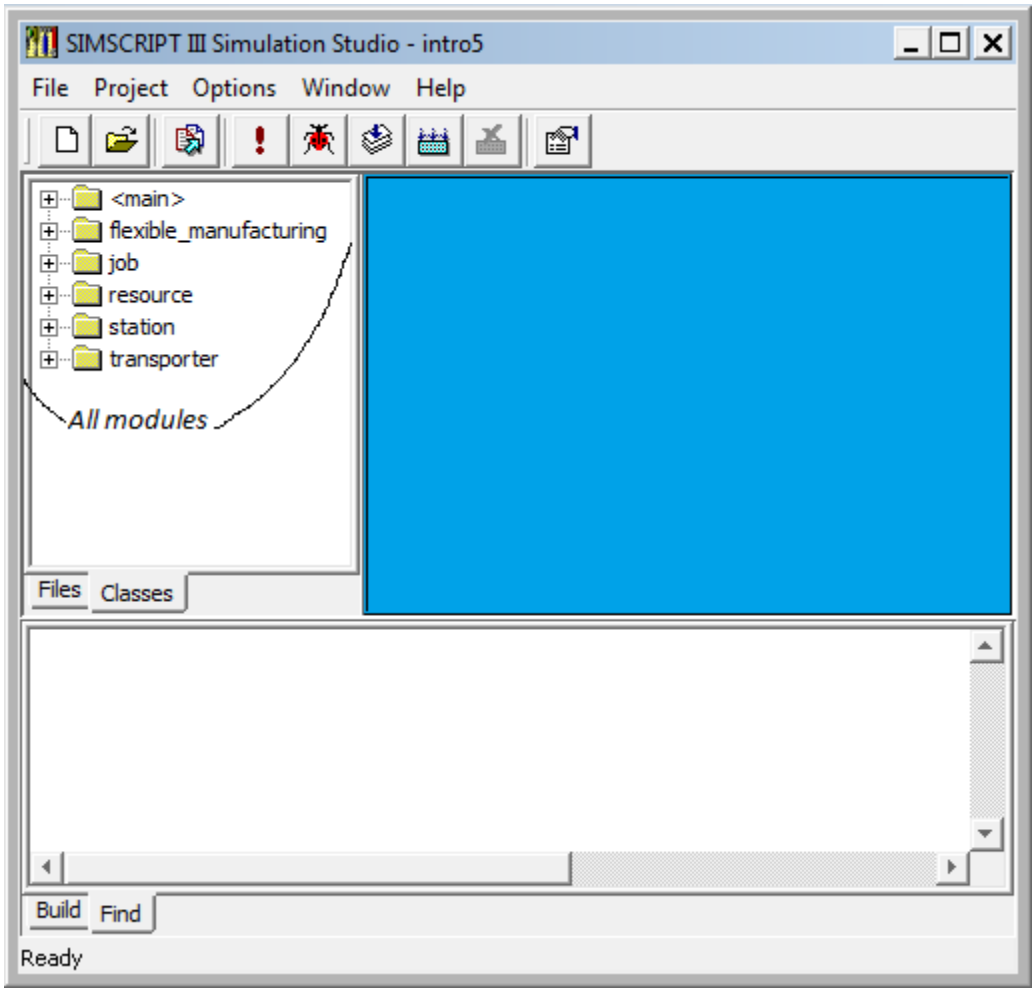

Figure 1-14: The Class Browser

Clicking to expand one of these items will allow you to choose to browse the public declarations, private declarations or the implementation code.

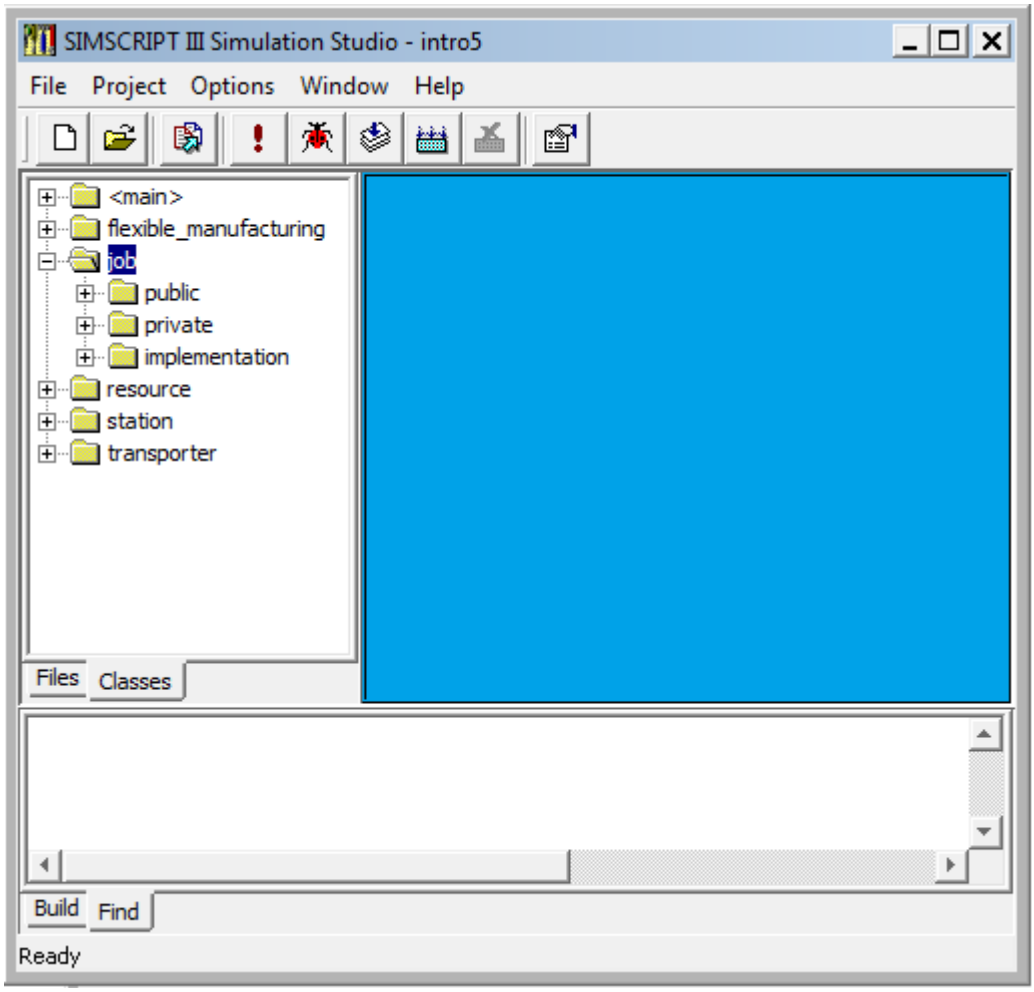

Figure 1-15: Class browser – selecting a module.

1.7 Executing a Model

After building, the model's executable is located in the project directory **executable**. To run it, use the menu option **Project->Execute**.

The *graphics.sg2* file will be found in the *executable* directory. All input data necessary to run the model should be placed in this directory.

Projects built in Debug mode should be executed using **Project->Debug.**

1.7.1 Passing Command-Line Arguments

To pass command line arguments to the model, or to redirect model output use the

Command line text box of **Project Options** to enter the command.

Project_name.exe – arg1 –arg2 …

Here is an example of the redirection of output of the model intro5.exe to a file intro5.out. When the program is run, intro5.out will be created in the **executable** directory.

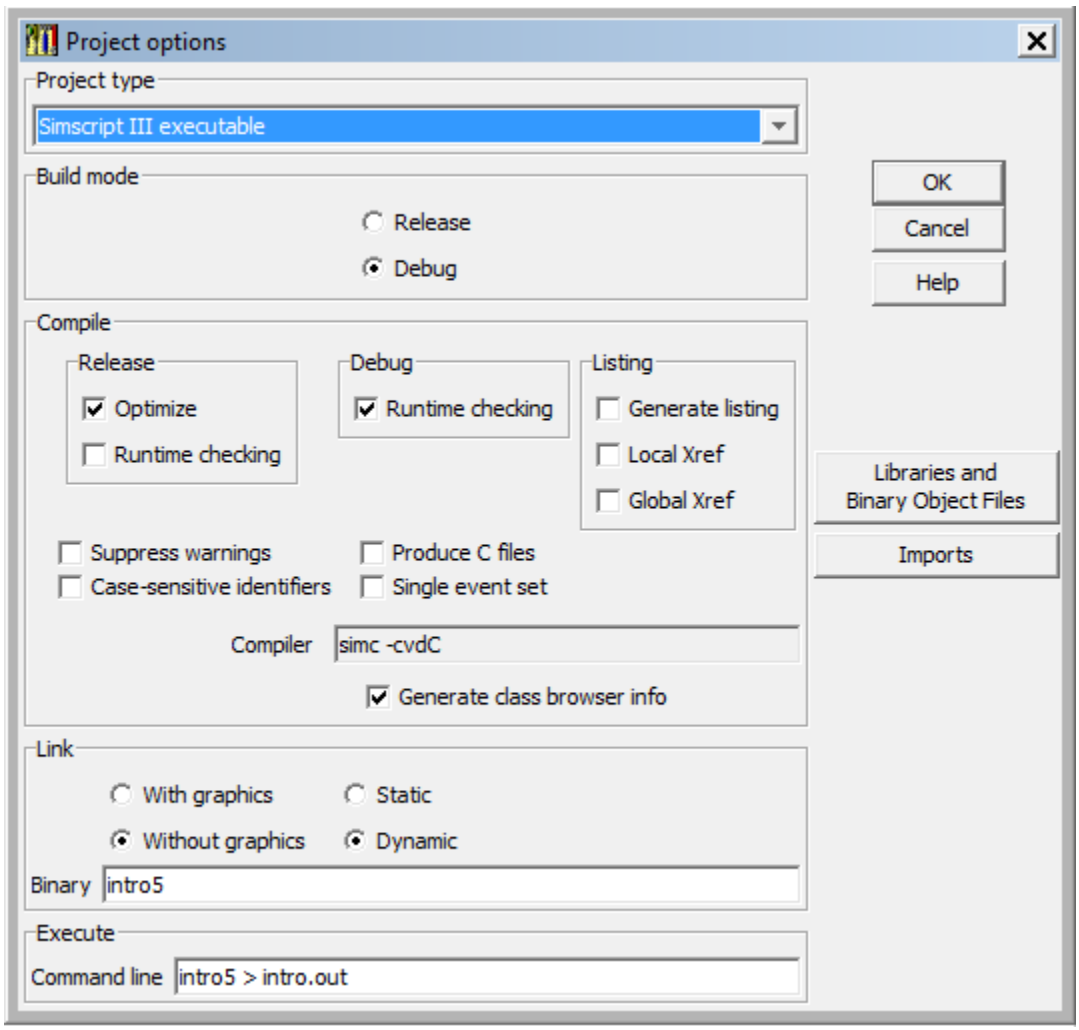

Figure 1-16: Defining the command line for model execution

1.6.2 Running the Executable with the Symbolic Debugger

If the executable was built in Debug mode it can either be executed using menu options **Project->Execute** or **Project->Debug.**

Project->Debug will invoke the symbolic debugger and the user will be able to have full debugging control during execution like: stepping, setting break points and viewing

model variables. Chapter 4 of this manual explains all debugging commands and facilities.

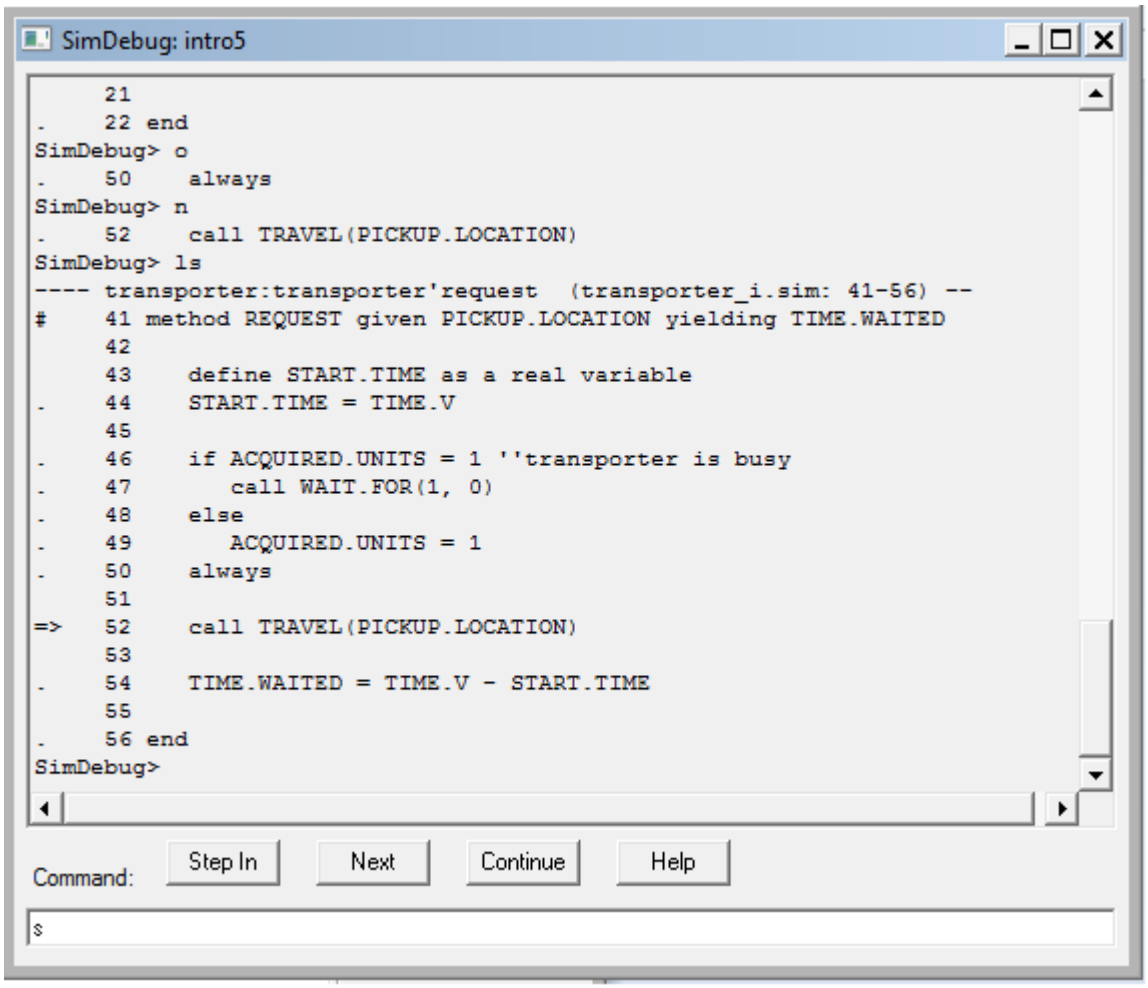

Figure 1-17: Simscript Symbolic Debugger window

1.8 Closing the Project

To close a project use menu option **Project-> Close**.

1.9 Setting Simstudio Preferences

Chose menu option **Options->Preferences**.. and set your preferences in the dialog box .

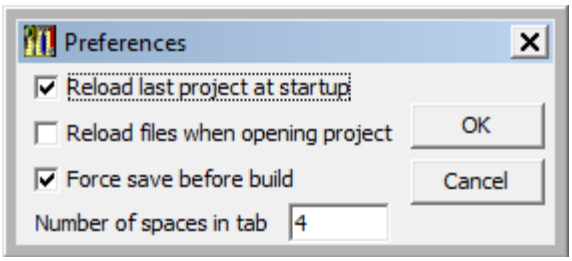

Figure 1-18: SimStudio preferences dialog**.**

"Reload last project at startup" will load the last project that was open at the time Simstudio was last shut down.Marking check box "Reload files when opening project", allows Simstudio to open files from the previous editing session. Checking "Force save before build" will automatically save an edited source code file before the project is built. For editing, the number of spaces per <tab> can be set also.

1.10 On-line Help

Simstudio provides full on-line help for all aspects of developing SIMSCRIPT Models, including: SIMSCRIPT language constructs, Simulation graphics Editors and graphics library, Simstudio, Command-line interface for developing models, List of Compiler and Run-time errors., using Symbolic Debugger, Data Base connectivity, etc.

Use menu option **Help** to invoke on-line help system.

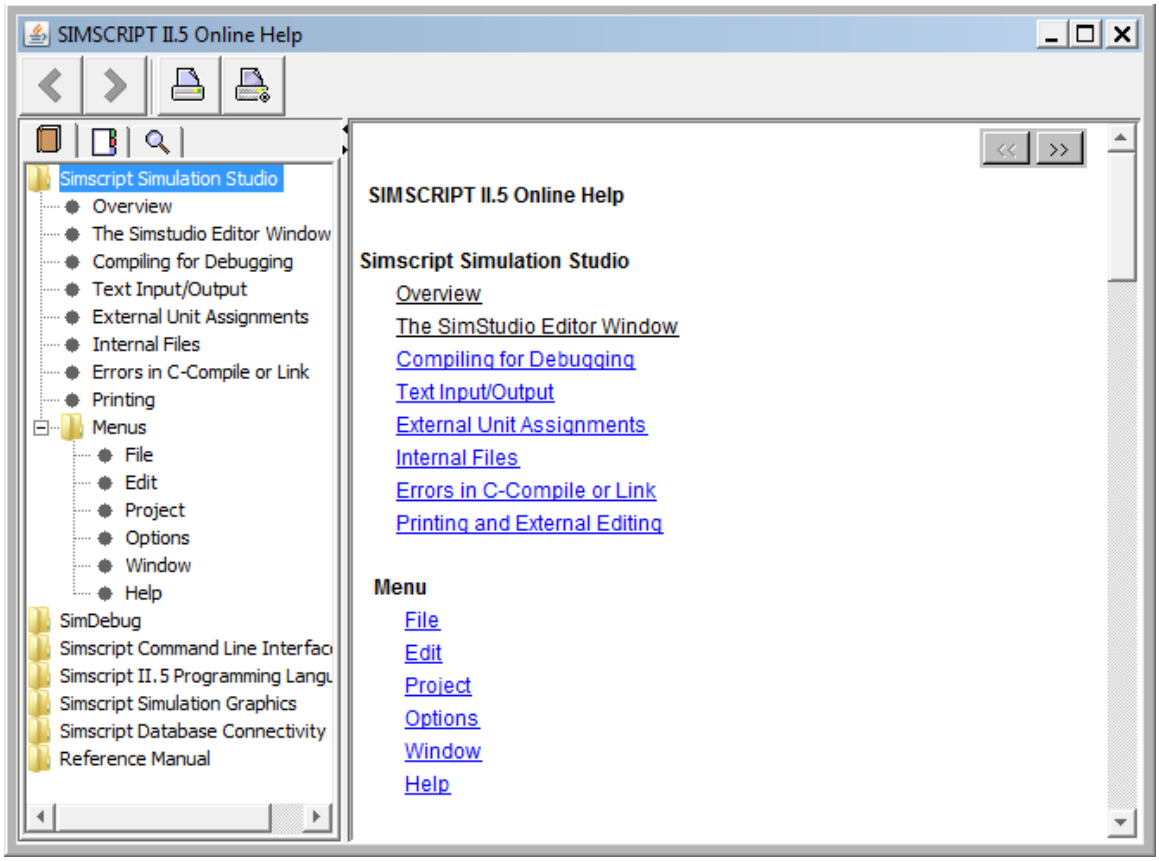

Figure 1-19: Simstudio on-line help window

1.11 Advanced Compiler/Link Options

Produce C Files…

Project options dialog box has two buttons used for more advanced model development. The "Produce C Files" check box will preserve the "C" files that are generated for each SIMSCRIPT source file at the time the model is built. If you check this option, you will only generate C files. If you want to build the model, you should not check this option.

Libraries and Binary Object Files…

Facilitates linking executable with objects from additional objects or external libraries.

Press "Move up"/"Move down" to change order of libraries and objects to be linked.

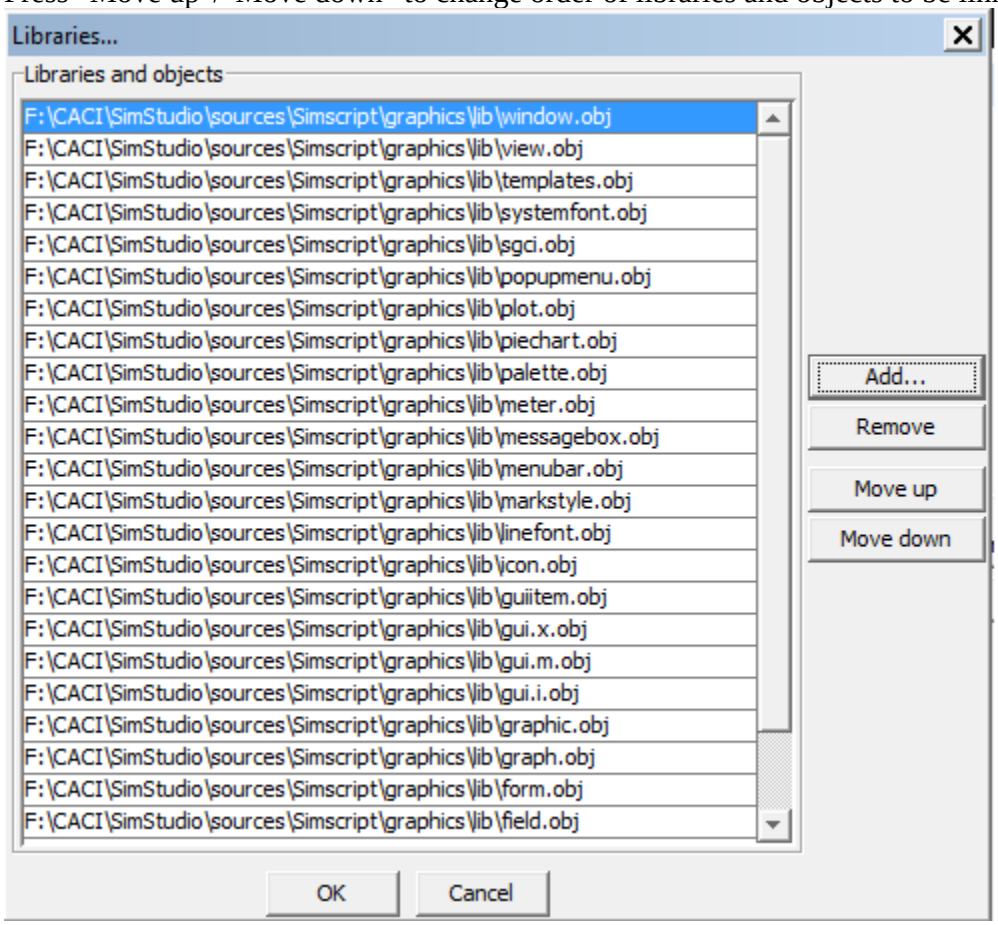

Figure 1-20: Importing libraries and objects for linking

Press "Add…" to select additional libraries and objects. Use the browser dialog to locate and select object files to add to the project. The <Shift> key can be used to select multiple files.

Imports…

This option is available for Simscript III projects. It specifies search path for "import" declarations found at the beginning of the public or private preamble.

Press "Add…" to select directories to be searched. Press "Move up"/"Move down" to change order of these directories.

By default, the path to the "defs" folder found in the 'SIMHOME' (distribution directory) is included. The **defs** folder contains public preambles for the **gui.m, 3d.m, 3dshapes.m, sdbc.m,** and **continuous.m** subsystems.

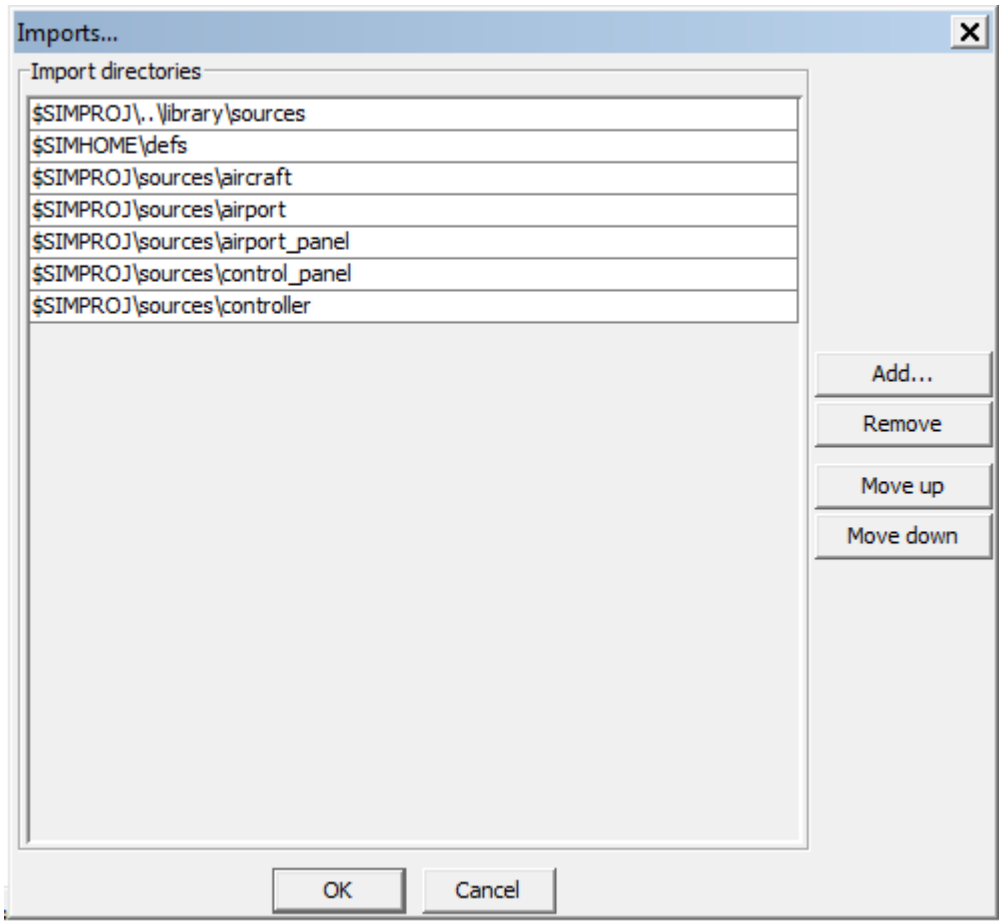

Figure 1-21: The Imports.. dialog box

2. Developing Simulation Models Using Command-Line Interface

Developing a SIMSCRIPT III program, using command-line interface commands, typically involves the following steps:

1. Preparing one or more SIMSCRIPT III source files using a text editor.
2. Compiling the program and checking for compilation errors.
3. Editing and re-compiling the program, as needed, until there are no compilation errors.
4. Linking the object files generated by the compiler to produce an executable file.
5. Executing the program.
6. Debugging the program. In case of errors during execution, the program should be compiled with the debugging option linked and then executed with the interactive SIMSCRIPT symbolic debugger to examine the state of the program and find the cause of the error.

2.1 Preparing Source Files

A SIMSCRIPT III program may be prepared using **ed4sim** SIMSCRIPT syntax color coded editor, **vi, emacs** or any other text editor.

If the program is small, it is convenient to store the entire program within a single file. If the program is large, it is best to organize the code as subsystems. Files containing SIMSCRIPT III source code must be given names that end with **.sim** or **.SIM**. The public preamble of a subsystem called *name* MUST be placed in the file "name.sim". There are no naming restrictions on source files containing private preambles or implementation code.

As a convention, you can place the private preamble followed by the implementation and the file "name_i.sim". Although not a requirement, it is easier to compile and link a SIMSCRIPT III program that is stored in a directory of its own; i.e., a directory containing the entire source files of the program in question and none of the source files of other programs.

2.2 Compiling

The SIMSCRIPT III compiler translates a program written in the SIMSCRIPT III programming language into one or more object files. The compiler uses C as an intermediate language, but this is transparent to you, the SIMSCRIPT III program developer. The compiler

will write diagnostics — error messages and warning messages — to **stderr**. Errors prevent the generation of object files; warnings do not. See Appendix A for a complete list of error and warning messages that are issued by the compiler.

The **simc** command is used to invoke the SIMSCRIPT III compiler and linker. Its general form is:

% simc [options] file1.sim, file2.sim, ...

For example, to compile and link a program consisting of a single source file named **abc.sim,** enter:

% simc abc.sim

This command will compile the SIMSCRIPT source file **abc.sim**, reporting compilation errors and warnings to the terminal. If the compilation is successful, the object module **abc.o or abc.obj** will be linked producing an executable file named **a.out** or **a.exe**, depending on a platform.

The SIMSCRIPT compiler options follow the same general format as many C compilers and other standard UNIX compilers. The options available should be familiar to experienced programmers. Below is a brief overview of a few of the most commonly used options:

-c Do not link any object files after compilation.

-d Enable SIMSCRIPT symbolic debugging.

-e Compatibility switch for SIMSCRIPT II.5 programs

-h Use a single event set for all process notices.

-I Specify path for imported public preambles.

-l Display a program listing.

-o name When linking, create an executable with the name provided.

-s Turn on case sensitivity for names.

-v Compile the preamble as "`VERY OLD`". See below for more details.

-w Do not report any compiler warnings

-w kk, nn, mm Do not report compiler warnings number kk, nn, mm

-O Maximize speed by eliminating the traceback in the event of a crash.

-T Link the program statically instead of dynamically.

-X Display a local cross-reference listing for a module

Below is a complete list of the options available in the SIMSCRIPT III compiler:

-a Compiler will produce a file containing the generated source code for the together with the SIMSCRIPT source code as comments. Produces a **.c** file with "ALLSTARS" comments, which shows the expansion of complex SIMSCRIPT statements into simpler ones.

-c The compiler's default behavior is to link using **simld** after compilation. If you want to stop this from happening, use this option.

-C The compiler will generate code to perform full runtime checking. This code validates every array element reference and every attribute reference (for both object and entities). Also, in the event of a runtime error, a more elaborate traceback will be provided. This option allows SIMSCRIPT III to detect a larger class of runtime errors and should be used when compiling a program that is not fully debugged. Both the traceback and runtime error checking will make your programs run somewhat slower. Note that runtime checking is not enabled by default.

-C0 Provide runtime checking for array element reference only without entity class checking and object type checking. Note this is **C**"zero", not **C**"oh".

-d Selects 'compiling for debug'. The compiler is fully integrated with the SIMSCRIPT III symbolic debugger. After linking, the program can be activated with the command line switch **-debug** to provide interactive dialog with the debugger. The SIMSCRIPT III symbolic debugger allows you to study and change the behavior of a model at runtime. Debugging features include the following:

- Setting a break point in a method or a routine, or in an active SIMSCRIPT process instance or process-method.
- Single stepping one source line at a time
- Viewing source code
- Displaying of local, global variables, object attributes temporary entities in various formats and their modifications
- Displaying the status of the program: I/O and memory usage statistics etc.

To use all the debugger functions, a SIMSCRIPT II.5 program must be compiled with the **-d** compilation switch. To start a program in "'debugging mode" where you can set breakpoints etc., the executable should be invoked with the **-debug** option:

```
simc -d prog.sim -o prog
prog –debug
```

The **-debug** option is internal to SIMSCRIPT and will not be seen by the user program.

A runtime error will automatically activate the debugger so that you can examine the current stack and variables that led to the error. If the program was not compiled with the **-d** option, only a minimal set of debugging functions will be available. If the program was compiled with the **-d** option, all debugger functions will be available. An on-line help command **h** will display a list of available debug commands and parameters. See chapter 4.

-e Compatibility switch for use with existing SIMSCRIPT II.5 models. It will suppress warnings for deprecated functionality, but the model will not have benefits of SIMSCRIPT III faster execution and handling of ranked sets.

-G Link a SIMGRAPHICS program using **simgld.**

-h The compiler will place all event and process notices into a single event set EV.S(1). This means that the ordering of process notices is in chronological order. Process notices that are scheduled for the same exact time will be ranked as FIFO (first activated first run). Note that the *priority order* and *break ties* statements are not allowed when compiling with the **–h** option.

-l The compiler will write a listing to the standard output. Typically, standard output is redirected to a file. For example, to write a listing to a file named **listfile**, enter:

```
% simc -l *.sim > listfile
```

The listing shows the source statements together with diagnostic messages, if any. It may also include local and/or global cross-references (see the **–x** and **-X** options).

-I path Import declarations which are in the public preambles in the path

-L n The compiler will produce output listings with n lines per page. The default value is 55.

-o name When linking, the executable file created will be called **name**. If this option is not specified, **a.out** is the default executable name. For example, the following command creates an executable called **file** after compiling all the **.sim** source files in this directory.

```
simc *.sim -o file
```

-O This will compile the SIMSCRIPT III code without any traceback support in an effort to improve speed. Normally, if a runtime error occurs the debugger

window is shown. Even if the model is compiled without debugging, you can type the "t" command to see which routine or method that caused the runtime error. When a model compiled with **–O** crashes, the debugger window is not shown.

simc -O -o prog filename.sim

-s Turn on case sensitivity. Code compiled with this option will have case sensitive names (but not keywords). This applies to all routine, method, attribute, variable, etc names. For example:

```
Define My_Var as an integer variable
. . .
Let my_var = 6  ``error! my_var is undefined!
```

-v This option means a `VERY OLD PREAMBLE`. It is used during re-compilation of SIMSCRIPT routines when there are no changes to the **Preamble.sim**. It will speed-up the re-compilation process because **Preamble.o** will not be generated. Also, the `PREAMBLE` will not appear in the listing.

For example, enter the following command to re-compile **file1.sim** into an object file (which will be called **file1.o**). The name of the file which contains the `PREAMBLE`, **Preamble.sim,** must always be given because it contains definitions for SIMSCRIPT data structures. The **-c** option prevents linking.

simc -cv Preamble.sim file1.sim

Enter the following command to create an executable called **a.out** (the default name) from the object files in this directory after re-compiling **rout1.sim .**

simc -v Preamble.sim rout1.sim *.o

It can also be used for recompilation of implementation part of the subsystem when public preamble has not been changed.

-w The compiler will suppress warning messages, i.e., no warning messages will be displayed.

-wA,B The compiler will suppress warning messages, i.e., no warning messages will be displayed for errors number A and B. For example, to get rid of only the warning 1002 "Local variable not used":

simc –w1002 Preamble.sim rout1.sim

-X The compiler will append to the listing a global cross-reference for the entire program. A global cross-reference shows the name of every routine or method, which references each globally defined name.

The following command compiles a program consisting of main module with three source files: **abc.sim, def.sim** and **ghi.sim**. Warning messages will be suppressed (**-w** option) and runtime checking code will be generated (**-C** option).

% simc -w -C abc.sim def.sim ghi.sim

The following is a convenient way to compile a program consisting of many source files within a single directory:

% simc *.sim

In this example, ***.sim** is automatically expanded into a list of source files sorted by name.

If the program consists of main module mname and a set of subsystems sub1, sub2, sub3, *sub1.sim, sub2.sim,* and *sub3.sim* must contain (at the beginning of the file) the public preambles for the subsystem *sub1, sub2,* and *sub3.* As a convention we can put implementation code for *sub1* in sub1_i.sim, i.e. private preamble and methods and routines (although this is not a requirement). By convention the preamble for the main module "mname" should be placed in file "mname.sim".

The SIMSCRIPT III compiler uses the headings (first line of code) at the beginning of a source file to determine what the file contains. Here are the possible headings

Public preamble for the *name* subsystem -- Public preamble. Source file MUST be named "name.sim". (Can optionally be followed a private preamble or implementation code).

Private preamble for the *name* subsystem – Private preamble (optionally followed by implementation code). There is no file naming restriction.

Preamble for the *name* system – Main module preamble (optionally followed by implementation code). There is no file naming restriction.

Implementation for the *name* subsystem – Contains implementation code for subsystem *name.* There is no file naming restriction. This statement must be used if implementation code for a subsystem is to span multiple files.

<no heading> -- Contains implementation code for the main module or "the system".

Importing

If the program imports some external modules/subsystems like exsub1, exsub2, which reside in the directories dir1 and dir2 respectively, then directories where the public preambles of the imported modules reside, have to be given to the compiler in the command like:

simc –c –I dir1 –I dir2 *.sim

Object files exsub1_i.obj and exsub2_i.obj or the library which contain the implementation should be included during the linking of the model.

2.3 Recompiling

Main module

If the model contains only a main module, whenever a change is made to the PREAMBLE, it is necessary to re-compile the entire program. If a changes are made only to routines of the program, only the modified routines need be re-compiled, not the entire program. Suppose that the routine in file **xyz.sim** has been modified. If this routine does not require anything declared in the PREAMBLE, then the following command can be used to re-compile it:

```
% simc -c xyz.sim
```

If this routine does reference something declared in the PREAMBLE, it is necessary to recompile the PREAMBLE along with it:

```
% simc -cv PREAMBLE.sim xyz.sim
```

The **-v** option is specified to avoid regenerating the scripted routines contained in the **PREAMBLE.o**.

Subsystems

When a program contains main module and a set of subsystems the subsystems can be compiled separately. If any file in a subsystem is modified, the whole subsystem has to be recompiled. If the public preamble of a subsystem is modified, all the subsystems which import this module (or import modules that import this module) have to be recompiled and subsequently re-linked.

2.4 Linking

If the **-c** option is used to suppress linking, the compiler generates object files, which need to be linked. Each of these files has a name that ends with **.o** on Unix or **.obj** on Windows platform. The **simld** command is used to link a SIMSCRIPT III non-graphical program. Its general form is:

```
% simld file.o ...
```

If there are any undefined references, the name of each missing routine will be displayed.
If there are no undefined references, an executable file named **a.out** will be produced. (**a.exe** is produced on Windows). Suppose a program consists of only three routines: **main.sim, sub1.sim** and **sub2.sim**. Then the object files generated by the compiler are **main.o, sub1.o** and **sub2.o**. The following command will link this program:

```
% simld main.o sub1.o sub2.o
```

The following is a convenient way to link a program consisting of many object files within a single directory:

% simld *.o

Note that it is necessary to link all of the object files generated by the compiler. Even if just a single routine has been modified and re-compiled, it is necessary to re-link the entire set of object files.

simld is a shell script which invokes the UNIX C compiler, **cc**, to link object files. Any option, which may be specified to cc, may also be specified to **simld**. The most useful of these is the **-o** option. It is used to name the executable file something other than **a.out**. For example, to create an executable file named **compute**, enter:

% simld -o compute *.o

simgld is another shell script which invokes **cc**. It must be used instead of **simld** to link with graphics or with the built in subsystems distributed with SIMSCRIPT III. For example, to link a graphics program and name the executable file **animate**, enter

% simgld -o animate *.o

It is possible to create a library of SIMSCRIPT III routines using the UNIX archive utility, **ar**. To create a library named **xyz** from the object files in a directory, enter the following command:

% ar r libxyz.a *.o

To make the library accessible to all users, enter the following sequence of commands:

% mv libxyz.a $SIMHOME/lib
% ranlib $SIMHOME/lib/libxyz.a
% chmod 644 $SIMHOME/lib/libxyz.a

SIMHOME is the environment variable, which contains the full path where SIMSCRIPT III is installed. For more details of the **SIMHOME**, see the Installation Notes for the current SIMSCRIPT III release.

Note that **ranlib** is not available on all systems. On systems where it is not available it is not needed. To link the object files in a directory with this library, enter:

% simld *.o –lxyz

A SIMSCRIPT III program can call routines written in other languages, such as C or FORTRAN. To link such a program, specify to **simld** (or **simgld** if the program makes use of SIMGRAPHICS features) the name of each object file created by the other compiler, along with the name of each object file created by the SIMSCRIPT III compiler.

Compiler switch **-G** will link graphical models with the graphics libraries.

```
simc -G *.sim
```

SIMSCRIPT III runtime libraries are distributed in two versions: dynamic link libraries and archive libraries. This facilitates **dynamic** and **static linking**. By default programs will be linked dynamically.

When a model is linked dynamically, the executable image does not include the entire object modules it needs for execution. It contains pointers to the dynamic link libraries also called "shareable libraries". The benefits of dynamic linking are twofold: first linking time is shorter, second all SIMSCRIPT models in the same computer platform share the same runtime libraries which results in substantial savings of disk space. When you use existing link commands: **simld, simgld** your model will be linked dynamically.

If you want to execute your model on some other platform, which does not have the same release of SIMSCRIPT III, or does not have SIMSCRIPT III installed at all, your model must be linked statically. SIMSCRIPT III provides commands for platform independent static linking or "total linking" for both non-graphical and graphical SIMSCRIPT models:

```
tsimld      - static link of non-graphical models
tsimgld     - static link of graphical models by with graphics
```

2.5 Executing

A SIMSCRIPT III program is executed by entering the name of the executable file. For example:

```
% a.out
```

Parameters specified on the command line are available to the SIMSCRIPT III program in the global **text** array, **parm.v**. For example, consider the following command:

```
% a.out -i 10 WXYZ.dat
```

Upon entry to this program, **parm.v** will be set up as follows**:**

```
DIM.F(PARM.V(*)) = 3
PARM.V(1) = -i
PARM.V(2) = 10
PARM.V(3) = WXYZ.dat
```

A SIMSCRIPT III program can read from standard input by reading from UNIT 5. It can write to standard output by writing to UNIT 6 and can write to standard error by writing to UNIT 98. Any redirection of these units, which is allowed by the operating system, may be specified on the command line.

Internal command line switches used for debugging, like **-debug** and **-batchtrace,** will not

be seen by the program in **parm.v**.

If a runtime error is detected by SIMSCRIPT III, the program will be stopped and:

1. A runtime error message will be written to standard error (see Appendix B for a complete list of runtime error messages) and the interactive debugger dialog will be entered allowing you to examine the state of the program;

2. If the program was invoked with the command line switch **-batchtrace**, a runtime error message, a traceback, a simulation status report, a memory status report and an I/O status report will be written to a file **simerr.trc** and the user-supplied snapshot routine, **snap.r**, will be called, if it exists. The level of debugging information included in a traceback depends on the compiler switches used for compilation: **-d** will provide method names and routine names with local variables and line numbers. Compiling with no switches will include routine and method names only.

Any SIMSCRIPT III program may be invoked from a shell script. The exit status returned by the program will be zero if the program was terminated by a **stop** or **end** statement, and will be non-zero if the program was aborted due to a runtime error. However, you may explicitly call **exit.r** to terminate your program and return a particular exit status.

2.6 Makefiles

The file-naming scheme that this compiler uses is compatible with the naming scheme used by the C language compiler. Because of this, it is possible to use the UNIX *"make"* utility. This utility only recompiles the source files that have changed since the last compilation. This is an easy and reliable way to manage models of medium to large size. *Make* is not very good at handling models whose sources are spread over many directories but, with care, it is possible.

The *make* utility relies on a special file, called a "make file", to describe the rules for rebuilding your particular model. By default, the "make file" is named either **makefile** or **Makefile .** Other file names may be specified with the **-f** option of *make*. See the man page for **make(1)** for more information.

2.6.1 Compilation Sequence

The compiler knows about the following kinds of file extensions, and treats them as follows:

.sim: Compile as SIMSCRIPT source files.

.SIM: Alternate suffix for SIMSCRIPT source files.

.o: Object files.

.c: `C` source files, produced in intermediate stage.

.a: Archive libraries to include in linking.

Files *must* be named using this convention. For other kinds of file extensions, consult the manual for your **C** compiler. Files are named after the SIMSCRIPT source using the following convention

```
myfile.sim -> myfile.o
```

This allows the use of makefiles.

The easiest way to use the compiler is to simply specify all the sources you want compiled, and let the compiler compile and link them into an executable program. However, during development of a large program, only recompiling those source files that have changed since the previous compilation can save much time. This is accomplished by saving the object file for each source file. Then, when a source file is recompiled, the new object file replaces the old, and all of the object files can be relinked to create a new executable. Linking all of the object files is *much* faster than compiling all of the source files.

Make takes this one step further. It checks the modify time of each source file, and only recompiles it if it is *newer* that its object file or the target executable. This way, only the source files that need compiling are actually compiled. The actual compilation and linking commands are specified in the makefile.

2.6.2 Make Description File Format

The descriptions in this section are simplified. For a complete description of the file format, see the documentation that came with your system.

Entries in a makefile are of the following form:

target1 [target2 ...] : [dependent1 ...]
<tab> command [# comments ...]

Items in square brackets are optional. The **<tab>** must be a "tab" character. Shell metacharacters such as **'*'** and **'?'** are expanded. The entry is concluded with a blank line.

Makefiles can also contain simple macros. Macros can be defined in the make command line, or more commonly, in the makefile. The definition is simple: a macro name, an "equal" sign, and the macro value. An example is **PREAMBLE = Preamble.sim.** A macro is invoked by preceding the name with a dollar sign (**$$** is used to represent a real dollar sign). Macro names longer than one character must be parenthesized like this: "**$(PREAMBLE)**"**.** When the macro is invoked, its text is replaced with its current value, so in our example, "**$(PREAMBLE)**" would be replaced with **Preamble.sim.** *Make* also has four predefined macros specific to the job it performs. These special macros are **$*, $@, $?,** and **$< .** These macros are re-evaluated before each command. They are evaluated as follows:

- The **$*** macro is the root file name of the current file. For example, if the current file were **frequency.sim**, **$*** would equal **frequency.**
- The **$@** macro represents the current "target" file name.
- The **$?** macro is the string of file names found to be newer than the current target.
- The **$<** macro is the name of the file which caused this command to be executed.

2.6.3 Transformation Rules

A transformation rule is what *make* uses to "transform" a source file into an object file, or several object files into an executable. Many useful transformation rules are built into *make*, such as rules to compile **C,** FORTRAN, or even assembler. Unfortunately, the rules for SIMSCRIPT are not built in.

To provide *make* with this information, *make* must first be informed of the new source suffix, **.sim.** This is done using a fake target called. SUFFIXES**.** For our purposes, SUFFIXES**: .sim .o** is sufficient. Next, *make* needs to know how to transform **.sim** files into **.o** files. We do this using a transformation rule called. sim.o**.** See the sample makefile in paragraph 2.7.5 for an example. In transformation rules, the special macros are set as follows: **$*** is set to the file name without the suffix, **$<** is the name of the file to be transformed, and **$@** is the name of the file to be created (or updated).

2.6.4 Special Notes

Each line in a makefile is executed by a new invocation of the shell, so commands like **cd** for example, must be combined into one line using the shell command separator, "**;**"**.**

By default, *make* displays each command before executing it. This can be prevented by preceding the command with an at sign (@)**.**

If a macro is defined on the *make* command line, it supersedes the makefile's definition, if any is present. A typical use of this is to use **make SFLAGS=-O** to use optimization on any compiles that need to be performed.

There are several ways to force recompilation:

1. Use **touch(1)** to update the source file's modify time. *Make* will then consider the source file "changed". This will also force relinking if the corresponding object file is a dependent of the executable.

2. Delete the corresponding object file. This has the same effect as the above.
3. Delete the executable. This will force relinking, but will not recompile any sources unless they are out of date.

2.6.5 Sample Makefile

```
#
#                        Generic makefile for SIMSCRIPT programs
#
# MAKE ARGUMENTS:
#     <no arg> : Make executable with the name in the "PRG" parameter.
#     clean    : Remove all non-source files, i.e. object files and
#                  the executable and all intermediate files.
#     cleanexe   : Remove the executable.
#---------------------------------------------------------------------



#=====================================================================
#    FILL IN THE PARAMETERS BELOW UNTIL THE LINE
#    ">>> END OF PARAMETERS <<<"
#=====================================================================
#
# <<< PARAMETERS >>>
# PRG: The name of the executable.
PRG = bounce

# PREAMBLE: SIMSCRIPT source file containing the preamble.
# SIMFILES: All other SIMSCRIPT source files. A "\" followed
#           immediately by a carriage return must be put at the
#           end of the line to continue to the next.
PREAMBLE = Preamble.sim
SIMFILES = ball.sim bounce.sim done.sim init.sim main.sim menu.sim \
           menuctl.sim output.sim
# SFLAGS: SIMSCRIPT compile flags.
SFLAGS = -d
# SIMLINK:   Specify link command with SIMGRAPHICS I, SIMGRAPHICS II,
#           or no graphics; dynamic or static link.
#
#           <<< DYNAMIC LINK >>>
#           SIMGRAPHICS I - simgld1
#           SIMGRAPHICS II - simgld2 or simgld
#           NO GRAPHICS - simld
#
#           <<< STATIC LINK >>>
#           SIMGRAPHICS I - tsimgld1
#           SIMGRAPHICS II - tsimgld2
#           NO GRAPHICS - tsimld
SIMLINK = simgld
# >>> END OF PARAMETERS <<<
#
#=====================================================================
#========== BELOW HERE NO CHANGES SHOULD BE NECESSARY ===============
#=====================================================================
```

```
# SIMC:      SIMSCRIPT compile command.
SIMC = simc

# OBJS: List of .o files.
OBJS = $(PREAMBLE:.sim=.o) $(SIMFILES:.sim=.o)

# The first (empty) .SUFFIXES clears the SUFFIXES list. The second
# acknowledges only the .sim and .o suffixes. This avoids problems
# with extraneous .c files and others.
.SUFFIXES:
.SUFFIXES: .o .sim .c



$(PRG)  : $(OBJS)
        @echo "-- Linking ..."
        $(SIMLINK) -o $(PRG) $(OBJS)
        @echo "-- $(PRG) was successfully built!"

clean :
        @echo "-- Removing all intermediate files and the executable."
        rm -f *.o *.c *.i *.s *~ core a.out $(PRG)

cleanexe :
        @echo "--- Removing executables."
        rm -f core a.out $(PRG)
#------------------------- RULES ----------------------------------
#
# If preamble was changed, we need to recompile everything. Since
# after that all *.o will be current, just the link is left in the
# target above.

$(PREAMBLE:.sim=.o): $(PREAMBLE)
        @echo "-- $(PREAMBLE:.sim=.o) outdated or missing!"
        @echo "-- Recompiling everything ..."
        $(SIMC) -c $(SFLAGS) $(PREAMBLE) $(SIMFILES)


# How to make an individual object file from a simcript source file.

.sim.o:
         $(SIMC) -cv $(SFLAGS) $(PREAMBLE) $*.sim
```

2.7 Obtaining Online Help

Online documentation regarding the use of the SIMSCRIPT compiler can be obtained at the SIMSCRIPT WEB Site

http://www.simscript.com

2.8 Example Program

The following is an example of a complete program and compilation.

```
 % ls
```

```
      main.sim     SIMU01    job.sim     stop.sim
 % simc -l *.sim > listing
 % ls
      Preamble.o  a.out*     job.o       main.o      stop.sim
Preamble.sim     generator.o job.sim     main.sim
SIMU01           generator.sim listing   stop.o
% cat listing
                                                          PAGE 1
CACI  SIMSCRIPT II.5 (R) v2.0                 6/26/1997 15:23:42

      1 PREAMBLE
      2
      3     RESOURCES INCLUDE CPU AND MEMORY
      4     PROCESSES INCLUDE GENERATOR AND STOP.SIM
      5        EVERY JOB HAS A JB.PRIORITY
      6                  AND A JB.MEMORY.REQUIREMENT
      7        DEFINE JB.PRIORITY AND JB.MEMORY.REQUIREMENT
      8                  AS INTEGER VARIABLES
      9        DEFINE JOB.DELAY.TIME AS A REAL VARIABLE
     10    EXTERNAL PROCESS IS JOB
     11    EXTERNAL PROCESS UNIT IS 1
     12    DEFINE SMALL.JOB.INTERARRIVAL.TIME,
     13      MEAN.SMALL.JOB.PROCESSING.TIME, RUN.LENGTH
     14          AND STOP.TIME AS REAL VARIABLES
     15    DEFINE NO.CPU AND MAX.MEMORY AS INTEGER VARIABLES
     16    DEFINE MAX.MEMORY.QUEUE TO MEAN 1MAX.MEMORY.QUEUE
     17
     18    ACCUMULATE CPU.UTILIZATION AS THE AVG OF N.X.CPU
     19    ACCUMULATE MEMORY.UTILIZATION AS THE AVERAGE
     20       OF N.X.MEMORY
     21    ACCUMULATE AVG.CPU.QUEUE AS THE AVG AND
     22       MAX.CPU.QUEUE AS THE MAXIMUM OF N.Q.CPU
     23    ACCUMULATE AVG.MEMORY.QUEUE AS THE AVG
     24       AND MAX.MEMORY.QUEUE AS THE MAXIMUM OF N.Q.MEMORY
     25   TALLY AVG.JOB.TIME AS THE AVERAGE AND NO.JOBS.PROCESSED
     AS
     26          THE NUMBER OF JOB.DELAY.TIME
     27
     28    DEFINE HOURS TO MEAN UNITS
     29
     30 END ''PREAMBLE
```

```
                                                                PAGE 2
CACI SIMSCRIPT II.5 (R) v2.0                  6/26/1997 15:23:42

1 PROCESS GENERATOR
2
3          UNTIL TIME.V >= STOP.TIME
4          DO
5            ACTIVATE A JOB NOW
6            LET JB.PRIORITY.. = RANDI.F(1,10,1)
7          LET JB.MEMORY.REQUIREMENT.. = RANDI.F(1,MAX.MEMORY,2)
8          WAIT EXPONENTIAL.F(SMALL.JOB.INTERARRIVAL.TIME,3) MINUTES
9          LOOP
10
11 END
```

```
                                                                PAGE 3
CACI SIMSCRIPT II.5 (R) v2.0                  6/26/1997 15:23:42

1 PROCESS JOB
2
3     DEFINE ARRIVAL.TIME AND PROCESSING.TIME
4       AS REAL VARIABLES
5     IF PROCESS IS EXTERNAL
6       READ JB.PRIORITY..,JB.MEMORY.REQUIREMENT.. AND
7           PROCESSING.TIME
8     ELSE
9       LET PROCESSING.TIME = MIN.F(EXPONENTIAL.F
10          (MEAN.SMALL.JOB.PROCESSING.TIME,4),2 *
11          MEAN.SMALL.JOB.PROCESSING.TIME)
12    ALWAYS
13    LET ARRIVAL.TIME = TIME.V
14    REQUEST JB.MEMORY.REQUIREMENT.. UNITS OF MEMORY(1)
15       WITH PRIORITY JB.PRIORITY..
16    REQUEST 1 CPU(1) WITH PRIORITY JB.PRIORITY..
17    WORK PROCESSING.TIME MINUTES
18    RELINQUISH JB.MEMORY.REQUIREMENT.. UNITS OF MEMORY(1)
19    RELINQUISH 1 CPU(1)
20    LET JOB.DELAY.TIME = TIME.V - ARRIVAL.TIME
21
22 END
```

```
                                                                    4
CACI SIMSCRIPT II.5 (R) v2.0                  6/26/1997 15:23:42

1 MAIN
2
3     WRITE AS /, "A COMPUTER CENTER STUDY", /, /
4
5     Open unit 1 for input
6
7     LET HOURS.V = 1
8     CREATE EVERY CPU(1) AND MEMORY(1)
9     Let U.CPU(1) = 1
```

```
10    Let U.MEMORY(1) = 6
11    LET NO.CPU = U.CPU(1)
12    LET MAX.MEMORY = U.MEMORY(1)
13
14    Let SMALL.JOB.INTERARRIVAL.TIME = 2.0
15    Let MEAN.SMALL.JOB.PROCESSING.TIME = 0.8
16    Let RUN.LENGTH = 12.0
17    LET STOP.TIME = RUN.LENGTH / HOURS.V
18
19    PRINT 6 LINES WITH U.CPU(1), U.MEMORY(1),
20       60/SMALL.JOB.INTERARRIVAL.TIME,
21       MEAN.SMALL.JOB.PROCESSING.TIME AND RUN.LENGTH THUS
                A C O M P U T E R C E N T E R S T U D Y
NO. OF CPU'S     ** STORAGE AVAILABLE ****
SMALL JOBS ARRIVE AT THE RATE OF *** / HOUR
      AND HAVE A MEAN PROCESSING TIME OF ***.*** SECONDS
LARGE JOBS ARE SUPPLIED AS EXTERNAL DATA
THE SIMULATION PERIOD IS    **.** HOURS
28
29    ACTIVATE A GENERATOR NOW
30    ACTIVATE A STOP.SIM IN STOP.TIME HOURS
31    START SIMULATION
32
33 END ''MAIN
                                                        5
CACI SIMSCRIPT II.5 (R) v2.0                 6/26/1997 15:23:42

1 PROCESS STOP.SIM
2
3     SKIP 6 LINES
4   PRINT 9 LINES WITH TIME.V, CPU.UTILIZATION(1)*100/NO.CPU,
5           MEMORY.UTILIZATION(1)*100/MAX.MEMORY,
6           AVG.MEMORY.QUEUE(1), MAX.MEMORY.QUEUE(1),
7           AVG.CPU.QUEUE(1), MAX.CPU.QUEUE(1),
8           NO.JOBS.PROCESSED AND AVG.JOB.TIME * MINUTES.V
9           THUS
A F T E R **.** HOURS
THE CPU UTILIZATION WAS           *.** %
THE MEMORY UTILIZATION WAS        *.** %
THE AVG QUEUE FOR MEMORY WAS      *.** JOBS
THE MAX QUEUE FOR MEMORY WAS      *.** JOBS
THE AVG QUEUE FOR A CPU WAS  *.** JOBS
THE MAX QUEUE FOR A CPU WAS  *.** JOBS
THE TOTAL NUMBER OF JOBS COMPLETED WAS  ***
WITH AN AVERAGE PROCESSING TIME OF *.*** MINUTES
19
20 STOP
21
22 END
```

```
% cat SIMU01
JOB 1.00 3 1 5.00 *
JOB 2.46 1 2 7.00 *
JOB 3.78 3 3 10.00 *
JOB 9.28 2 2 30.00 *
JOB 10.48 1 4 40.00 *
JOB 24.22 1 5 60.00 *

% a.out

A COMPUTER CENTER STUDY

          A C O M P U T E R C E N T E R S T U D Y
      NO. OF CPU'S 1 STORAGE AVAILABLE 6
      SMALL JOBS ARRIVE AT THE RATE OF 30 / HOUR
         AND HAVE A MEAN PROCESSING TIME OF .800 SECONDS
      LARGE JOBS ARE SUPPLIED AS EXTERNAL DATA
      THE SIMULATION PERIOD IS 12.00 HOURS

A F T E R  12.00 HOURS
THE CPU UTILIZATION WAS           47.74 %
THE MEMORY UTILIZATION WAS        10.39 %
THE AVG QUEUE FOR MEMORY WAS      1.16 JOBS
THE MAX QUEUE FOR MEMORY WAS      19.00 JOBS
THE AVG QUEUE FOR A CPU WAS  .15 JOBS
THE MAX QUEUE FOR A CPU WAS  2.00 JOBS
THE TOTAL NUMBER OF JOBS COMPLETED WAS  364
WITH AN AVERAGE PROCESSING TIME OF      3.527 MINUTES
%
```

3. SIMSCRIPT II.5 Language Considerations

Some features of the SIMSCRIPT II.5 programming language vary from one implementation to another. This chapter describes implementation-specific features of
UNIX SIMSCRIPT II.5.

3.1 Input and Output

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

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

EXPRESSION1 specifies the unit number. If **input** is specified, the **unit** may appear in **use for input** statements. If **output** is specified, the unit may appear in **use for output** statements. If both **input** and **output** are specified, the **unit** may appear in both **use for input** statements and **use for output** statements. However, it is necessary to execute a **rewind** statement before reading from an output file or writing to an input file since the intermingling of I/O operations is not allowed.

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

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

Expression2 specifies the size of records in bytes. If the **recordsize** phrase is omitted, the size of records is assumed to be 80. For binary files, this is the actual length of each record. For files with variable length records, this is the maximum length of a record. Note that the "newline" character is not counted as part of the record length. Examples are:

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

Normally, if a file cannot be opened for some reason, such as the file does not exist or the

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

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

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

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

```
open UNIT-NUMBER for input and output
```

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

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

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

```
close [ unit ] EXPRESSION1
```

where **EXPRESSION1** specifies the unit number.
If the current input unit is closed, unit 5 becomes the current input unit. If the current output unit is closed, unit 6 becomes the current output unit.

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

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

example, to produce paginated output on unit 1, with 60 lines per page, specify:

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

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

3.2 Modes

The following modes are supported:

`Alpha`	An 8-bit unsigned integer used to store an ASCII character code (0 to 255)
`Integer2`	A 16-bit unsigned integer (0 to 65535)
`Signed integer2`	A 16-bit signed integer (-32768 to +32767)
`Integer4`	A 32-bit unsigned integer.
`Signed Integer4`	A 32-bit signed integer.
`Integer`	A signed integer 32 or 64 bits.
`Real`	A floating-point number of 32 bits
`Double`	A floating-point number of 64 bits
`Pointer`	An address
`Subprogram`	An address of a routine
`Text`	An address of a character string

For the 64-bit SIMSCRIPT III release, modes *integer*, *pointer*, *subprogram* and *text* are all 64-bits. 64-bit SIMSCRIPT users can use the *integer4* mode to get a 32-bit integer.

3.3 Non-SIMSCRIPT Routines

This section illustrates how a SIMSCRIPT II.5 program can call a routine written in C or FORTRAN.

3.3.1 Calling C Routines

Suppose we wish to call a subroutine named **Sub,** which is written in C and has two arguments:

```
void Sub(long inarg, long *outarg);
```

The first argument is an input to the subroutine, and the second argument is an output. The subroutine must be declared in the preamble:

```
define Sub as a nonsimscript routine given
      1 integer argument yielding
      1 integer argument
```

When calling this subroutine, the first argument should evaluate to `integer` since this is the SIMSCRIPT III mode, which corresponds to the C type **long**. The second argument should be passed using the **yielding** keyword. For example:

```
define IN.ARG, OUT.ARG as integer variables
write as "Enter the input value:", /
read IN.ARG
call Sub(IN.ARG) yielding OUT.ARG
write OUT.ARG as "The output value is ", I 10, /
```

Note that the "C" nonsimscript names are case sensitive. They must be declared in the preamble and used in the program with the correct case. This is true even if the **–s** option is not applied to simc. Also, if the arguments to the nonsimscript routine are prototyped, arguments will be automatically converted to the correct mode at the time the routine is called. Real and double arguments are rounded, not truncated. For example:

```
define IN.ARG as a double variable
define OUT.ARG as an integer variable
Let IN.ARG = 37.6
call Sub(IN.ARG) yielding OUT.ARG  /~ passes 38 to Sub ~/
```

Suppose we wish to call a function named **FUNC,** which is written in C and has one argument:

```
long func(double inarg);
```

The declaration of the function in the preamble specifies the mode of the function:

```
define func as an integer nonsimscript function
      given 1 double argument
```

Here is an example of a call to this function:

```
define IN.ARG as a double variable
define RESULT as an integer variable

write as "Enter the input value:", /
read IN.ARG
let RESULT = func(IN.ARG)
write RESULT as "The function result is ", I 10, /
```

It is very important that the SIMSCRIPT III mode of each argument and function matches its C type. Here is a list of C types and the corresponding SIMSCRIPT III modes:

unsigned char	`alpha`
unsigned short	`integer2`
short	`signed integer2`
long	`integer  (32-bit Linux,Windows, 64 bit Linux)`
long long	`integer  (64-bit Windows)`
float	`real`
double	`double`
char *	`pointer`

If an argument is a pointer to a null-terminated character string, you can pass a `text` value. You should not however "yield" or return a value of type "text".

3.3.2 Calling FORTRAN Routines

Suppose we wish to call a subroutine named **SUB, which** is written in FORTRAN and has two arguments:

```
subroutine SUB(inarg,outarg)
integer inarg
integer outarg
```

The first argument is an input to the subroutine, and the second argument is an output. The subroutine must be declared in the preamble:

```
define SUB as a fortran routine
```

Unlike SIMSCRIPT II.5 and C, FORTRAN passes arguments by reference, i.e., the address of the argument is passed rather than its value. The compiler for all routines declared as FORTRAN routines does this automatically.

```
write as "Enter the input value:", /
read IN.ARG
call SUB(IN.ARG, OUT.ARG)
write OUT.ARG as "The output value is ", I 10, /
```

Suppose we wish to call a function named **FUNC,** which is written in FORTRAN and has one argument:

```
integer function func(inarg)
double precision inarg
```

The declaration of the function in the preamble specifies the mode of the function:

```
define FUNC as an integer fortran function
```

Here is an example of a call to this function:

```
write as "Enter the input value:", /
```

```
read IN.ARG
let RESULT = FUNC(IN.ARG)
write RESULT as "The function result is ", I 10, /
```

It is very important that the SIMSCRIPT II.5 mode of each argument and function matches its FORTRAN type. Here is a list of FORTRAN types and the corresponding SIMSCRIPT II.5 modes:

```
integer*2        signed integer2
integer               integer
logical               integer
real                  real
double precision      double
```

Calling a FORTRAN routine that returns a **real** or uses **real** arguments results in a special case. Unlike SIMSCRIPT III and C, which interpret **real/float** function results and assignments as 64-bit values, FORTRAN uses a 32-bit value. To obtain this value within a SIMSCRIPT II.5 program, it is necessary to declare the function not as **real** but as **integer** and then "equivalence" an **integer** and **real** array to interpret the value as **real**. For example, suppose we wish to call a function named **RFUNC, which** is written in FORTRAN and has one argument:

```
real function rfunc(inarg)
real inarg
```

Declare the function in the preamble as follows:

```
define RFUNC as an integer fortran function
```

To call the function:

```
define IRESULT as a 1-dim integer array
define RRESULT as a 1-dim real array

write as "Enter the input value:", /
read IN.ARG
reserve IRESULT(*) as 1
let IRESULT(1) = RFUNC(IN.ARG)
let RRESULT(*) = IRESULT(*)
write RRESULT(1) as "The function result is", D(10,3),/
```

4. SimDebug Symbolic Debugger

SimDebug is the SIMSCRIPT III Symbolic Debugger. In contrast to other debuggers that are separate programs, this debugger is built into the language. Simply compile the modules you want to debug with debugging and then run your program with the command line argument **-debug**. This will bring up the SimDebug dialog before the program starts. Since the debugger is "always there," any runtime error will also put you into the SimDebug dialog, where you can examine the stack, local and global variables, etc. SimDebug's features include:

- single stepping
- setting breakpoints
- viewing stack and global variables
- displaying temporary and permanent entities
- displaying sets and arrays
- displaying system variables, I/O and memory statistics
- displaying the I/O buffer
- displaying simulation status
- changing variables and attribute values
- stopping at a certain simulation time
- command/dialog logging
- and a lot more.

This chapter describes how to use SimDebug. We first describe how to compile for and run SimDebug. Then we will give you a quick tour that introduces the usage and major features of SimDebug in the style of a tutorial. A detailed alphabetical description of all the SimDebug commands is given in paragraph 4.3. Some advanced topics related to SimDebug are given in paragraph 4.4.

4.1 Compiling for Debug and Invoking SimDebug

4.1.1 Compiling for Debug

This paragraph describes how to compile for debugging using the SIMSCRIPT II.5.
There are three levels of debugging support that can be selected for compilation. The debugging level is controlled through a command line option to **simc**. The three levels of debugging are none, traceback only, and full debug. The selected debugging level applies to all routines in the modules supplied to that invocation of **simc**. The option to use is **-d** for full debug. If no options are specified you will get "traceback only". The **–O** flag will elimiate all debugging.

To be able to look at entities, system variables and global variables you must compile both public and private `PREAMBLE` with debugging, i.e. with the **-d** option.

You should not mix the debug and optimization flags in the **simc** call. That is, do not specify **-d** and **-O** at the same time, since this can lead to erroneous output from SimDebug.

4.1.2 Invoking SimDebug

To invoke SimDebug simply invoke your program with the command line option **-debug**. This option will only be recognized by SimDebug and will not be visible to your SIMSCRIPT III program as a command line argument. The position of the **-debug** option on the command line is irrelevant.

SimDebug Dialog

When you invoke your program with **-debug** you will be put into the SimDebug dialog. Here you can examine the source, set breakpoints, and start your program. When you do not specify the **-debug** option, your program will run as usual without any interference from SimDebug.

At the beginning of the SimDebug dialog (whether you invoked it with **-debug** or entered the dialog through a runtime error) SimDebug looks for a file **simdebug.ini** in the current directory. If this file exists, it is loaded as a SimDebug command file (see **READCMDS**). This way you can easily customize the setup and initialization of SimDebug.

SimDebug will always show a **SimDebug>** prompt when it is ready for a new command.

Runtime Errors

Even when you do not compile your program with the **-d** option and you do not call your program with **-debug,** when SIMSCRIPT detects a run-time error, you are put into the SimDebug dialog. You can then perform all SimDebug commands to inspect your program, with one exception: You cannot continue execution from floating point errors, segment violations and bus errors! Keep in mind, that to detect more runtime errors, you should compile with the **–C** option to enable subscript checking.

When you do not want to enter the SimDebug dialog in case of a runtime error, you can set the global system variable **batchtrace.v = 1**. This results in the traceback being written to **simerr.trc**, after which the program exits. This is a change from the behavior of the previous release 1.9 where the traceback would always be output on the current output device (according to **write.v**). However, using the `trace` statement in your program will still write the traceback to the current output unit (**write.v**).

Instead of setting **batchtrace.v = 1** in your program, you can also call your program with the command line argument **-batchtrace**. This automatically sets **batchtrace.v=1**. As with **-debug**, this command line argument will not be seen by your SIMSCRIPT program.

If you want your program to exhibit the old traceback behavior and have a runtime error, just write a traceback and then exit. Compile your program with no options then execute with the option **-batchtrace**. The traceback will be written to **simerr.trc.** For further information see paragraph 4.4.1.

Interrupting Running Programs

You can interrupt a running program by pressing **Ctrl-C** (or the INTERRUPT key combination defined for your system). This will put you in the SimDebug dialog where the program is currently executing. This is very useful to detect endless loops or recursions. See the **Ctrl-C** command in the command reference paragraph for more details.

Text Input/Output

On UNIX platforms, the SimDebug dialog runs in the terminal window from which the program was started. This means that the program's input/output using units 5,6, or 98 will be intermixed with the SimDebug dialog, as you would expect.

However, when you redirect input or output when calling your program, this will not affect the dialog of SimDebug. Thus, even if you type **prog -debug < infile > outfile** the SimDebug dialog will still be connected to your terminal (window). This allows you to debug programs that read a lot of input from unit 5 (standard-in) without the input interfering with the SimDebug dialog.

4.2 A Quick Tour of SimDebug

In this paragraph we will introduce SimDebug by example. In the following tutorial user input is shown in **bold face Courier**, and SimDebug `output` and example `program source` are shown in the `regular Courier font`. The SimDebug dialog is indented, our comments appear in between the dialog segments in *italic*.

We assume that we have recompiled our entire program using the **-d** compiler option (including the `PREAMBLE` so that we can see the attributes of entities).

4.2.1 Tour 1: Showing the Stack and Variables

Our program contains a runtime error. When the error occurs, SimDebug shows the error message, **floating point error.** The meaning of the minor error code is machine specific; here it means division by zero.

```
OS-prompt$ tst -debug
ERROR: Floating point error. Minor error code = 200
----- R1 (sample.sim) ------------------------------------ Line =
39
```

```
. 39> write B/A as I 4,/
```

SimDebug shows that the error occurred in routine **R1**, source file **sample.sim**, at line **39**. The actual source code at that line is shown on the next line. To see a traceback of the routine call hierarchy, type **t**.

```
SimDebug> t
==================== call stack =====================

----- R1 (sample.sim) ---------------------------------Line = 39
Given Arguments:
   A             =          0          (Integer)        00000000]
   B             =          2          (Integer)        [00000002]
Local Variables:
   I             =          5          (Integer)        [00000005]
   J             =          1          (Integer)        [00000001]
----- R1 (sample.sim) ---------------------------------Line = 36
Given Arguments:
   A             =          1          (Integer)        [00000001]
   B             =          2          (Integer)        [00000002]
Local Variables:
   I             =          5          (Integer)        [00000005]
   J             =          1          (Integer)        [00000001]
----- MAIN (sample.sim) -------------------------------Line = 62
Local Variables:
#1 AARR          = (null)              (Pointer)
   I             =          6          (Integer)        [00000006]
#2 IARR          = 00060548       ( 1-dim Integer array)
#3 IARR2         = 0005C268       ( 2-dim Pointer array)
#4 LE            = 0005C3E8       (Ptr--> class LISTELEM)
```

We now see that **R1** is recursive and that **A** is **0**. Obviously we tried to divide by zero.
A few more comments on the traceback output: The types of variables distinguished in the output for each routine are: Given Arguments, Yielded Arguments, Local Variables, and Saved Local Variables. Given and yielded arguments appear in the order in which they were defined in the routine source code. All other variables (including the global variables) appear in alphabetical order. Each line that shows a variable has basically the same format:

VarName	Variable name
Value	The value. Pointers are shown as 8 hex digits.
Mode	Mode information for that variable. For pointers, SimDebug shows where it points to (which kind of entity, array etc.). For integers we also show the value again as hex in [].

To see the global variables, type **glob**. They are ordered by name and appear in the same format as the variables in the traceback.

```
SimDebug> glob

#1    DSPLY.E     = (null)              (Pointer)
#2    F.LISTSET   = 0005C368       (Ptr--> class LISTELEM)
      GLOBALD     =     0.              (Double)
```

```
      GLOBALI     =     0                (Integer) [00000000]
#3    LISTELEM    = (null)               (Pointer)
#4    L.LISTSET   = 0005C3E8       (Ptr--> class LISTELEM)
N.LISTSET         =          5          (Integer2) [00000005]
```

Again, we want to see where we are. The **w** command shows us the context of the current line (default ± 5 lines) with a "=>" in front of the current line.

```
SimDebug> w
----- R1 (sample.sim) --------------------------------- Line = 39
.     34    J = A-B
.     35    if A > 0
.     36      call R1(A-1, B)
.     37    else
.     38      write as "B/A = "
=>    39      write B/A as I 4,/
. 40        endif
. 41  end
```

All these commands still apply to the current routine or the current frame in the traceback (called hierarchy). If we want to see where we are in the routine that called this `R1`, we must move the current frame one level down ("Top of stack" is the last routine called, "Bottom of stack" is `MAIN`). The **dn** command moves the current frame one level down and SimDebug shows us the current line on that level. Then we use **tc** to get a traceback of only the current routine frame which is now **R1** at stack level **2**. Note that in this frame, **A=1**. With **pv** we can ask for only one variable. When it is in the current routine, that value is printed. Otherwise, SimDebug looks at the global variables. Before actually printing the line with the variable name, value and type, **pv** first prints whether the found variable is a given or yielded argument, and whether it is a local, local saved, or a global variable.

```
SimDebug> dn
----- R1 (sample.sim) ---------------------------------Line = 36
   36>.     call R1(A-1, B)
SimDebug> tc
----- R1 (sample.sim) ---------------------------------Line = 36
Given Arguments:
   A       =          1     (Integer)        [00000001]
   B       =          2     (Integer)        [00000002]
Local Variables:
   I       =          5     (Integer)        [00000005]
   J       =          1     (Integer)        [00000001]
SimDebug> pv A
Given Argument:
   A       =          1     (Integer)        [00000001]
```

In large programs, variable names as well as routine names are generally quite long. To avoid having to type in the whole variable name, you can enter just the first few letters.
SimDebug *matches* your input with the defined variables. When your input uniquely identifies a certain variable, it will be printed as usual. When you enter **pv G*** and there are several variables (locals or globals) that begin with **G**, you will be offered a list of matches from which you can select by number. In the same way, you can select from all variables that

end with a certain *suffix* by using **pv *suffix**. When we want to use the input as a *prefix* the "*****" is optional. **pv** always looks in the current frame first, and then at global variables to find variables with a certain name/pattern.

```
SimDebug> pv g*
---- Matching GLOBAL variable names ----
   1  GLOBALD
   2  GLOBALI
---> Select variable by number (0=none) > 2
Global Variable:
   GLOBALI  =          0     (Integer)        [00000000]
SimDebug> pv li
#1 LISTELEM           = (null) (Pointer)
SimDebug> pv *set
---- Matching GLOBAL variable names ----
   1  F.LISTSET
   2  L.LISTSET
   3  N.LISTSET
---> Select variable by number (0=none) > 3
Global Variable:
   N.LISTSET          =     5               (Integer2)
[00000005]
```

In the same way you can restrict the output from the **GLOB** command with a **prefix*** or a ***suffix** argument. The following example ends our first tour:

```
SimDebug> glob g
   GLOBALD            =     0.              (Double)
   GLOBALI            =     0               (Integer) [00000000]
SimDebug> glob *set
#1    F.LISTSET        = 0005C368  Ptr--> class LISTELEM)
#2    L.LISTSET        = 0005C3E8       Ptr--> class LISTELEM)
N.LISTSET              = 5                   (Integer2) [00000005]
SimDebug> quit

Leaving SSDB ...
OS-prompt$
```

4.2.2 Tour 2: Breakpoints and Single Stepping

We are now going to a different program that will illustrate the use of breakpoints, single stepping and SimDebug's advanced pointer handling features. This program creates a few entities and arrays. We call our program with **-debug** so that we are immediately put into the SimDebug dialog. With the **lr** command we get a list of the routines in the program that were compiled with debugging and their line number range. You can use wildcards at the beginning and end of a routine name argument in **lr** in the same way as with variable names. Note how **R2**, a *left routine*, gets displayed. In these routines we can single step, set breakpoints, etc. With **ls** we can look at the source of the routine `main`.

A "**.**" in front of a source line means that this line is executable and that you can set a breakpoint there.

```
OS-prompt$ tst -debug
SimDebug (SIMSCRIPT Symbolic Debugger) Version 1.0

SimDebug> lr { lists all routines compiled with debug or trace }
      MAIN              (sample.sim      :          44- 64)
      R1                (sample.sim      :          29- 41)
      R2-L              (rtns.sim        :          1- 32)
SimDebug> lr r{ lists all routines that begin with an "R" }
   R1                   (sample.sim      :          29- 41)
   R2-L                 (rtns.sim        :          1- 32)
SimDebug> ls m{ lists the (only) routine that begins with "M" }

---- MAIN -----------------------------------(main.sim: 44-64)
. 44 main
  45  define LE as pointer variable
  46  define IARR as 1-dim integer array
  47  define AARR as 1-dim alpha array
  48  define IARR2 as 2-dim integer array
  49  define I as integer variable
  50
. 51        reserve IARR as 10
. 52        reserve IARR2 as 5 by 5
  53
. 54        for I = 1 to 5
  55        do
. 56          create a LISTELEM called LE
. 57          ATTRI(LE) = I
. 58          ATTRP(LE) = IARR2(I,*)
. 59          file LE in LISTSET
. 60        loop
  61
. 62        call R1(3,2)
   63
. 64 end
```

We can start our program simply by invoking the **s** command (single step). But instead we will set a breakpoint on the line where a new entity gets created and where **R1** gets called. With **lb** we get a list of the currently set breakpoints. With **r** we start the program which runs until it hits the first breakpoint. A message is printed and the source line that will be executed next is shown.

Note: The current line in SimDebug is the line that gets executed next. Thus, a breakpoint at a certain line stops execution *before* that line.

We also set a breakpoint at the beginning of **R2**. Note that SimDebug asks for missing argument information.

```
SimDebug> sb main 56
```

```
   SimDebug> sb m* 62{ "M" uniquely identifies MAIN, the "*" is optional}
   SimDebug> sb r*
----- List of matching routines -----
      1 R1
      2 R2-L
   Enter routine by number > 1
   Enter line number > 1
   *** No executable source code at that line. Used line 4 instead.
   SimDebug> lb
------- List of Breakpoints --------
      1 MAIN @ line 56
      2 MAIN @ line 62
   SimDebug> r

   BREAK: User breakpoint
----- MAIN (sample.sim) ---------------------------------
Line = 56
  56># create a LISTELEM called LE
```

After reaching the breakpoint, we single step through the program for a while. After each **s** command, SimDebug shows the new 'current line' (that will be executed next). Since an empty command repeats the last command we can simply press **Return** to repeat the singlestep. If a line contains a routine call, **s** will step *into* the routine, whereas **n** will step *over* the routine. After we have stepped enough, we use the **c** command to continue the program until the next breakpoint.

```
      SimDebug> s
      57 ATTRI(LE) = I
      SimDebug> { no input = repeat last command }
      58 ATTRP(LE) = IARR2(I,*)
      SimDebug>
      59 file LE in LISTSET
      SimDebug>
      60 loop
      SimDebug> c { continue execution }
      BREAK: User breakpoint
      ----- MAIN (sample.sim) ---------------------------------
      Line = 62
      #> 62 call R1(3,2)
      SimDebug> ls { lists source of 'current routine' }

. 44 main
45 define LE as pointer variable
46 define IARR as 1-dim integer array
47 define AARR as 1-dim alpha array
48 define IARR2 as 2-dim integer array
49 define I as integer variable
50
. 51 reserve IARR as 10
. 52 reserve IARR2 as 5 by 5
53
. 54 for I = 1 to 5
```

```
55 do
# 56 create a LISTELEM called LE
. 57 ATTRI(LE) = I
. 58 ATTRP(LE) = IARR2(I,*)
. 59 file LE in LISTSET
. 60 loop
61
#> 62 call R1(3,2)
63
. 64 end
```

Conditional Breakpoints: You can programmatically set conditional breakpoints on arbitrarily complex conditions by calling SimDebug itself! See paragraph 4.4.6.

4.2.3 Tour 3: Pointer Handling: Entity / Set Display

Now the set is created and we are ready to look at the set and the entities. The set **LISTSET**was declared in the `PREAMBLE` as 'owned by the system'. This is why the fields for the set**F.LISTSET, L.LISTSET** and **N.LISTSET** are global variables. We first display theglobal variables to see the variable **F.LISTSET**, which holds the pointer to the *first* element in the set. Once we are in the set, we follow the pointers using **fp** (follow pointer debugger command) along **S.LISTSET** (successor) to get to the next elements. Observe that the attribute **ATTRI** is **1,2,3...** and that the **ATTRP** points to the different arrays as assigned in the loop.

```
SimDebug> glob
#1    DSPLY.E    = (null)              (Pointer)
#2    F.LISTSET  = 0005C368       Ptr--> class LISTELEM)
      GLOBALD    = 0.                  (Double)
      GLOBALI    = 0                   (Integer)        [00000000]
#3    LISTELEM   = (null)              (Pointer)
#4    L.LISTSET  = 0005C3E8       (Ptr--> class LISTELEM)
      N.LISTSET  = 5                   (Integer2) [00000005]

SimDebug> fp #2
------- Entity #2: 0005C368 (class LISTELEM) ----------
      ATTRI      = 1                   (Integer)        [00000001]
      ATTRA      = 00 (hex)       (Alpha)
#1    ATTRP      = 0005C2C8       (Ptr--> Array (5) of Integer)
#2    S.LISTSET  = 0005C388       (Ptr--> class LISTELEM)
#3    P.LISTSET  = (null)              (Pointer)
      M.LISTSET  = 1                   (Integer2) [00000001]

SimDebug> fp #2
------- Entity #2: 0005C388 (class LISTELEM) ----------
      ATTRI      = 2                   (Integer)        [00000002]
      ATTRA      = 00 (hex)       (Alpha)
#1    ATTRP      = 0005C2E8       (Ptr--> Array (5) of Integer)
#2    S.LISTSET  = 0005C3A8       (Ptr--> class LISTELEM)
#3    P.LISTSET  = 0005C368       (Ptr--> class LISTELEM)
      M.LISTSET  = 1                   (Integer2) [00000001]
```

```
SimDebug> {Pressing Return repeats last FP command. Step through set }
------- Entity #2: 0005C3A8 (class LISTELEM) ----------
      ATTRI      = 3                    (Integer) [00000003]
      ATTRA      = 00 (hex)        (Alpha)
#1    ATTRP      = 0005C308        (Ptr--> Array (5) of Integer)
#2    S.LISTSET  = 0005C3C8        (Ptr--> class LISTELEM)
#3    P.LISTSET  = 0005C388        (Ptr--> class LISTELEM)
      M.LISTSET  = 1                    (Integer2) [00000001]

SimDebug> fp #1 { "FP" knows how to interpret pointers ; this is IARR(3,*) }

      #1(1) = 0   [00000000]
      #1(2) = 0   [00000000]
      #1(3) = 0   [00000000]
      #1(4) = 0   [00000000]
      #1(5) = 0   [00000000]
```

This concludes our quick tour of SimDebug. All commands are fully documented in paragraph 4.3.

4.3 SimDebug Command Reference

The SimDebug commands and their options are listed below in alphabetical order. When commands have abbreviations, the abbreviations are given on the next lines below the command. To list each command with its optional arguments the following notation is employed:

CMD arg: Command names and keywords are shown in UPPER CASE, arguments are shown in lower case.

[. . .] Optional arguments are enclosed in square brackets.

a **|** b Alternatives are separated by the vertical slash.

For example, **LOG [CMDS|DIALOG|START|STOP|CLOSE]** means that the **LOG** command can have no argument, or can have one of the listed arguments. The notation **T [from [to]]** means that the command **T** (traceback) can have one or two optional arguments, **from** and **to**. Command names and keyword arguments are shown in UPPER CASE, arguments of commands are shown in lower case (e.g. **READCMDS cmdfile**).

Basic Syntax: Each SimDebug command consists of the **command name** followed by one or more arguments, each seperated from each other by one or more spaces. There are no parentheses and there is no nesting of expressions needed. SimDebug commands are **not case sensitive**. Except for file names, upper/lower case is irrelevant.

Missing Arguments: Whenever possible, SimDebug will ask you for a missing argument instead of issuing an error message.

Repeat Last Command: When you press **Return** (no command entered), the last command will be repeated. This is particularly useful for the **S, N** and **FP** commands.

Scrolling Output: The output of SimDebug will appear in the 'terminal window' from which you invoked your program. If your 'terminal window' does not allow scrolling back, you can set a parameter **SET SCROLLINES n** so that the output will pause after every **n** lines (press **Return** to continue).

Routine Names: Several SimDebug commands take routine names as arguments. You can type the routine name just as you use it in your program (e.g. **TACK.ORDER.QUEUE**). Upper/ lower case in routine names is irrelevant.

Variable Names: You may use wildcards, i.e. the "*****", when entering variable names, or may enter just the first few letters of the desired name. When the input matches several names you will be offered a list from which you can select the desired variable. Whenever SimDebug looks for a variable, it looks in the 'current frame' first (local variables on the stack), and when the specified variable is not found there, in the set of global variables.

List of SimDebug Commands:

#

Comment: The remainder of this line is discarded. This is useful for inserting comments in command files (see `READCMDS`).

?

Help: See `HELP` command.

BOT

Bottom: Set the 'current frame' to the bottom of the stack, i.e. to `MAIN`. See note on 'current frame' in the `DN` command.

BPDIS n

Breakpoint disable: Disables breakpoint **n** (**n** comes from the **LB** command).

BPEN n

Breakpoint enable: Enables breakpoint **n**. The **LB** command shows each defined breakpoint with a number that can be used for **BPEN, BPDIS** and **DB**.

BR rtnname

Break in Routine: Sets breakpoint on the first executable line of the given routine. Execution stops when the routine is entered.

BUF n

Show Buffer: Show the contents of the buffer of unit `n`. This can also be used to show the contents of **the buffer**, i.e. unit 99.

Ctrl-C (INTERRUPT key)

This command interrupts your running program and enters SimDebug so you can see where you are in the program's execution. The '**current routine**' is the currently executing routine.

INTERRUPT in no-debug routine: When you do not compile the current routine with debug, you will not be able to see the current line of execution or the local variables/ arguments. You will only see the routine name. An **s** (single step) command in a routine that was not compiled with debug will take you to the next line of code that was compiled with debug (this may be several levels up in the calling hierarchy).

INTERRUPT during simulation: When you press the INTERRUPT key while a simulation is running, SimDebug may report the current line as the line that contains the **start simulation** statement. This means that your program is in between the last and the next process/event. A single-step command **s** will take you into the next line of the next process when you compiled that process routine with debug.

C

Continue: Continues execution. When there is no breakpoint set in the 'execution path' the program runs until completion, until a runtime error occurs, or until you press **Ctrl-C** to interrupt the running program.

DB n

Delete Breakpoint: Deletes breakpoint **n** (n is defined from the **LB** command).

DM [addr [type [count]]]

Display Memory: For the rare cases where you might want to look at memory in an unstructured way (e.g. for non-SIMSCRIPT data), the **DM** command allows you to view areas of memory as Hex values (4 bytes each), as Integers (4 bytes), Reals (4 bytes), 4 Doubles (8 bytes) or 40 characters (1byte each). To display contiguous areas of memory, you can use **DM** in two ways: First with **DM addr type count**, you set the starting point, the type and the count of your memory display. Then, subsequent **DM** commands (with NO ARGUMENTS) will continue memory display where the previous display left off. The arguments are:

addr Starting address (in hex)

type Type of display of item: **H, P**: 4 bytes as hex, **I**: integer, **R**:real, **D**: double, **A**: alpha. Default is **H** = hex.

count Number of items to display per command (always 4 per line). For Alpha mode non-printable characters are shown as ".".

DN [n]

Down: Move 'current frame' **n** levels down (towards `MAIN`) in the stack. Default: **n = 1**. **Note:** The **current frame** is the routine being looked at in the call stack shown by the **T** traceback command. When you look at a certain variable with the **PV** command, you look first at the current frame, and then at global variables to find this variable. Thus, with **UP** and **DN** you can move the current frame to allow inspection (e.g. a certain instance of a recursive routine call).

ECHO arg1 arg2 ...

Echo: Echoes the words **arg1, arg2, ...** to the output. This is useful to output messages from within a command file.

EV

Event set: Prints information about the simulation, including the event set, the current simulation time, the current and next process etc. For each process/ event the time of the next scheduled process/event and of the last scheduled process/event of that class is shown with **pointer numbers** [**#n**] in brackets behind the times. Using these pointer numbers you can step through the event sets for each process/event type using the **FP** command. The event/process that is scheduled next is marked with a "*" behind the class number. When only one process is scheduled in a class, only the **time.a(First)** is printed (so you can easily tell that there is just one).

Entity in process.v: Process.v is a pointer to the process/event notice of the currently active process/event. For a process **'CUSTOMER'** the entity class will be **'CUSTOMER'**. This entity contains any user declared attributes as well as some internal attributes. *Never change any of the internal attributes!*

FP ptrvariable

FP ptrvalue

FP #n

Follow Pointer: With this command you can display the contents of an object that a pointer points to. This will generally be an entity, in which case the entity attributes are shown, or an array, in which case the array elements are shown. There are three varieties of the command:

FP ptrvariable: Ptrvariable is the name of a (local or global) pointer variable.

FP ptrvalue: Ptrvalue is a pointer value (in hex) taken from previous output.

FP #n: n is a pointer index**.** Whenever a pointer is shown as output from the **T, FP** or other commands, it is displayed with a prefix of the type **#n** where **n** is a running index. This way each pointer can be uniquely identified by **#n**. The running index **n** is 'restarted' by each command that displays a pointer value. Thus **#n** applies to the last displayed **#n**. Thus, with the **FP #n** command you can follow a previously displayed pointer. This is very useful for all data structures that employ pointers, such as lists, sets, your own graph structures etc.

Example: Walking through a set: To step through all elements of a set, simply type **FP #n** where **n** is the index of the pointer that represents **.setname** (pointer to first in set). The first displayed element will have a pointer field **S.setname** (to successor), say with index **#3**. Repeated commands **FP #3** will display one set member after another.

Temporary Entity Display: For temporary entities SimDebug shows the whole entity with all attributes. Packing (*/2, */4, bit packing, overlap) is fully supported.

To see just one field of an entity, type **FP entname attrname**.

Note on Destroyed Entities: Remember that when you destroy an entity, the pointer to that entity is still there. But the storage freed by the 'destroy' command will generally be reused immediately. Thus, a pointer variable that points to an entity might suddenly display "**Ptr --> Text ! Error !!**" in its mode field, or appear to point to a different entity class even though you did not touch that variable. This is especially noticeable for the *global process entities* that are deallocated when the corresponding process is suspended or terminated.

Note on Global System Variables: When global variables are listed you will also see several internal/ system variables that are implicitly defined by SIMSCRIPT II.5 (such as resources, temporary entities etc). Instead of hiding these values, SimDebug shows these internals since they are documented, (such as fields of resources, etc). However, you *should never change a variable that you did not create/define yourself.*

Printing Text Values: SimDebug shows only a few characters of the text in the normal **PV** output. To see the whole text, use **FP textvar**. See notes on the text display at the **FP** command. **Note on Integers Used as Pointers:** Since many SIMSCRIPT programs use **integer** variables to store `pointers` as well, SimDebug allows you to 'follow integers' as if they were pointers.

FPN ...

Like **FP,** but this command does not reset the pointer number counter. This allows you to keep the 'access handle' **#n** to the entity after you have displayed it. This is needed for the **SEV** command (set entity values). See the notes for the **SEV** command.

GL [pattern]

GLOB [pattern]

Globals: Prints a list of all global variables and their values (in alphabetical order). See the **T** command for a description of the output. **Pattern** can be **prefix** or **prefix*** which shows all variables that begin with the given prefix, or ***suffix** which shows all variables that end with the given suffix.

H

HELP [name]

HELP: Gives an overview (just the names) of all SimDebug commands. When **name** is given, SimDebug gives a more detailed description of the topic/command with that name. **Name** can be either a command name, or a topic name (such as 'breakpoints'). Both the command and topic names are given in the help overview.

IO

I/O Information: Shows information about the I/O status of your program, i.e. for each unit used whether it is input or output, which file is attached (if any), how many

records were read/written etc. Use the **BUF** command to look at buffer contents for units.

LB

List Breakpoints: Lists all currently defined breakpoints. Disabled breakpoints (see **BPEN, BPDIS**) appear in parentheses.

LOG [CMDS|DIALOG|STOP|START|CLOSE] [logfilename]

Command and Dialog Logging: You can have SimDebug write all commands or all of the dialog (commands and SimDebug output) to a log file. Command *and* dialog logging cannot be active at the same time (there is only one log file). The variants of the command are the only arguments listed:

(without argument) Show status of logging.

CMDS [logfilename] Start command logging. Default file: **cmdlog.log**

DIALOG [logfilename] Start dialog logging. Default file:**dialog.log**

STOP Stop current logging.

START Resume logging

CLOSE Close current log file. Allows you to start a new log (command or dialog).

When *command* logging is turned on, only the actual commands and not the **SimDebug>** prompts are put into the log file. As a special case, `LOG` *commands are* ***not*** *put into the command log* since you generally do not want them when repeating the command sequence. They are written to the dialog log, however.

When you press **Return** to repeat the last command, the full command name will still be written to the command/dialog log.

LR [rntname|prefix*|*suffix|ALL|NODEBUG]

List Routines: Lists the names of the routines in your program in the following order: `PREAMBLE, MAIN`, and then all others in alphabetical order.

LR List all user routines compiled with debug or trace.

LR ALL List all user routines (nodebug routines prefixed with **N**; routines compiled with **-g** are prefixed with **T**).

LR TRACE List all routines compiled with traceback (**-g**).

LR NODEBUG List all user routines that were **not** compiled with debug.

LR prefix*

List user routines that begin with **prefix** ("*****" is optional).

LR prefix*-L

Append **-L** after the "*" to see only left routines.

LR *suffix

List routines that end with a suffix (e.g. **LR *.CTRL**)

Note: Continuous variables will display as right and left routines. When you have a routine with the name **ALL, TRACE or NODEBUG**, you must use **ALL*, TRACE*, or NODEBUG*** to get the routine individually.

LS [rtnname [from [to]]]

List Source: Lists the source lines of the given routine. The default is to show the whole routine. Line numbers (for **from** and **to**) are given relative to the file (not relative to the routine beginning or the like).

When the program is active the **rtnname** can be omitted in which case the 'current routine' (the source of the current frame) is shown.

Source Listing Format: Each output line consists of four fields:

1. A "." for executable source lines (you can set breakpoints there) or a "#" when a breakpoint is set on that line
2. A ">" when this is the current line (of execution)
3. The source line number of the line (in the source file), and
4. The first 72 characters of the source line itself.

Note: Only the first 72 characters of a source line are printed so that all output fits on one line.

MEM

Memory Information: Shows memory statistics, such as how many entities of each type are currently created, and how many strings and arrays there are.

Note: Since string and array counters are for both SIMSCRIPT internal use and for user data, the numbers do not directly reflect your program's memory usage. Also, since SimDebug uses strings, the numbers will be higher when compiling with debug. A good way to find out if your program has a 'memory leak' is to write down the number of strings, arrays etc. at the beginning of the program, and then let it run for awhile. Interrupt the program with **Ctrl-C** and look again.

N [n]

Next: Execute the next **n** (default: 1) SIMSCRIPT source lines and then return to the SimDebug dialog. **N** steps over a routine call. This routine and all routines called from this command execute until you are returned to the SimDebug dialog. Unless, of

course, there is a breakpoint set somewhere in the called routines.

Also, see comment on "**Specifying Repeat index n**" in the **S** command.

Context Switch: When a context switch occurs during a **N** or **S** or **SR** command, a message is printed accordingly.

O

Step out: Step out of the existing routine. Continue executing until the current routine or method returns. At this point the debugger will become active again.

PAV arrvarname [selvec]

Print array variable: With this command you can display all or part of a multi-dimensional array or parts thereof. **Arrvarname** must be an array variable name and the whole array is printed by default. **Selvec** is the 'selection vector' that allows you to limit the output. It consists of several elements with the following meanings:

n Show only this element from the current dimension

***** Show all elements of this dimension

+ Stop display at this dimension.

A few examples will clarify this command. Assume **ARR3I** is a 3-dimensional integer array, reserved as (5,5,5). Then:

`PAV ARR3I 1` Prints all elements of **ARR3I(1,*,*)** (25 integers)

`PAV ARR3I 2 3` Prints **ARR3I(2,3,*)** (5 integers)

`PAV ARR3I * 4 5`
Prints **ARR3I(1,4,5), (2,4,5), (3,4,5), ...** (5 integers)

`PAV ARR3I 3 +` Prints 5 pointers to the integer arrays of the last dimension, ie. **(3,1,*), (3,2,*), (3,3,*),...**

Equivalencing: An array may be defined and reserved as a 5-by-5 integer array. But if you assign this pointer to an array variable of mode "**2-dim alpha array**" you can look at the data as alphas. The **PAV** command uses the mode of the given array variable (**arrvarname**) to determine how to interpret the data.

PDV arrvarname [selvec]

PDV ptrvariable [selvec]

PDV ptrvalue [selvec]

Print descriptor variable: Same as **PAV** except that the array is printed from the information contained in the array descriptors. That is, the array will be printed with the mode it was first reserved as.

PT textvar|textptr

Print text values in full: This command prints the whole text of a text variable or a pointer pointing to a text value. This command is needed since **PV**only prints the first few characters of a text string. The whole text value isprinted with string quotes around it and a "-" at the end of each line whenthe text continues on the next line. Thus, on an 80-character line you can see77 characters of text (with two string quotes around it, and a "-" at the end).

Text attributes: If the text you want to see in full length is an attribute of an entity, you can use the address of the text that is given with the attribute output as an argument for **FP**. The same holds for arrays of text pointers.

PV varname

Print Variable: Prints the value and type information for the variable `varname`. SimDebug *first searches the current frame*, and if **varname** is not defined there, *then the global variables* for **varname**. As described at the beginning of this paragraph, you can use wildcards to specify the variable name (**prefix, prefix*, *suffix**). When several variables match, a selection list is shown.

Format of output: Before printing the line with the actual variable, SimDebug prints the type of variable it found: Given Argument, YieldingArgument, Local Variable, Local Saved Variable, or Global Variable.

Then each line follows basically the same format:

ptrnum varname = **value** `(mode information)`

where the fields contain:

ptrnum For pointers: The **#n** entries for the **FP** (follow pointer) command.

varname The variable name.

value The value. Text is shown to the extent that it fits in the space, where internal string quotes are not doubled (i.e. a string containing a single string quote is printed as """). Integers and alpha characters are printed as usual, where nonprintable **alpha** values are also printed in hex. For **real**s and **double**s you can define the output format with **SET OREALF** (see `SET` command). Pointers and subprogram variables are shown in hex.

mode info

Mode information. For integers, the value in "[]" in hex is appended. For pointers, pointer destination information (e.g. entity class, array type) is shown. ***** Bad pointer ***** means that this is an illegal address, i.e. an address that would cause

a segment violation if it were used. For subprogram variables the subprogram name is shown. Use **SET EXTINFO 0** when you do not want this extended information for pointers.

Array mode info

Normally, arrays mode information is shown as the array was declared in the program, e.g. "**2-dim integer array**". With the **SET** parameter **SHOWARRAYPTRS** you can choose to see the internal structure of the arrays, instead. That is, you can see the pointer structure (arrays of pointers) that makes up multi-dimensional arrays. This is necessary when dealing with ragged arrays or assigning array fragments.

Printing Text Variables: The normal output of **PV** and **T** shows just the first 10 characters of the text. If you want to see the whole text, use **PT textvar**.

QUIT

Quit: Quit/exit from SimDebug. All open log files will be closed. Synonyms are: **Q, EXIT, END, BYE**.

R

Run: Run/start your program from the beginning. You cannot start your program 'in the middle', or restart the program with the **R** command. To restart for debugging you must call your program again with **-debug**.

READCMDS cmdfilename

Read Commands from File: With this command you can put a series of commands into a file and read them in just as if you had interactively typed them at SimDebug. This is useful in conjunction with command logging (see **LOG**) when you want to store and then replay a sequence of commands that got you to a certain place.

Normally SimDebug does not echo commands read from a file, even though output from these commands (e.g. **LR**) is, of course, visible. When you want to see the commands read from a command file you can **SET OREADCMDS 1**.

Init Command File: At the beginning of the SimDebug dialog, SimDebug looks for a file **simdebug.ini** in the current directory. When this file exists, it is read as a SimDebug command file before you enter the SimDebug dialog. In this file you can store your preferred SimDebug parameter settings (see **SET** command).

Empty lines in a command file are ignored. Commands from a command file are not remembered in the "last command" buffer. However, since 'empty commands' that re-execute the last command are still written to the command log file in full, you will still get exactly the same behavior when reading a command file previously written as a command log.

S [n]

Step: Single step. It executes the next **n** (default: **1**) SIMSCRIPT source lines and

then returns to the SimDebug dialog. **S** steps in to routines when the next instruction is a routine call. That is, it stops on the first instruction in the called routine.

Specifying Repeat Index n: After a single step command, SimDebug will show you the next executable source line. This is the source line that will be executed by the next **S** command. When you specify a repeat index **n** you generally do not want to see the output of the **n** source lines executed. However, if you do, you can enable the output for repeatable commands (**S,N, UP, DN**) with **SET OREPCMDS 1**.

Context Switch: When a context switch occurs during a **N** or **S** or **SR** command, a message is printed accordingly and the current simulation time is printed.

SET [[parname] [newvalue]]

Set SimDebug Parameter: Several aspects of SimDebug commands are controlled by parameters that you can change. **SET** without arguments lists the values of all SimDebug parameters. When a paremeter name (`parname`) is given, you can change its value. For example, **SET OREPCMDS 1**. *You only have to type the first few letters of a* `SET` *parameter that make it unique.*

SimDebug "SET parameters" and their meanings (**n**: integer > 0; **m**: 0 or 1; defaults are given in []):

SET WW n [5] (WhereWidth) Show ± **n** lines with **W** command. [5]

SET OREALF de a b
(OutRealFormat): Output format for Reals/ Doubles. They are output as **de**(**a**,**b**), e.g. **"E(14,4)"** [**D 17 6**]

SET OREPCMDS m
Show output from repeated commands (**n=1**) or not. [0]

SET OREADCMDS m
Show output from read commands (**n=1**) or not. [0]

SET EXTINFO m
Show extended information for pointer in mode field. [1]

SET GLOBWTRACE m
Show global variables (**GLOB**) with trace command (**T**). [0]

SET SHOWINTGL m
`m=1`: Show internal global variables with **GL**. [0] Internal global variables (**A.*, I.*, G.***) are created by SIMSCRIPT and are, in general, not useful to see.

SET SCROLLINES n

n>0: Output pauses after **n** lines. Press **Return** to continue. [0]

SET SHOWARRAYPTRS m

m=1: Show array mode information not as '**2-dim integer array**' but as the internal pointers that implement this array. [0]

SET SHOWSTACKLEVELS m

m=1: Show **SL=..**, (the stack level in traceback). [0]

SET SHOWLIBRTNS m

m=1: Show library routines in traceback [0].

SET NAMECOMPLETION m

m=1: Variable and routine names are automatically completed by SimDebug. That is, **FP CU** will follow the pointer that *begins* with **CU**. In case of multiple matches, you are offered a choice.

Note on **OREPCMDS** and **OREADCMDS**: Even when output from read or repeated commands is turned off, the output from the last command that was read or repeated will be shown so that you can see 'where you landed'.

SEV entname attrname value

Set Entity Values: Allows you to change the attribute value of a temporary entity. For quoting rules to set text values see the **SV** command. For **entname** you can enter the same values as for **FP**: an entity pointer name, an entity pointer value (in hex) or a **#n** (pointer number).

Using #n for entname: When you get to an entity using **FP** (follow pointer) commands, the display of the pointer attributes *in* the entity will 'overwrite' the pointer number **n** you used to display this entity (with **FP #n**). Thus, there is no longer a valid **#n** to use for **entname**. You should 'go back outside' of the entity (e.g. back one element in a list) and then use **FPN #n** to display the entity. **FPN** works like **FP** except that it does not reset the pointer numbers. This way you will keep all pointers along the way for use by **SEV**.

Limitations: It is currently impossible to change values of permanent entities (i.e. arrays). Also, you cannot set the values of packed temporary entity attributes.

SB rtnname lineno

Set Breakpoint: Sets a breakpoint in routine **rtnname** at line **lineno**. You can use "." for the routine name to denote the current routine (routine in the current frame).

SNAP

Snap: Calls your specified 'snap routine' **SNAP.R**. This is useful for debugging

complicated data structures that require special (user) code to display relevant information. You can use normal write statements to output your data.

Note that the output from this 'snap routine' will NOT appear in the log file (see **LOG**) but in the normal program's output. Thus, when output is redirected, the 'snap routine' will write into your output file.

SRCDIRS [src_dir_list]

Allow you to specify alternate directories where SimDebug can find the SIMSCRIPT source files (for **LS, W** etc.). **src_dir_list** is a list of directories separated by spaces. When no **src_dir_list** is given, the current source directory list is shown.

In searching for source files, SimDebug always starts at the current directory. If the source file is not found there, SimDebug looks into the directories in the order they were given in the **src_dir_list**. When your executable runs in a directory other than where it was built, it is advisable to specify the source directories as *absolute paths*.

Example:

SRCDIRS /src/d1 /src/d2 /src/d3

STOPTIME [stoptime]

Stop at Simulation Time: Allows you to stop execution (and call SimDebug) when the simulation time reaches the given stoptime. A stoptime of 0.0 means that there 'is no stoptime active'. The stoptime is only valid for 'one stop'. It is then reset to zero (set inactive again).

SV varname value

Set Value: Allows you to the change values in your program! Use **SV** to change values of simple variables of any type. You can change local variables, arguments and global variables.

For **text values:** Enter the text enclosed in string (double) quotes. When the string you want to enter should contain a string quote itself, it must be *doubled*, i.e. a single string quote is denoted by """". Use **SEV** to set attributes of entities.

SYSVARS

System Variables: Shows the values of several system variables such as **read.v, write.v, buffer.v, prompt.v**, and **hours.v**.

T [from [to]]

Traceback: Prints a traceback of the current call stack (the hierarchy of called routines) starting at the last called routine down to `MAIN`. The arguments **from** and **to** can be given to limit the traceback to a range of routines (useful for deep recursions). **From** and **to** are specified as the level numbers given in the traceback for each routine (`MAIN` is at level 1), where "**.**" as a level number means the 'current frame'.

By default, the level numbers (**[SL=...]** in the routine header in traceback) are not

given in the traceback. However, they are useful for deep tracebacks (when you want to see only part of the traceback) and for recursion. You can enable the display of these stack levels with **SET SHOWSTACKLEVELS 1**. See **SET** command.

Global variables: Generally the global variables are not considered a part of the traceback and hence are not shown with the **T** command. If you **SET GLOBWTRACE 1** (see **SET** command) you will also get the global variables at the end of each traceback (implicit **GLOB** command).

Output: For each routine, SimDebug first prints a line with the routine name, the file name, possibly the stack level and the current line number. When a routine is compiled with debug, all its local variables are shown with its values and modes. When a routine is not compiled with debug, only the routine name is shown. The variables are given in a sequence of sections: Given Arguments (ordered as in routine definition), Yielding Arguments (ordered as in routine definition), Local Variables (ordered *alphabetically*) and Local Saved Variables (also ordered *alphabetically*).

Several SimDebug controls the extent of the output for each variable parameter. See the **SET** command. The format of the output for each variable is described by the **PV** command.

The 'current frame' and 'current routine': The **T** command shows you the whole traceback, i.e. all routines in the call stack. Each invocation of a routine that is on the stack is called a **(stack) frame.** Initially, after a **T** command, the **top** routine on the stack (farthest away from `MAIN`) is called the **current routine**, which is in the **current frame.** Since a routine can be called recursively we must destinguish between 'routine' (the source code) and the 'frame' (invocation of the routine [its arguments and local variables]). When **PV** looks up a variable, it starts at the current frame and when the variable is not found there, it looks at global variables. The commands **up. dn. top. bot** move the 'current frame' up, down, to the top (last routine called), or bottom (`MAIN`).

TC

Traceback Current: Write trace of current frame.

TOP

Top Frame: Set 'current frame' to the top of the stack, which is the last user routine, called (farthest away from `MAIN`). See note on 'current frame' in the **DN** command.

UP [n]

Up Frame: Set 'current frame' `n` levels up (away from `MAIN`) in the stack. Default: **n = 1**. **[SL=...]** in the header line shows the stack level. See note on 'current frame' in the **DN** command.

W [n]

Where: Shows where you are in the source in the current frame. It shows **n** source lines around the current line. The default **n** is taken from the SimDebug parameter **WW** (see **SET** command). The 'current source line' is shown with a ">" in front of it.

Breakpoints appear with "**#**" in front of the line.

WT [filename [from [to]]]

Write traceback (output of **T**) and the output from the **IO**, **MEM**, and **EV** commands to a file. The default filename is **trace.out**. By specifying **from** and **to** you can limit the traceback to those levels. When the trace file exists it is overwritten.

WTA [filename [from [to]]]

Write Trace Append: Same as **WT** except that the output is appended to the trace file.

4.4 Advanced Topics

4.4.1 Batchtrace.v

Normally, when a SIMSCRIPT program runs into a runtime error, SimDebug will be called so you can examine the stack and variables to find out what went wrong. Sometimes you may want to just get a traceback into a file and want the program to terminate on a runtime error, e.g. when you run i t in batch mode. When you set the system variable **batchtrace.v = 1**, a runtime error will cause the traceback. The I/O, event and memory information will be written to a fixed file **simerr.trc**.

Another way of setting **batchtrace.v** to **1** is to call your executable with the command line option **-batchtrace**. As with **-debug** your application program does not see this option.
Setting **batchtrace.v = 2** causes an immediate exit in case of a runtime error or a user interrupt (e.g. **Ctrl-C**). No traceback is written.

4.4.2 Signal Handling / External Events

SimDebug uses the signal handling facilities of the operating system to catch events like floating point errors, segment violations etc. If your program uses C code that sets its own signal() handling routines, you must comment out that code as long as you want to use SimDebug on that program. Any mix will not work.

4.4.3 Reserved Names

In SIMSCRIPT all names that begin with "**<letter>.**" or end with "**.<letter>**", where "**<letter>**" is any letter, are reserved for the system's usage. This is why they do not appear in SimDebug.

If you use such an illegal name, e.g., for a routine, it will not appear as a user routine in

SimDebug. You cannot see it with the **LR** command. Thus, even if such a routine name does not clash with a system routine, you should not use these kinds of names.

4.4.4 Displaying Arrays

Before discussing SimDebug's array display capabilities we must discuss some background information. Each SIMSCRIPT object that a pointer can point to, such as arrays, text or dynamic entities, has a descriptor that contains information on what this 'object' is and how to interpret the data. For instance, an entity descriptor contains the entity ID and, an array contains the size of the array and the type of its elements. This means that the **FP** (follow pointer) command can always follow a pointer to anything and display what it finds.

Apart from that, SIMSCRIPT supports array equivalencing. You can define an array **IA(*)** for instance as a 1-dim integer array, and then assign the pointer **IA(*)** to a variable of type 1-dim **alpha** array **AA(*)** and look at the data as characters.

The command **PAV** (Print Array Variable) looks at the array '*through the eyes of the array variable*', i.e. in the above example **AA(*)** as **alpha**.

The command **PDV** (Print from Descriptor Variable) always looks at the array with the data given in the descriptor. It looks at how the array was first created, and, in the example above, looks at the array as **integer**.

4.4.5 Permanent Entities and System Owned Variables/Sets

Permanent entities are implemented as a set of 1-dimensional arrays that will appear as global arrays. Use the **GLOB** command. At this point the different fields of a permanent entity are not shown together (e.g. with the entity name), but appear separately in the **alpha**betical listing of all global variables.

'**The system owns**' ... variables and sets show up as global variables, in **alpha**betical order.

4.4.6 Conditional Breakpoints

Certain problems only appear after a large amount of data has been processed. For example, after 10000 iterations in a loop. To allow you to break the process and go into the debugger upon any arbitrarily complex condition, SimDebug offers you a direct call to **SIMDEBUG.R**.

When you call this routine from your application program you are put into the SimDebug dialog just as if you had set a breakpoint. You can examine the stack, global variables, entities, and single step through the program in the usual manner.

Example:

```
for i = 1 to 10000
```

```
do
  .... do something ....
  if i>10000 and A+B-C > DATTR(ENTPTR)
    call SIMDEBUG.R
  endif
loop
```

4.4.7 Continuous Variables

Continuous variables (for continuous simulation) are implemented as *right and left functions.* Therefore, they will show as right and left routines in the **LR** command, but not as variables.

WARNING

Simdebug Recursion: SimDebug protects itself from errors that normally cause a program to fail, such as attempting to use a bad pointer, or having unaligned accesses. However, in some rare cases it can happen that SimDebug does not catch an error condition that then causes another error 'within' SimDebug. Since SimDebug is a program that is called when an error occurs, *SimDebug will be called from within SimDebug*! You will get a warning message.

You can look at some more variables, but you cannot continue the execution. Exit from SimDebug with **QUIT** and restart your program to find the error.

Appendix A Compiler Warning and Error Messages

A.1 Warning and Error Messages

During compilation, warning messages and error messages may be produced. The text of each message appears below:

1001 Invalid syntax
A word found in the input stream did not conform to the syntax requirements of the SIMSCRIPT II.5 language. The unrecognized word is ignored and the error scan resumes with the next statement keyword in the input stream.

1002 Missing ')'
An arithmetic expression or subscript is missing a right parenthesis. A (possibly misplaced) right parenthesis is assumed.

1003 Missing terminal " in ALPHA literal
An **ALPHA**numeric string must be contained on one line.

1004 More format specifications than variables
In formatted **read** and **write** statements, there must be a one-to-one correspondence between variables and format descriptors. The format descriptors, including "/," must be separated by commas. In a **print** statement, fields are defined by "*" or a sequence of at least 8 contiguous periods.

1005 More variables than format specifications
See message 1004.

1006 Conflicting or redundant properties in define
More than one **MODE, DIMENSION** or **TYPE** specification appears in the same **define** statement. The indicated statement is ignored.

1007 Number of subscripts different from definition or previous use
A subscripted variable is redefined with a different number of subscripts than originally, or a set name in a **file** or **remove** statement is improperly subscripted.

1008 else or always without matching if
The indicated statement is misplaced in the program.

1009 if not terminated by always
This error is detected at the end of a routine.

1010 Use conflicts with definition
The previous definition or use of this name precludes its use in this context. This message can apply in a number of cases. The most common are described below.

- A **belong** clause in an **every** statement does not refer to a set name.
- Common membership in sets is limited to temporary entities.
- An **every** statement attempts to define an entity but the name has already been defined differently.
- A **define** statement attempts to define a variable, a procedure or a set, but the name has already been defined differently.
- The variable in an **external unit** statement has already been defined differently.
- The attribute of a **has** clause has already been defined differently or a common attribute is defined with a different **word** assignment or packing code.
- Attempt to **read** or **write** a variable defined as a set.
- Attempt to **release** a quantity, which is not an array, a routine or a subprogram variable.
- Attempt to store in a **random** variable.

1011 Illegal assignment target
This error is caused by an illegal attempt to store information in a built-in function. Builtin functions include **abs.f, div.f, int.f, real.f, mod.f, max.f, min.f** and all **text**-related functions. Except for **substr.f**, these functions cannot be used on the left-hand side of assignment statements or as **yielded** arguments.

1012 Array number out of range
Application has more than 8000 variables and/or permanent entities. The maximum permissible array or word number for global variables or permanent attributes is 8000. Use of an array number larger than this is not permitted in this implementation.

1013 Context requires routine name
A **routine** statement uses an incorrect name or the name appearing is not a routine name.

1014 return with not allowed here
Event routines and left-handed routines cannot return any values.

1015 loop without a matching do
The **compiler** ignores the loop statement.

1016 Implied subscripting attempted on a common attribute
Common attributes must be explicitly subscripted.

1017 Number of given arguments inconsistent with definition
A **call** or function reference uses a number of arguments different than that defined for the subject routine.

1018 Multiple definition of label
The label has been defined elsewhere in the routine.

1019 Subscript required on label
The label name was previously encountered with a subscript.

1020 Name repeated in parameter list
The names in the **given** arguments list or in the **yielded** arguments list may each appear only once in the list.

1021 Undefined label
This error is detected at the end of a routine.

1022 do without a matching loop
This error is detected at the end of a routine.

1023 MAIN routine should use stop
The `MAIN` routine should not use a **return** statement. The compiler substitutes a **stop** statement.

1024 Missing end
The compiler supplies the `end` statement and completes the processing for the routine.

1025 define to mean or substitute incomplete
An end-of-file was encountered during the processing of a **substitute** statement or no substitutable text was found. Blanks and comments (") are invalid substitutable text. The statement is ignored.

1026 Inappropriate mode or dimension for implicit subscript
Due to local redefinition, the mode or dimensionality for this implied subscript is inappropriate. The compiler ignores the dimensionality but uses the new mode.

1027 Attribute in first 5 words of event notice is illegal
The first five words of an event notice contain the **time.a, m.ev.s, p.ev.s, s.ev.s** and **eunit.a** attributes. These attributes cannot be redefined. The compiler ignores the specification.

1028 Context requires an un-subscripted subprogram variable
An indirect call to a function using the **$** name feature requires that the **subprogram** variable name be unsubscripted, as the subscripts are treated as given arguments for the indirect call.

1029 Attribute in first 8 words of process notice is illegal
See message 1027. In addition, a **process notice** contains the **ipc.a, rsa.a, sta.a** and **f.rs.s** attributes.

1030 Temporary attribute word number out of range
The maximum permissible entity length is 1023 words. Entities of this size should never be required.

1031 Subscripts not permitted for this variable
A variable defined as unsubscripted is used with a subscript.

1032 Non-integer subscript on a temporary attribute
Temporary attribute subscripts must be pointers.

1033 Negative constant used as a subscript
This illegal condition cannot be compiled.

1034 Subscript not permitted on label
A label is used with a subscript in a **go to** statement or is defined as subscripted although it has already appeared without a subscript.

1035 then if statement appears outside if
The **then** keyword can only be used within an **if** block. The compiler ignores the **then** word.

1036 Missing ')' in logical expression
A (possibly misplaced) right parenthesis is assumed.

1037 div.f valid only with integer values
A floating-point division is performed.

1038 Number of yielding arguments inconsistent with definition
See message 1017.

1039 Attribute of mixed compound entity must be a function
Attributes of mixed compound entities (compound of at least one permanent entity and at least one temporary entity) must be functions. The compiler assumes a function definition.

1040 Attempt to equivalence function attributes
Function attributes are not assigned any storage and therefore cannot be equivalenced.

1041 Missing ')' in equivalence attribute group
A (possibly misplaced) right parenthesis is assumed.

1042 Attempt to pack function attribute

Function attributes are not assigned any storage and therefore cannot be packed.

1043 Attempt to pack un-subscripted system attribute
The packing definition cannot be honored.

1044 Illegal packing code
For bit packing, the bit numbers should satisfy the inequality **1** ≤ **n** ≤ **m** ≤ **32**. For field packing and intra-packing, the denominator must be **2** or **4**.

1045 Packing code (*/n) illegal for temporary attribute
The ***/N** packing codes can only be used for arrays (such as attributes of permanent entities or subscripted attributes of **the system**). A field packing of **1/N** is assumed.

1046 Compound entity may not belong to a set
The compiler ignores the **belong** clause.

1047 Attempt to define non-local variable as saved or recursive
This is an attempt to define a local variable in the `PREAMBLE`. The definition is not processed.

1048 Incorrect mode specified for packed variable
Packing applies only to **INTEGER** quantities.

1049 Defining set not previously declared in every statement
Set definitions must be placed after the **owns** and **belongs** clauses defining their owner and members. The definition of the set is ignored. This may cause follow-on errors.

1050 Statement should be preceded by a control phrase
A **compute** statement, **find** statement, **when** statement or a controlled **read** or **write** statement must be within a **for**, **while** or **until** block.

1051 write format used in read statement
A character string appears in the **as** clause of a **read** statement.

1052 Illegal or out of place '*'
Either an attribute of a temporary entity or an argument to a function call is subscripted by an *****, or an array reference has an ***** before the last subscript.

1053 Attempt to perform set operation on a non-set
A **file** statement, a **remove** statement, a **for each of set** statement, an **if set is empty** or a **before** or **after** statement references a quantity not defined as a set.

1054 Statement requires attributes not defined for named set
A **file** statement, a **remove** statement, an `if set is empty` or a **for each of set** phrase is used, but the necessary set attributes were deleted by a **without** phrase.

1055 Name of a permanent entity required in this context
A **create each** statement or a **for each** statement must refer to a permanent entity.

1056 also statement outside do ... loop
An **also** statement appeared outside of a **do** block. The compiler assumes a **do** statement after the **also** block.

1057 Name of a temporary entity required in this context
A **create** statement, **destroy** statement or **before** or **after** statement must refer to a temporary entity.

1058 group used without column repetition
An **in groups of** phrase must be controlled by a **for** phrase. The statement is ignored.

1059 Name of an event required in this context
The **event, process, activates, cause, cancel, break ties** and priority statements must refer to an event or process name. In the case of an **event** or **process** statement, a routine named **R0** is assumed.

1060 Misuse of suppression amid column repetition group
The suppression phrase is misplaced.

1061 Context requires a for phrase to follow the word printing
The **printing** phrase is not properly programmed.

1062 Column repetition context requires in groups of phrase
The column repetition clause must include an **in groups of** phrase.

1063 Column repetition group size is illegal
The **in groups of** phrase specifies a **0** group size. The compiler assumes a value of **1** in its subsequent error scan.

1064 end statement required to terminate report heading
The compiler assumes an **end** statement at this point.

1065 end statement required to terminate report
The compiler assumes an **end** statement at this point.

1066 print 0 lines statement is ignored
Subsequent error messages may refer to form lines.

1067 Too few formats or too many expressions in print
There must be a one-to-one correspondence between expressions and format specifications.

1068 Set owner or member not defined
A set name must appear in both an **owns** clause and a **belongs** clause to be defined. Both the **owns** and the **belongs** clauses must precede the set definition.

1069 Attributes of common set must be declared in an every statement
The set pointers must appear in an **every** statement. No attribute definition takes place.

1070 Mode of quantity conflicts with automatic definition
The **M** or **N** attribute for a set, or the **N.entity** name for a permanent entity were explicitly defined with **real** mode. They must be **integer**.

1071 Number of subscripts conflicts with automatic definition
The attributes of a set were explicitly defined with an incorrect dimension, or the **N.entity** name for a permanent entity was defined as a subscripted variable.

1072 Explicit definition conflicts with automatic definition
One of several conditions has appeared:

- The owner or member attributes of a set were explicitly defined and their definition conflicts with the **owns** or **belongs** clause for the set.
- The **N.entity** name for a permanent entity is neither a global variable nor apermanent attribute of **the system**.
- The **F.name** or **S.name** of a **random** variable should be left for automatic definition.

1073 Ranking attribute must be declared in an every statement
The ranking attribute in the **define set** statement is not an attribute of the member entity.

1074 Illegal file statement for ranked set
The **file first, file last, file before**, and **file after** statements are not permitted on ranked sets.

1075 Number of given arguments exceeds the maximum allowed
The combined number of **given** and **yielding** arguments cannot exceed **127**.

1076 Number of yielding arguments exceeds the maximum allowed
See message 1075.

1077 Number of subscripts exceeds the maximum allowed
The maximum number of subscripts allowed is **254**.

1078 Label subscript must be between 1 and 3000
The maximum subscript allowed on a label is **3000**. Since subscripted labels require a table as large as the maximum subscript value, smallest program size suggests that subscripts should normally range from **1** to **n** in increments of **1**.

1079 Number of recursive local variables exceeds available space.
Each routine has 1024 words of storage available for recursive local variables. Some of this total is used by variables which the compiler generates internally.

1080 Context requires subscripted label
A subscripted label is required at this point.

1081 Yielding arguments illegal in left-function
Yielding arguments are not allowed in monitoring routines or left-handed functions. Ignoring the yielding argument list scans the routine.

1082 enter statement permitted only in left-functions
This statement should be the first executable statement in a left-handed routine.

1083 Global properties specified in local define
Local variables cannot be monitored, packed, or defined as **stream** variables.

1084 Incorrect number of given arguments in left-function
A routine monitoring a variable must be given the same number of arguments as the number of subscripts originally defined for the variable.

1085 move statement not allowed here
A **move to** statement can only appear in a right-handed routine. A **move from** can only appear in a left-handed routine. The statement is out of place.

1086 before creating and after destroying options not allowed
After creating and **before destroying** can be used to collect usage statistics.

1087 More arguments than defined attributes in process or event
It is necessary to define an attribute to hold each argument received by the event. The excess arguments supplied can receive no values.

1088 More arguments than defined attributes in activate
It is necessary to define an attribute to hold each argument received by the event. The excess arguments supplied cannot be stored anywhere.

1089 Context requires name of an entity
A **list attributes of** statement does not refer to a temporary entity.

1090 Illegal attempt to break ties on an external event
External events cannot appear in **break ties** statements.

1091 Illegal attempt to equivalence random attributes
Random attributes cannot be equivalenced with other variables of any type.

1092 Illegal mode for a random variable
A **random** variable cannot be of **alpha** or **text** mode.

1093 stream phrase ignored - variable not defined as random
The **define name as stream** statement should be placed after the definition of the variable as a **random** variable.

1095 cycle or leave ignored - no loop in effect
Either **cycle** or **leave** must appear within a **do ... loop** block.

1096 Missing here for a jump back

A **here** statement must exist prior to the occurrence of a matching **jump back** statement.

1097 Missing here for a jump ahead
A **here** statement must appear after a **jump ahead**. This error is detected at the end of the routine.

1098 Both accumulate and tally illegal on the same variable
The mixing of statistics type is not allowed for a given variable. See message 1099.

1099 accumulate/tally illegal for monitored/random variables
These operations are in fact implemented by constructing monitoring routines.

1100 Statistic requested twice for the same variable
One statistical keyword appeared more than once for a given variable.

1101 Improper type of variable for accumulate or tally
Accumulate or **tally** can be requested for unsubscripted global variables, attributes of permanent entities, temporary entities, event notices, processes, resources and compound entities. They cannot be requested for subscripted global variables, subscripted attributes of **the system**, or common attributes of temporary entities.

1102 Attribute for accumulate or tally improperly pre-defined
The variables containing the accumulated or tallied statistics should be left for automatic definition by the compiler. They should not appear in **define** statements.

1103 Accumulate or tally on an undefined variable
The name of the variable is probably spelled wrong.

1104 Histogram of attribute of a temporary entity is forbidden
Histograms may be requested for global variables, system attributes, and attributes of permanent entities.

1105 Improper word boundary for a variable of mode double
Certain systems — the Gould and IBM mainframes among them — require that all double precision floating point numbers be aligned on a double-word boundary. This requires that un-subscripted **double** permanent attributes be assigned to odd-numbered **in array** numbers, and that **double** temporary attributes be assigned to odd **in word** numbers. Other systems — such as the VAX — do not require such assignments, but are compatible with them.

1106 Multiple else statements not allowed on a if
The language allows only one **else** statement. Other diagnostic messages may indicate the prior **if** statement was not processed.

1107 Then if statement after else - obscure structure
The **then if** construction is not permitted on a structured **if**. Correct by explicitly using **else** and **always** statements as appropriate instead of using **then if**.

1108 Else statement after then if - obscure structure
See message 1107.

1109 A statement above this point is unreachable
An unlabeled statement or group of statements follows a **return** or an unconditional transfer. This may be due to a missing label, **else,** or **case** statement.

1110 Process not declared - routine assumed
The **process** routine has not been declared in the PREAMBLE.

1111 This statement may appear only in a process

1115 Illegal implied conversion between text and other modes
Use **ttoa.f** or **atot.f** or access conversion routines by **write** and **read using the buffer**.

1116 Improper argument mode for intrinsic function
An argument of mode **text** was expected and not found, or a **text** argument was given where a numeric argument was expected.

1119 Packed variable cannot be passed in this context
Array rows of variables that are bit packed, or packed (n/m), cannot be passed as arguments to NONSIMSCRIPT routines. Individual elements or arrays packed (***/m**) are valid arguments.

1120 Improper first argument to left substr.f
The first argument to **substr.f** must be an unmonitored text variable.

1121 Attempt to equivalence text variable
Text variables cannot be equivalenced with other variables.

1124 Conflicting parameters in open or close
The **open** or **close** statement was used improperly.

1126 open does not specify either input or output
Either **input** or **output** (or both) must be specified as an **open** statement option.

1127 text function illegal in store statement
The **store** statement should generally not be used with **text** data. In this instance, its use would result in permanent loss of a block of memory.

1128 double variable overlap caused by equivalencing
A double variable occupies two successive array number locations. The second of these should not be assigned to any other use.

1129 always is preferred usage in this context
The **else (otherwise)** statement should be changed to an **always**.

1130 Number of labels exceeds allowed maximum
Implementation constraints impose a limit on the allowed number of statement labels. The routine should be subdivided into two or more routines.

1131 Subprogram variable used out of context
A **subprogram** variable may not be used within a computation.

1132 Implicit conversion of subprogram variable
Only **subprogram** variables or **subprogram** literal values may be assigned to a variable declared as mode **subprogram**.

1133 Dimensioning of attributes not permitted
Attributes of temporary and permanent entities are implicitly 1-dimensional, subscripted by an entity pointer value. The explicit dimensioning of these may cause ambiguity. A dimension of 1 is substituted.

1134 Illegal use of store with quantities of differing mode
This usage of **store** may have undesirable side effects and is no longer permitted.

1135 Use of store with text quantities may have undesired effect
The use of the **store** statement between **text** quantities is allowed, but strongly discouraged, because it disables the automatic actions that assure the integrity of **text** values.

1136 Variable is undefined or not fully defined
This message appears when the background mode has been explicitly set to **undefined** using a **normally** statement.

1137 Parameter in open statement not supported
Differences in operating systems do not allow complete compatibility between SIMSCRIPT III implementations of the `open` statement. Unsupported parameters are ignored.

1138 Release routine statement no longer supported
The statement is ignored.

1139 Reset references variable not accumulated or tallyed
Totals do not exist for a variable which has not been the object of an **accumulate** or **tally** statement.

1140 Reset uses qualifier not declared as such
Only a qualifier defined for an **accumulated** or **tally**ed statistic may be specified in a **reset** statement.

1141 This statement not supported or no longer required

1142 Local variable used only once
The indicated local variable appears only once in the routine. This could be due to a typographical error.

1143 Local variable never modified
The indicated local variable has not been modified by the routine. This means that its value is always zero (or `""`, if a **text** variable). This could be due to a typographical error.

1144 Bad Block structure - overlapping do and if
The statement violates SIMSCRIPT III's structured programming nesting rules, by overlapping one of the following three control structures:

- **do ... loop**
- **if ... else ... endif**
- **select ... case ... default ... endselect**

For example, if the statement in error is a **loop** statement, then an **if** block was not terminated by an **endif**, or a **select** was not terminated by an **endselect**. The error will also be seen when one block overlaps a portion of another block, as in **if ... do ... else ... loop ... endif**.

1145 Variable or function name required
A non-numeric quantity — such as a set — cannot be the object of a **read, print**, or **list** statement. A statement such as **list attributes of each entity in set** may have been intended.

1146 Assignment between incompatible data types
Check the modes on both sides of the equal sign in an assignment (**let**) statement.

1147 Implicit conversion of pointer variable
The indicated variable must be either mode **pointer** or mode **integer**.

1148 Name of a resource required
The **request** and **relinquish** statements apply to resources only.

1150 Multiple MAIN routines encountered
Only one `MAIN` routine may be included in any compilation.

1151 case control outside select...endselect
A **case** or **default** statement can be used only between a corresponding **select ... endselect** pair.

1152 Mode of case term does not match select
The mode of the term is incompatible with the mode of the **select** expression. Some mode conversion is performed. A **real** expression may include integer terms, and both **text** and **alpha** expressions require string literal **case** terms. If necessary, assign the expression to a variable of the appropriate mode.

1153 case term duplicates previous term(s)
This term is unreachable because it is completely blocked by corresponding terms in an earlier **case** statement. This message will not be given for **select** expressions with a mode of **real, double**, or **text**.

1154 Statement not allowed after default
The **case** or **default** statement is not valid within a **select** block after the use of the **default** statement.

1155 No case statements appear within select

Each **select ... endselect** block must include at least one **case** statement.

1156 Select case without matching endselect
Each **select case** block must be terminated by a matching **endselect** statement.

1158 Symbol redefinition
A local **define to mean** is redefining a global **define to mean**, without an intervening **suppress substitution**. This may have unexpected consequences. For example, if the PREAMBLE contains the statement **define .NUMBER to mean 10**, and a routine contains the statement **define .NUMBER to mean 20**, the compiler will first substitute **10** for **.NUMBER** in the routine, making the statement read **define 10 to mean 20**, and will then substitute **10** for **20** throughout the remainder of the routine.

1161 Changing PROCESS pointer may affect implicit subscripting
Changing the pointer to a **PROCESS** within its **PROCESS** routine will prevent the routine from later accessing the attributes of the current process. Such attributes are often referenced through implied subscripts. This warning may be the result of an **activate**, **create** or **remove** statement intended to point to a different process notice. Use a different pointer name to avoid this problem.

1162 Storage may not be de-allocated on destroy of a process
When a **PROCESS** terminates normally, SIMSCRIPT II.5 automatically performs some memory management functions. By explicitly **destroy**ing the **PROCESS** pointer, these functions are disabled. In general, if a **PROCESS** may be terminated prematurely, the **PROCESS** itself should check for the conditions requiring termination, rather than having the **PROCESS** pointer destroyed by a separate routine.

1163 Context requires the name of a HISTOGRAM
A statement of the form **accumulate HISTOGRAM.NAME (LO to HI by INCREMENT) as the histogram** of **VARIABLE.NAME** must appear in the PREAMBLE. Also see message1104.

1164 Name of routine is not a monitored variable
SIMSCRIPT II.5 monitors global variables by defining routines with the same name. In this case, you have provided a routine with the same name as a global variable, but the variable is not being monitored. Rename the variable or the routine.

1165 Statement out of place
A **PREAMBLE** type statement appeared in a routine, or vice versa. The unrecognized word is ignored and the error scan resumes with the next statement keyword in the input stream.

1166 Invalid literal value
The value of the literal provided is too large to hold in a variable location.

1167 Returned Function mode undefined
The mode of the value returned by a function must be declared in the PREAMBLE (**define FN as a FN.MODE function**). If the mode is not explicitly included in the **define** statement, the background (i.e., **normally mode is...**) mode currently in effect is assumed.

1168 Function should return a value

1169 Statement incomplete

1170 Pointers can test for equality only

1171 Used as implicit subscript
SIMSCRIPT II.5 is free format and allows for usage of implicit subscripts. This increases the expressive power of the language but sometimes is error prone. You can suppress implicit subscripts by using the SIMSCRIPT II.5 language statement:

```
suppress implicit subscripts
```

The compiler will generate warning message 1171 whenever it detects implicit subscripts usage. The scope of the **suppress** statement is global if used in a PREAMBLE or local if used in a routine. Usage of implicit subscripts can be resumed by the statement:

```
resume implicit subscripts
```

Any number of `suppress/resume` statements are allowed in a routine.

1172 Subscript should be pointer mode

1173 Attribute doesn't belong to subscript
If a reference pointer is used as a subscript, the compiler can detect when the attribute is unrelated to the reference type.

1174 Mode of given variable does not belong to set
An attempt was made to FILE or REMOVE from a set given a reference pointer that does not belong to the set.

1175 Mode not specified
A mode was not specified in a list of argument prototypes.

1176 Mismatched number of dimensions in argument
When passing an array argument to prototyped routine or method, the number of dimensions defined in the prototype must exactly match the number of dimensions of the argument.

1177 Mode of argument does not match prototype
When passing an argument to prototyped routine or method the mode of the argument must be "assignment compatible" with the mode defined in the prototype.

1178 Name is ambiguous and needs proper qualification
This message will be reported when an unqualified name is used that matches different classes or entities declared in two or more modules. The error also occurs if the name matches more than one inherited class or object attribute.

1179 A class cannot be cyclically derived from itself
Illegal cycles are detected and reported by the compiler when the "IS A" clause is used to derive a new class from an existing class that is already derived from this new class.

1180 END required to terminate a class declaration
Every BEGIN CLASS statement must be terminated by an END statement.

1181 Subscript must be a reference variable
All pointer variables used as subscripts to an object attribute or method must be REFERENCE pointer of the matching type.

1182 Must be declared as an object or class attribute
All object attributes and methods referred to in a DEFINE statement must be declared in the same BEGIN CLASS block.

1183 Class attributes must be defined inside a class definition block
The "THE CLASS HAS" statement can only appear inside a BEGIN CLASS...END block.

1184 Invalid name
This message is usually reported when a qualification is used in a place that it is not expected, or if the qualification is of the improper form.

1185 Word assignment of object attributes not allowed
The IN WORD clause cannot be used to explicitly assign storage locations for object attributes.

1186 Data size mismatch for equivalent object attributes
Equivalent object attributes must have the modes that are the same size in bytes. For example, a DOUBLE variable cannot be equivalanced to an INTEGER.

1187 The name of a class is required in this context
The compiler expected the name of a class declared in either the current module or an imported module.

1188 Method must be overridden
When using multiple inheritance, two or more parent classes may derive from the same class. If any parent class has overridden a method defined in this "common" base class, then the name of this method has now become ambiguous from the standpoint of the multiply derived class. This method must be overridden.

1189 Unknown module
The name of an unknown module was used in a qualification.

1190 Qualifier is not a class
A class qualifier must the name of a known class.

1191 Name not defined or inherited by class
The name of an object attribute, object method, or class set was expected. The name must be defined or inherited by the class reference in the nearby subscript, or the current class.

1192 Invalid use of set
The name of a set was used in a context where it was unexpected. Or a set was expected in a

certain context and not found.

1193 Invalid reference
The name of a class or temporary entity was expected as a reference type.

1194 Undefined name
The name is not defined in the current module or any imported module.

1195 Set must be declared by a belongs phrase of this module
The name given for a set is not defined in the current module or any imported module. This error can occur if necessary set name qualification is omitted.

1196 Set must be declared by a belongs phrase of this class
An attempt was made to define a something as a set that has not been declared as a set through the BELONGS phrase.

1197 Name cannot be used to rank entities or break ties
Ranking attributes must be attributes of the same entity or object that the set belongs to.

1198 Process method expected
A name was used in a context where a process method was expected

1199 A function cannot have yielded arguments
Only subroutines and subroutine methods can have yielded arguments.

1200 Constant used as a subscript
It is not allowed to use a constant in any context where a pointer is required.

1201 Invalid method
If inside a BEGIN CLASS block, the name used after the word CALL in a before/after statement must be the name of a method.

1202 Process method not declared in enclosing class
A process method name is expected for before/after scheduling/canceling statement appearing inside a BEGIN CLASS block.

1203 Method not declared in enclosing class
An attempt was made to define an undeclared name as a method.

1204 Unknown event
An event heading was found that was not declared in the preamble under the EVENTS section or was not used in a EVENTS INCLUDE statement.

1205 Unknown process
An event heading was found that was not declared in the preamble under the PROCESSES section or was not used in a PROCESSES INCLUDE statement.

1206 Unknown class
The class name used in a qualification or in a METHODS FOR statement was not declared in

a BEGIN CLASS block.

1207 Incompatible reference
In a CREATE/DESTROY statement, the mode of reference variable appearing after CALLED must match the class or entity name appearing before CALLED.

1208 Entity type or class not permitted in this statement
The name of a process, event or process method was expected.

1209 Source code file could not be opened
The error message will occur when attempting to import a module that cannot be found. The source code file containing the public preamble of the imported module should be named after the module with ".sim" appended.

1210 Only one main module is allowed
A second main module was found on the command line.

1211 Redefinition of a system entity
In SIMSCRIPT III, predefined system entities cannot be modified through the every statement.

1212 Word number mismatch for equivalent attributes
If the IN WORD clause is used for two equivalent attributes the word numbers must match

1213 Name already specified in a priority order statement
The same event, process, or process method cannot appear in more than one PRIORITY ORDER statement.

1214 Comparison of pointer to a constant
Only integer variables should be compared to a constant.

1215 Private preamble is unexpected here
A private preamble must proceed its public preamble. This error can also occur if a duplicate private preamble is found.

1216 Public preamble is unexpected here
Only one public preamble is allowed per file.

1217 LIFO and FIFO are ignored for ranked sets
The LIFO and FIFO keywords do not apply if the set is ranked.

1218 Name of private preamble must match public preamble
If a private preamble is included in the same source as the public, its name must match that of the public preamble.

1219 Public preamble was not found
The preamble heading for an imported module was not found.

1220 Public preamble name must match source file name

The file containing a public preamble must be named after the preamble. For example, the public preamble for a module named FACTORY must be found in the file named "factory.sim"

1221 Left monitoring routine/method not found
The compiler will generate a warning whenever routines and methods are not found in the any of the source code files presented by the user. Use the "-v" option to prevent these warnings from being displayed.

1222 Right monitoring routine/method not found
The compiler will generate a warning whenever routines and methods are not found in any of the source code files presented by the user. Use the "-v" option to prevent these warnings from being displayed.

1223 Override on the right not allowed here
If an object attribute is overridden, only a LEFT METHOD can be provided for this attribute.

1224 Override on the left of non-numeric not allowed here
Only object attributes of mode alpha, integer2, integer, real and double can be overridden.

1225 Initialize not allowed in main module
The INITIALIZE routine is only allowed in subsystems.

1226 Found duplicate routine or method
A second routine or method heading was found.

1227 Implementation of method not found
The compiler will generate a warning whenever routines and methods are not found in any of source code files presented by the user. Use the "-v" option to prevent these warnings from being displayed.

1228 Mode of yielded argument does not match prototype
If the dimensionality of a prototyped yielded argument is greater than 1, the mode must match exactly. For 0-dim arguments, the mode of argument must be assignment compatible with the mode of the prototype.

1229 Process methods cannot return a value
A process method cannot be defined as a function. A RETURN WITH statement is not allowed in the body of a process method.

1230 Pointer variable used in expression
A warning will be displayed if a variable of mode POINTER is used in an arithmetic expression.

1231 Invalid boundaries for histogram
The "high" value used in a histogram declaration must be greater than the low value. The DELTA value must be greater than zero.

1232 Definition cannot be modified privately

In some cases it is not permissible to modify the definition of a name in the private preamble that was originally declared in the public preamble.

1233 Argument list ignored when scheduling existing processes
If a "THE" keyword is used in conjunction with the SCHEDULE statement, arguments are not passed to the process or process method.

1234 Implicit conversion of a reference variable
A reference variable cannot be used in the same way as a generic pointer variable, and cannot be used in any expression. It must only point to an object or entity of the type.

1235 Unsigned INTEGER4 bit 32 ignored under this architecture
Currently, the unsigned INTEGER4 mode is not supported on 32 bit platforms. The "SIGNED" term can be used when defining a variable as INTEGER4 to get rid of this warning.

1236 MAIN cannot appear inside a subsystem
The MAIN routine can only appear in the Main module.

1237 Invalid context for definition of a dummy variable
A dummy variable cannot be defined here

1238 Invalid prototype for routine or method
Prototypes and headings for routines and methods specified in a BEFORE/AFTER statement are subject to restrictions since these routines are called automatically. See the SIMSCRIPT III reference manual for a description of the arguments for these routines.

1239 Invalid name for override of object method or attribute
The name used in an override clause must be inherited by the object.

1240 Set must be declared by an owns phrase of this class
The set referenced in a BEFORE/AFTER FILING/REMOVING statement must be owned by the class containing the statement.

1241 Set must be declared by an owns phrase of this module
The set referenced in a BEFORE/AFTER FILING/REMOVING statement must be owned by an entity or the system within the module containing the statement.

1242 Missing comment delimiter
Multiple line comment delimiters /~ and ~/ must be used in pairs.

1243 Invalid assignment to a constant
Constants cannot be used on the left side of an assignment statement.

1244 Global, class or local constant expected
Some statements (such as the CASE statement) expect only numeric or predefined constants.

1245 Use of a deprecated feature
Some features of SIMSCRIPT II.5 have been deprecated in SIMSCRIPT III.

These features can still be used in practice but are no longer formally documented. (Passing the "-w1245" option to simc will eliminate these warnings.)

1246 Duplicate definition or specification
A name was used twice in the same list.

1247 Use of a monitored variable in substring assignment
Using the SUBSTR.F call on the left while passing a left monitored text variable as an argument is not supported.

1248 Invalid mode for statement
A variable used within the statement was not of the expected mode. (Pointer/integer mode interchangeability is not allowed in some cases in SIMSCRIPT III).

1249 Attempt to create a process method
Unlike process routines, the notice for process methods cannot be instantiated using a CREATE statement. The process notice must be created using a ACTIVATE/SCHEDULE/CAUSE statement.

1250 Assignment to a statistical variable
Statistical holder variables such as those declared in a tally or accumulate statement are for reading only and cannot be assigned.

1251 Attempt to divide by zero
A zero or a constant with the defined value of zero cannot be placed in the denominator of a fractional expression.

1252 Prototype mismatch with base class of covariant override
Overridden methods that accept object reference types as parameters can be redefined within the same class as the override. This definition must match up with the base class definition in accordance with the rules for covariant methods.

1253 Low value of range is greater than high value
When a range of values is specified in a CASE statement, the second constant must be greater than or equal to the first.

1254 This automatic variable should not be reserved/released
The automatically declared derivative of a continuous array is reserved automatically at the time the continuous variable is reserved. This array should not be reserved by the application code.

1255 This method does not match original declaration.
The heading of the implementation of a method originally declared as a "process" method must have contain the word "process". Methods not declared as process methods must not be implemented as process.

1256 Dollar sign not allowed in names compiled with case sensitivity.
When compiling with the **–s** options for case sensitivity, the dollar sign is not allowed in a routine, variable, attribute, etc name.

1257 BREAK TIES and PRIORITY ORDER statements not allowed with '-h' option.
When compiling with the **–h** option (single event set) the break ties and priority order statements are not currently allowed. These statements must be removed or commented out.

Appendix B Runtime Error Messages

B.1 Runtime Error Messages

When a runtime error is detected, a runtime error message is written to standard error. The text of each message appears below:

2001 zero raised to a negative power

2003 negative number raised to a real power

2004 invalid I/O unit
The unit number is less than 1 or greater than 99.

2005 negative expression in SKIP INPUT statement

2006 attempt to file an entity in a set it is already in
The **M.set** attribute of an entity being **FILE**d in a set is not equal to zero.

2007 attempt to file before or after an entity that is not in the set
The **M.set** attribute of the entity in the **before** or **after** phrase is equal to zero.

2009 attempt to remove from an empty set
The **F.set** attribute is equal to zero when a **remove** operation is attempted.

2010 attempt to remove an entity that is not in a set
The **M.set** attribute is equal to zero when a **remove** specific operation is attempted.

2011 invalid random number stream
The absolute value of the stream number is less than 1 or greater than the number of random number streams (normally 10).

2013 attempt to schedule an event/process already scheduled
The **m.ev.s** attribute of the event/process is not equal to zero when a `schedule` operation is attempted.

2014 attempt to cancel an event/process not scheduled
The **m.ev.s** attribute of the event/process is equal to zero when a **cancel** operation is attempted.

2016 no memory space available
The program is attempting to dynamically allocate more memory than the operating system will allow.

2017 negative argument in itoa.f

2018 argument > 9 in itoa.f

2019 attempt to use a write-only I/O unit for input
An I/O unit opened for output only appears in a **use for input** statement.

2020 attempt to use a read-only I/O unit for output
An I/O unit opened for input only appears in a **use for output** statement.

2021 attempt to use a unit for input that is in the output state
An I/O unit last **used for output** appears in a **use for input** statement without an intervening **rewind**.

2022 attempt to use a unit for output that is in the input state
An I/O unit last **used for input** appears in a **use for output** statement without an intervening **rewind**.

2023 unable to open existing file
See the UNIX error message on the line following this message for more information.

2024 unable to create new file
See the UNIX error message on the line following this message for more information.

2025 subscript out-of-range
An array subscript is less than 1 or greater than the number of array elements.

2027 range error on computed go to
The index value used in a computed **go to** statement is less than 1 or greater than the number of labels.

2028 formatted read goes beyond the end of input record
An attempt is made to read characters beyond the record size specified for the unit.

2030 formatted write goes beyond the end of output record
An attempt is made to write characters beyond the record size specified for the unit.

2032 negative field width in input format

2036 negative field width in output format

2040 mixed binary and character I/O
An I/O operation allowed only on an ASCII file is attempted on a binary file, or vice versa.

2041 invalid character while reading 'C' format
A character is read which is not one of the following: blank, 0-9, A-F, or a-f.

2044 output format field width greater than record size

2048 input format field width greater than record size

2051 zero entity pointer
The pointer used to identify a temporary entity is equal to zero.

2052 reference to destroyed entity
This error can be caused by keeping copies of an entity pointer in several variables, destroying one copy, and referencing attributes of another copy. This error is detected by the runtime checking option. If the option (**-C**) is omitted, a "bus error" may occur instead, or bad values may enter a computation, causing a delayed failure. This is actually a special case of error "2053: invalid entity pointer." It is not always possible to detect a destroyed entity, since the memory may have been reused since it was destroyed. If this is the case, you will get error 2053 instead.

2053 invalid entity pointer
The pointer used to identify a temporary entity does not contain the address of a temporary entity.

2054 wrong temporary entity class
The pointer used to identify a temporary entity contains the address of a temporary entity which belongs to an entity class different from the one that was expected.

2058 reference to unreserved array
The pointer used to identify an array is equal to zero.

2060 zero or negative subscript specification in reserve statement
The number of array elements specified in a **reserve** statement is less than 1.

2061 dim.f for array is > 65535
The number of array elements specified in a **reserve** statement is greater than 65535.

2062 attempt to create invalid entity class
The entity class is not recognized when attempting to **create** an entity, which is usually caused by failing to link the compiler-generated routine **setup.r**.

2066 invalid array pointer
The pointer used to identify an array does not contain the address of an array.

2067 reference to a released array.
This error also appears for references to attributes of a permanent entity that has been destroyed. The error is detected by the runtime checking option. The comments that apply to destroyed entities apply here as well.

2068 end of file encountered during read operation while eof.v=0

2069 fatal I/O error during read
See the UNIX error message on the line following this message for more information.

2070 fatal I/O error during write
See the UNIX error message on the line following this message for more information.

2071 record length exceeds specified recordsize
A record is read from the current input unit, which is longer than the record size specified for the unit.

2072 'B' format input column is not within record
The column number is less than 1 or greater than the record size specified for the unit.

2076 'B' format output column is not within record
See error 2072.

2077 incomplete record on a fixed format file
The last record read from a binary file is shorter than the record size specified for the unit.

2084 invalid character in 'I' format during input
A character is read which is not one of the following: blank, +, -, or 0-9.

2088 integer number too large for input
A value is read which falls outside the range of **integer** values: **-2147483648** to **+2147483647**.

2093 attempt to create text string > 32,000 characters

2094 attempt to erase non-text entity
A value which is not **text** is encountered in a situation where a **text** value is required.

2095 position zero or negative in substr.f

2096 length negative in substr.f

2097 offset negative in match.f

2101 transfer to missing case in select
In a **select** statement, the expression is not equal to any of the values specified in any of the **case** statements and no **default** statement has been specified.

2103 wild transfer in subprogram variable CALL
The value of the **subprogram** variable is not equal to the address of a routine.

2104 wild transfer in subscripted go to statement
An attempt is made to **go to** an undefined subscripted label.

2106 attempt to suspend when no process is active
A **wait, work, suspend, request** or **relinquish** statement is executed by a routine which is neither a process nor a routine called from a process.

2107 attempt to relinquish more resources than requested
An attempt is made to **relinquish** units of a resource that were not previously obtained by a **request**.

2112 parameter 2 negative in 'D' or 'E' format
A negative number of decimal places is specified.

2116 parameter 2 > parameter 1 in 'D' or 'E' output format
The number of decimal places exceeds the total width of the field.

2122 parameter 2 > parameter 1 in 'D' or 'E' input format
See error 2116.

2124 real number too large for input
A value is read which falls outside the range of **double** values.

2128 invalid character in 'D' or 'E' format during input
A character is read which is not one of the following: blank, period, +, -, E, e, or 0-9.

2130 negative argument to skip fields — cannot skip backwards

2132 mean in exponential.f call ≤ 0

2133 mean in erlang.f call ≤ 0

2134 number of stages in erlang.f call ≤ 0

2135 mean in log.normal.f call ≤ 0

2136 standard deviation in log.normal.f call ≤ 0

2137 standard deviation in normal.f call ≤ 0

2138 mean in poisson.f call ≤ 0

2139 second parameter less than first in randi.f call

2140 second parameter less than first in uniform.f call

2141 number of trials in binomial.f call ≤ 0

2142 probability in binomial.f call ≤ 0

2143 shape parameter <= 0 in weibull.f call

2144 scale parameter ≤ 0 in weibull.f call

2145 mean in gamma.f ≤ 0

2146 shape parameter in gamma.f ≤ 0

2147 first parameter in beta.f call ≤ 0

2148 second parameter in beta.f call ≤ 0

2152 value of log.e.f or log.10.f argument ≤ 0

2153 absolute value of arcsin.f or arccos.f argument > 1

2154 values of arctan.f arguments = (0,0)

2155 value of sqrt.f argument < 0

2159 negative time expression in call of nday.f

2160 negative time expression in call of weekday.f

2161 negative time expression in call of hour.f

2162 negative time expression in call of minute.f

2169 (minimum ≤ mean ≤ maximum) is false in triang.f

2171 attempt to open a unit already open

2173 invalid recordsize in open statement
The record size is less than 1 or greater than 65534.

2176 attempt to close a file already closed
An attempt is made to **close** or **rewind** a unit that is not open.

2177 attempt to close a standard SIMSCRIPT unit
An attempt is made to **close** or **rewind** unit 5, 6 or 98**.**

2178 unable to close file
See the UNIX error message on the line following this message for more information.

2185 unable to record memory

2186 unable to restore memory

2188 unable to reopen or reposition a file during restore memory

2193 system service error
For VMS systems only - unexpected error condition from VMS received by SIMSCRIPT

library procedure.

2213 Origin.r must be called before calendar functions

2217 negative argument to out.f
An attempt is made to reference a column position less than 1.

2218 argument to out.f exceeds buffer length
An attempt is made to reference a column position greater than the record size specified for the unit.

2220 simulation time decrease attempted
The value of **time.v** has decreased since the last event occurred.

2221 no event/process to match name in external event data
The external event data contains the name of an external event/process, which has not been defined in the preamble.

2222 invalid external event name

2224 error in use of calendar time format

2225 attempt to destroy an entity owning a non-empty set
An **F.set** attribute of the entity is not equal to zero when a `destroy` operation is attempted.

2226 attempt to destroy an entity that is in a set
An **M.set** attribute of the entity is not equal to zero when a **destroy** operation is attempted.

2227 attempt to use a random variable that has not been read

2228 Alpha probability encountered in random variable data

2229 probability not between 0.0 and 1.0 in random variable data

2230 end of file while reading value field in random variable data

2231 Alpha value encountered in random variable data

2232 Real value where integer expected in random variable data

2233 first cumulative probability not zero in data for random linear variable

2234 cumulative probability values not in increasing order

2235 individual probability values not allowed for random linear variables

2236 sum of probability values more than 1 plus rounding margin

2237 Jump to missing Here statement
See compilation warning.

2238 Time.v decreased since last reset

2239 month origin error
A month is specified which is less than **1** or greater than **12**.

2240 day origin error
A day of the month is specified which is less than **1** or greater than the number of days in the month.

2241 invalid event/process class
An event/process class is specified which is less than **1** or greater than the number of event/ process classes.

2242 year origin error, year must contain four digits, like 1998
A year is specified in the pre-millennium two digit style (i.e. '97'). Years must include all places.

2243 zero object pointer
Access is made to an object attribute using a null pointer as the reference variable.

2244 reference to destroyed object
Access is made to an object attribute using an pointer to an object that has already been destroyed.

2245 invalid object pointer
Access is made to an object attribute using a pointer to something other than an object pointer.

2246 wrong class for attribute or method
Access is made to an object attribute that does not belong to the class of the reference variable or any of its sub-classes.

2247 override of class is required
There are some cases where an object must declare an override of a given method. (See compile-time error #1188). However, methods declared in a private preamble of a base-class are not known to the compiler if that base-class is declared in a different subsystem. These errors are therefore detected at run-time when the program is first initialized.

2248 cycle detected in inheritance, class cannot be derived from itself
A class that is a base-class cannot be derived from the class that it derived from itself. Normally these errors are detected at compile time. However, due to the fact that private preambles provide true data hiding, compile time detection of this case is not possible in every circumstance. A runtime mechanism is employed to check for this case when the program is initialized.

2249 left method called but not implemented

When implementing a method, a right and/or left methods can be provided in the implementation. The compiler will allow a function method to placed on the left side of an assignment, but since it does not know the implementation it cannot flag the lack of a left implementation at compile-time. This case is therefore detected at run-time.

2250 right method called but not implemented"
When implementing a method, a right and/or left methods can be provided in the implementation. The compiler will allow a function method to placed on the right side of an assignment, but since it does not know the implementation it cannot flag the lack of a right implementation at compile-time. This case is therefore detected at run-time.

2251 wrong class for operation
When the statement "destroy a <class> called <reference_var>" is executed, the runtime library will check to make sure that the object pointed to by <reference_var> is the same as or derived from <class>.

2252 exceeded maximum number of events allowed in student version
The maximum number of process notices allowed over the lifetime of the model is 10000 for the student version of SIMSCRIPT III

2253 attempt to release an array containing a non-empty set
When a set with dimensionality > 0 is released, every element must be empty.

2254 attempt to create a negative number of permanent entities
The **n.perm_entity** variable must be greater than or equal to zero before a "create each" statement is encountered.

2255 attempt create a permanent entity that already exists
A **create each** statement is executed that refers to an existing permanent entity.

2256 attempt to start simulation when a simulation is already running
A **start simulation** statement was encountered with a simulation already running.

2257 a simulation must be running to complete this operation
Operations that refer to a current process notices are not allowed if there is no simulation running.

2258 object reference pointer mismatch
If the **called** keyword is used in with the **cause** statement, the runtime library will check that the process notice matches the reference type specified.

2259 external unit variable cannot be assigned during a simulation

2260 external unit cannot be read from at this time
This error can occur if the external unit if reading from an external unit that is out of data.

2261 attempt to reserve an array which is already reserved
Re-reserving a previously reserved array is not supported. The existing array should be released before being reserved.

2300 graphics system error
See the error message on the line preceding this message for more information.

2301 value of vxform.v is invalid
The number of the current viewing transformation is less than **1** or greater than **15** when an attempt is made to define a window or viewport.

2302 invalid viewport dimensions
An attempt is made to define a viewport having dimensions, which do not satisfy the following requirement:

$0 \le xlo \le xhi \le 32767$ and $0 \le ylo < yhi \le 32767$

2303 invalid window dimensions
An attempt is made to define a window having dimensions, which do not satisfy the following requirement:

$xlo \ne xhi$ and $ylo \ne yhi$

2304 attempt to delete the open segment

2305 segment already open
An attempt is made to open a segment when there already is an open segment.

2306 segment already closed
An attempt is made to close a segment when there is no open segment.

2307 segment does not exist

2308 invalid segment priority
The segment priority is less than zero or greater than 255.

2309 invalid POINTS argument
The **points** array is unreserved or does not contain enough points.

2310 form/graph/icon not found
The name specified in a **show** statement does not match any graphic templates in the *graphics.sg2* file or currently loaded .sg2 file.

2311 graph not present
A dynamic histogram has been updated that does not have a visible graph associated with it.

Appendix C Standard SIMSCRIPT III Names

Standard SIMSCRIPT III names are in the standard library.m

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

A.1 Mode Conversion

---

atot.f (*alpha_arg*)

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

---

int.f (*double_arg*)

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

---

itoa.f (*integer_arg*)

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

---

itot.f (*integer_arg*)

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

---

real.f (*integer_arg*)

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

---

rtot.f (*double_arg, total_width, fractional_width, use_exponential*)

A text function that returns the floating-point representation of ***double_arg***. The total width in places of the resulting text string is given as the second argument. That is followed by the number of places to the right the decimal and followed by a flag to use exponential notation (0 or 1). For example, **rtot.f(3.0008, 5, 3, 0)** returns **3.001**.

---

trunc.f (*double_arg*)

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

---

ttoa.f (*text_arg*)

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

---

ttoi.f (*text_arg*)
Converts the text representation of an integer to its integer value and returns it. If the text cannot be converted, zero is returned.

---

ttor.f (*text_arg*)
Converts the text representation of a floating point number to its double value and returns it. If the text cannot be converted, zero is returned.

A.2 Numeric Operations

---

abs.f (*numeric_arg*)

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

---

and.f (*integer_arg1, integer_arg2*)

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

---

arccos.f (*double_arg*)

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

---

arcsin.f (*double_arg*)

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

---

arctan.f (*double_argY, double_argX*)

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

---

cos.f (*double_arg*)

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

---

dim.f (*array_arg*)

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

---

div.f (*integer_arg1, integer_arg2*)

An integer function that returns the truncated result of **(*integer_arg1* / *integer_arg2*)**. ***Integer_arg2*** must be nonzero. For example, **div.f(17, 5)** returns 3 and **div.f(–12, 8)** returns –1.

---

exp.c

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

---

exp.f (*double_arg*)

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

---

frac.f (*double_arg*)

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

---

inf.c

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

---

log.e.f (*double_arg*)

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

---

log.10.f (*double_arg*)

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

---

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

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

---

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

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

---

mod.f (*numeric_arg1, numeric_arg2*)

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

---

or.f (*integer_arg1, integer_arg2*)

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

---

pi.c

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

---

radian.c

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

---

rinf.c

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

---

shl.f (*integer_arg1, integer_arg2*)

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

---

shr.f (*integer_arg1, integer_arg2*)

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

---

sign.f (*double_arg*)

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

---

sin.f (*double_arg*)

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

---

sqrt.f (*double_arg*)

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

---

tan.f (*double_arg*)

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

---

xor.f (*integer_arg1, integer_arg2*)

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

A.3 Text Operations

---

concat.f (*text_arg1, text_arg2, …*)

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

---

fixed.f (*text_arg, integer_arg*)

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

---

length.f (*text_arg*)

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

---

lower.f (*text_arg*)

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

---

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

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

---

repeat.f (*text_arg, integer_arg*)

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

---

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

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

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

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

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

If the value assigned to the substring is not the same length as the substring, then space characters are appended to, or trailing characters are removed from, the assigned value. ***Integer_arg1*** must be positive and ***integer_arg2*** must be nonnegative. If ***integer_arg1*** is greater than the length of ***text_arg***, or ***integer_arg2*** is zero, then a null string is returned when **substr.f** is called as a right function, and no modification is made to ***text_arg*** when **substr.f** is called as a left function.

---

trim.f (*text_arg, integer_arg*)

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

---

upper.f (*text_arg*)

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

A.4 Input/Output

---

buffer.v

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

---

efield.f

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

---

eof.v

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

---

heading.v

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

---

line.v

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

---

lines.v

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

---

mark.v

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

---

out.f (*integer_arg*)

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

---

page.v

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

---

pagecol.v

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

---

rcolumn.v

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

---

read.v

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

---

record.v (*integer_arg*)

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

---

ropenerr.v

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

---

rreclen.v

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

---

rrecord.v

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

---

sfield.f

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

---

wcolumn.v

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

---

wopenerr.v

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

---

wrecord.v

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

---

write.v

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

A.5 Random-Number Generation

---

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

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

---

binomial.f (*integer_arg1, double_arg, integer_arg2*)

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

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

---

erlang.f (*double_arg, integer_arg1, integer_arg2*)

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

---

exponential.f (*double_arg, integer_arg*)

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

---

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

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

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

---

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

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

---

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

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

---

poisson.f (*double_arg, integer_arg*)

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

---

randi.f (*integer_arg1, integer_arg2, integer_arg3*)

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

---

random.f (*integer_arg*)

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

---

seed.v

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

---

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

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

---

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

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

---

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

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

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

A.6 Simulation

---

between.v

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

---

clearevents.r

This routine is a utility that can be called to remove and destroy all remaining process notices in the event set(s).

---

date.f (*integer_arg1, integer_arg2, integer_arg3*)

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

---

day.f (*double_arg*)

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

---

ev.s

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

---

event.v

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

---

events.v

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

---

f.ev.s

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

---

hour.f (*double_arg*)

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

---

hours.v

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

---

l.ev.s

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

---

minute.f (*double_arg*)

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

---

minutes.v

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

---

month.f (*double_arg*)

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

---

n.ev.s (*integer_arg*)

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

---

nday.f (*double_arg*)

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

---

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

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

---

process.v

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

---

time.v

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

---

weekday.f (*double_arg*)

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

---

year.f (*double_arg*)

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

A.7 Miscellaneous

---

batchtrace.v

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

---

date.r yielding *text_arg1*, *text_arg2*

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

---

err.message.f

A left function that can be assigned a text string in the event of an error detected while running the program. The error message will be printed and the debugger will be invoked.

---

exit.r (*integer_arg*)

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

---

high.f

Returns the upper index boundary of an array. "DIM.F" will be returned unless the array is reserved using the **reserve** statement in conjunction with **to** keyword. I.e. "**reserve ARR(*) as –10 to 10"**

---

low.f

Returns the lower index boundary of an array. "1" will be returned unless the array is reserved using the **reserve** statement in conjunction with **to** keyword. I.e. "**reserve ARR(*) as –10 to 10"**

---

parm.v

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

---

snap.r

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

---

wordsize.f

Returns 64 for 64-bit simscript and 32 for 32-bit simscript

Appendix D Latin 1 Character Set

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

À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ ß

à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ

and these special symbols:

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

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

ASCII Character Set

0	NULL	32	Space	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40		(72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	ú
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Appendix E Deprecated SIMSCRIPT II.5 features

SIMSCRIPT III is a superset of SIMSCRIPT II.5. Using language features of SIMSCRIPT II.5 that have been deprecated is allowed. These features will continue to function normally and be supported, but a warning message will be printed by the compiler whenever a deprecated feature is compiled.

Below, each deprecated feature is described followed by examples of code using the feature, and a suggested modification to the code.

Modification of the code is recommended, but if not possible, deprecation warnings can be suppressed by compiling using compiler switch "-w1245".

All deprecated warnings/errors can also be suppressed using compatibility switch "–e".

The deprecation warnings will NOT appear when compiling with the SIMSCRIPT II.5 compatibility switch ("-e"). When this switch is used, statements will exhibit the behavior described in SIMSCRIPT II.5 Language and Reference Manuals.

RELEASABLE routines

Defining a routine as "releasable" has no effect. The "releasable" keyword can be safely removed.

Old code:

```
define RTN as a releasable routine
```

New code:

```
define RTN as a routine
```

DUMMY variables

A dummy variable can be used like a traditional variable, but no storage is allocated to the variable. Such variables must be monitored on the left and/or right sides to be usable. The same functionality can be achieved by defining the name as a function instead of a variable.

Old code:

```
every TE has a X
define X as an integer dummy variable monitored on the left and right
```

New code:

```
every TE has a X function
define X as an integer function
…
routine X(TE.PTR)
   …
end
left routine X(TE.PTR)
   …
end
```

Using the ERASE statement for a text variable

The erase statement is sometimes used to clear a text variable. Assigning the empty string “” to a text variable has the same effect.

Old code:

```
erase T
```

New code:

```
let T = ""
```

Graphic input and output units

Using a unit for graphic input and output is a relic of 16 bit Windows SIMSCRIPT and has no effect in modern SIMSCRIPT II.5. For compatibility, the syntax remains in the language. These statements can be removed.

Old code:

```
use unit 8 for graphic output
use unit 7 for graphic input
```

New code:

```
<remove the statements>
```

Event routines

Event routines can be scheduled for future activation like process routines, but cannot elapse time (i.e. cannot include the “wait” or “work” statement). It is not a requirement that process routines elapse time, so event routines can be re-declared as process routines.

Old code:

```
preamble
events
   every SALE has a PRODUCT.TYPE, a PRICE and a QUANTITY
end
…
event SALE
…
end
```

New code:

```
preamble
processes
   every SALE has a  PRODUCT.TYPE, a PRICE and a QUANTITY
end
…
process SALE
…
end
```

External events

External events can be replaced with external processes.

Old code:

```
external event is SALE
external event unit is 3
…
if event is external
…
```

New code:

```
external process is SALE
external process unit is 3
…
if process is external
```

NORMALLY, TYPE IS UNDEFINED statement

The NORMALLY, TYPE IS… statement is used to force the specification of either RECURSIVE or SAVED clauses when defining a local variable. The statement can be removed.

Old code:

```
normally, type is undefined
```

New code:

```
<remove the statement>
```

Comparison operators @=, @>, and @<

The operators @=, @>, and @< are used to mean “not equal”, “not greater than” and “not less than” respectively. They can be replaced with the equivalent operators such as <>, <=, and >=.

Old code:

```
if X @> Y
```

New code:

```
if X <= Y
```

OLD, VERY OLD, and NEW preamble declaration

The behavior of clauses OLD, VERY OLD and NEW when preceding a preamble declaration is unclear and not implemented consistently. These keywords can be removed.

Old code:

```
very old preamble
…
end
```

New code:

```
preamble
…
end
```

The STORE statement

In very early implementations of SIMSCRIPT II.5 the STORE statement was used to assign variables without automatic mode conversion. In SIMSCRIPT II.5 the statement is identical to a LET statement and can be replaced

Old code:

```
store 5 in X
```

New code:

```
let X = 5
```

Bit, field, and intra packing

Packing is designed to allow more than one attribute or array element to share the same memory location. For bit packing the bit location and number of bits that an attribute takes up in a 32-bit word can be specified. Field packing allows the attribute to be assigned to individual bytes within the 32-bit word. With intra-packing, each 32-bit word of an array can hold more than one element. Packing specifications can usually be removed from the preamble without affecting the application. However, there are the following rare exceptions:

a) Overlapped packing – more than one variable needs to shares the same bits.
b) Memory usage – Packing used to save memory in an application where every bit is critical.
c) Linking with other languages – A specific entity structure is required "bit for bit" by part of the application written in a different language (like C/C++).
d) Packed attribute is written to a file whose format dictates that data elements are to be bit or field packed.

Old code:

```
temporary entities
   every TE has
      a (FIELD.PACK1(1/4),
          FIELD.PACK2(2/4),
          FIELD.PACK3(3/4)),
      a (BIT.PACK1(1-8), BIT.PACK2(9-16))
…
 the system has
    a INTRA.PACKED(*/4)
```

New code:

```
Temporary entities
   Every TE has
      a FIELD.PACK1,
      a FIELD.PACK2,
      a FIELD.PACK3,
      a BIT.PACK1,
      a BIT.PACK2
…
 The system has
    a INTRA.PACKED
```

Attribute equivalencing

An individual entity attribute can be assigned multiple names using attribute equivalencing. Equivalencing can be eliminated by removing the aliases in the preamble and changing the implementation code to refer to only one name. If there are too many references to the alias name in the implementation code, a substitution can be used in the preamble to redefine each alias to be the original variable.

Old code:

```
every DOG.HOUSE has
   a (DOG, CANINE, POOCH)
…
let CANINE(DOG.HOUSE) = MY.DOG  ''same as LET DOG(DOG.HOUSE) =
let POOCH(DOG.HOUSE) = MY.DOG  ''same as LET DOG(DOG.HOUSE) =
```

New code:

```
every DOG.HOUSE has
   a DOG
define POOCH to mean DOG
define CANINE to mean DOG
```

Word and array assignment of attributes

SIMSCRIPT allows the programmer to control the memory assignment of both entity and system attributes using the “IN WORD” clause where the attribute is declared. The IN WORD clause can be safely eliminated unless there is a compatibility issue with another language that somehow depends on the specific in-memory layout of SIMSCRIPT entities.

Old code:

```
every SHIP has
   a WEIGHT in word 1,
   a LENGTH in word 2
```

New code:

```
every SHIP has
   a WEIGHT,
   a LENGTH
```

The FIN statement

The FIN statement is a synonym for ALWAYS or ENDIF and can be replaced by either.

Old code:

```
if X <> 0
   list X
fin
```

New code:

```
if X <> 0
   list X
always
```

Using the FIND statement with embedded assignments

The FIND statement permits embedded assignment statements that are executed the first time the condition given in the statement is true. The same result can be achieved by moving the assignments after an IF FOUND clause.

Old code:

```
for each ITEM in COLLECTION with ACTIVE(ITEM) <> 0
   find ACTIVE.ITEM = ITEM, ID = ID.TAG(ITEM)
```

New code:

```
for each ITEM in COLLECTION with ACTIVE(ITEM) <> 0
   find the first case
if found
   let ACTIVE.ITEM = ITEM
   let ID = ID.TAG(ITEM)
always
```

Defining sets without attributes and routines

Sets can be defined to specifically exclude attributes and functionality that is normally provided automatically. F., L., N., attributes can be eliminated from the owner, and P., S., and M. attributes can be eliminated from member entities. Individual set routines that are scripted to handle the various types of file and remove operations (FF, FL, FB, FA, RF, RL and RS) can also be eliminated. The WITHOUT clause can be removed without affecting the behavior of the set.

Old code:

```
define COLLECTION as a lifo set
   without L., M. attributes without FL, RL routines
```

New code:

```
define COLLECTION as a lifo set
```

Mixed compound entities

SIMSCRIPT II.5 permits compound entities that are composed either of both permanent and temporary entities or of only temporary entities. Each attribute of a mixed compound entity must be a function attribute. The mixed compound entity specification can be removed and its attributes replaced by functions which accept the former component entity types as parameters.

Old code:

```
temporary entities
   every SOURCE has a LINK.LIST
   every DESTINATION has a NAME
   every SOURCE, DESTINATION has
      a DISTANCE function
   define DISTANCE as a double function
```

New code:

```
temporary entities
   every SOURCE has a LINK.LIST
   every DESTINATION has a NAME
define DISTANCE as a double function given
   1 SOURCE reference argument,
   1 DESTINATION reference argument
```

Use of a right monitored variable as a subprogram literal

Using any right monitored variable as a subprogram literal automatically refers to the right monitoring routine for the variable. For the same results, the monitoring routine can be replaced with a traditional function with left and right implementations. These implementations refer to a separate variable to hold the data.

Old code:

```
preamble
define X as a double variable monitored on the right
end
right routine X
   …
end
…
define subv as a double subprogram variable
define result as a double variable
let subv = ' X '
let result = $subv
```

New code:

```
preamble
define XV as a double variable
define X as a double function
end
right routine X
   …
end
left routine X
   enter with XV
end
…
define subv as a double subprogram variable
define result as a double variable
let subv = ' X '
let result = $subv
```

The LAST COLUMN statement

The LAST COLUMN statement specifies the maximum length of a line of code in the program. Lines longer than the specified value are truncated automatically by the compiler. "Last column" statements can be safely removed.

Old code:

```
last column is 80
```

New code:

```
<remove the statement>
```

Use of the STA.A attribute

STA.A is an attribute of a process notice used to indicate the current condition of the process. Using the WAIT statement automatically sets the process status (STA.A attribute) to 0. Using the WORK statement sets STA.A = 1. The value of STA.A has no effect on scheduling. An application defined status attribute can be used to replace STA.A.

Old code:

```
preamble
   processes include GENERATOR,LATHE
end
process LATHE
   wait 5 units
   work 5 units
end
process GENERATOR
```

```
   if STA.A(LATHE) <> 0
      write as “Lathe is working”, /
   else
      write as “Lathe is waiting”, /
   always
end
```

New code:

```
preamble
   processes include GENERATOR
      every LATHE has a STATUS
end
process LATHE
   let STATUS = 0
   wait 5 units
   let STATUS = 1
   work 5 units
end
process GENERATOR
   if STATUS(LATHE) <> 0
      write as “Lathe is working”, /
   else
      write as “Lathe is waiting”, /
   always
end
```

The GENERATE/INHIBIT LIST ROUTINES statement

This statement performs no function and can be removed from the preamble.

Old code:

```
preamble
inhibit list routines
end
```

New code:

```
<remove the statement>
```

The REWIND statement

The REWIND statement is equivalent to closing the unit number provided in the statement. It can be replaced with a CLOSE statement

Old code:

```
rewind unit 3
```

New code

```
close unit 3
```

Archaic options for the OPEN statement

Several of the options used in the OPEN statement have been deprecated which include:

```
SYNCHRONOUS, ASYNCHRONOUS, REOPENABLE, REPOSITIONABLE, VARIABLE,
FIXED, UNDEFINED, NUMBERED, SPANNED, ASA, DEVICE, SAVE, DELETE,
PRINT, NOPAD, STRIP, PASS, INSERT, LOCK, TEMPORARY.
```

The following options are not deprecated:

```
APPEND,   BINARY,   CHARACTER/FORMATTED,   NOERROR,   NAME,   and
RECORDSIZE.
```

Archaic options for the CLOSE statement

Options given to the CLOSE statement have been deprecated. These options include the following:

```
SAVE, DELETE, PRINT, NAME IS, UNLOCK.
```

Report generation statements

These statements include

```
BEGIN REPORT,
BEGIN HEADING
IF PAGE IS FIRST
PRINT ... A GROUP OF ... FIELDS SUPPRESSING FROM ...
```

Refer to the SIMSCRIPT II.5 documentation for a description of the above features.

Appendix F Non-Simscript routines can be case sensitive

In SIMSCRIPT III Handling Non-Simscript routines has been improved as follows:

1) Mixed case routine names can be defined. The nonsimscript routine defined in the preamble should match its definition in C routine. The nonsimscript routine called in the executable should match its definition in the preamble exactly. For example:

```
Preamble
Define MixedCaseRoutine as a nonsimscript routine
End
Main
   Call MixedCaseRoutine(5)
''generated C code:::   MixedCaseRoutine(5);
End
```

2) The argument prototyping enhancement can be applied to nonsimscript routines. Under SIMSCRIPT III caution must be taken when writing the code to call a nonsimscript routine. For example, if the implementation of MixedCaseRoutine takes “double” as its first parameter, and the integer ‘5’ is passed by the caller, no conversion on the argument will take place. (The SIMSCRIPT II.5 compiler has no way of knowing about the details of arguments to MixedCaseRoutine!) In SIMSCRIPT III, each mode of each argument to a nonsimscript routine can be specified explicitly in the preamble. The appropriate conversions of arguments are then performed by the calling function.

```
Preamble
Define MixedCaseRoutine as a nonsimscript routine
   given 1 real argument
End
Main
''generated C code:::    S_VOID MixedCaseRoutine(S_R);
''after seeing this prototype for MixedCaseRoutine, “C” will
''perform the correct conversion to “real”
   Call MixedCaseRoutine(5)
End
```

The prototype of input arguments is optional, though recommended. The prototype of output arguments is a must, to avoid error reported by the compiler.

3) The 'yielding' clause can be used in conjunction with the argument prototyping: A nonsimscript routine can be defined as "yielding" one or more arguments. This allows arguments to be passed to the nonsimscript routine by reference. For example, suppose that MixedCaseRoutine was prototyped in the "C" code as follows:

```
void MixedCaseRoutine(double *dp);
```

It should then be defined in the Preamble as:

```
Preamble
Define MixedCaseRoutine as a nonsimscript routine yielding
   1 double argument
End
```

The nonsimscript routine can be called like a prototyped SIMSCRIPT routine. Modes of variables used by the caller do not have to match exactly modes used by the callee. In the following code, an integer mode variable is yielded. Since the nonsimscript routine was prototyped in the preamble, the appropriate conversion takes place.

```
Main
Define V1 as an integer variable
Call MixedCaseRoutine yielding V1
end
```

4) If input of a nonsimscript routine is un-prototyped, variables passed as arguments that are of mode "REAL" are NOT automatically converted to 64-bit "double". The variable will be passed as a 32-bit float and it is assumed in this case that the "C" routine expects a "32-bit float" and not a "64-bit double".

Appendix G Continuous Simulation

SIMSCRIPT supports discrete and combined discrete/continuous simulation through several unique features: “wait continuously” statement, and capability to define continuous variables. In SIMSCRIPT III these features have been improved. They are re-implemented in a separate system module continuous.m. The public preamble for this module is in SIMSCRIPT directory SIMHOME/defs. Models which use SIMSCRIPT continuous capabilities should state in the preamble the import of this module/subsystem and to import this directory during compilation. All necessary library objects will be automatically linked with the model.
Here is description of the improved wait continuously statement.

Work/wait continuously evaluating ‘rtn’ … statement

When this statement is executed inside a process, that process is suspended allowing continuous attributes of the process notice to be updated. In SIMSCRIPT II.5 continuous attributes of a process are only updated while that process is suspended in a WAIT CONTINUOUSLY statement.

In SIMSCRIPT III, continuous attributes are always updated. Even if the process is suspended in some other way (such as a discrete WAIT/WORK statement). In SIMSCRIPT II.5 an “evaluate” routine can be passed in the WAIT CONTINUOUSLY statement as a subprogram literal. This routine is called frequently by the runtime library allowing the application to assign new values to the continuous attribute derivatives. In SIMSCRIPT III the SystemOfEquations object can be subclassed and its “evaluate” method overridden. Code for derivative evaluation should be placed in this overridden method.

Old code:

```
preamble
   processes
      every BALLISTIC has an X
      define X as a continuous double variable
end
routine EVALUATE.RTN(BALLISTIC)
   let D.X(BALLISTIC) = …
end
process BALLISTIC
   let D.X = 1200
   work continuously evaluating ‘EVALUATE.RTN’
end
```

New code:

```
preamble including the continuous.m subsystem
   processes
      every BALLISTIC has an X
```

```
      define X as a continuous double variable
   begin class Ballistic_Eq
      every Ballistic_Eq is a SystemOfEquations
           and overrides the Evaluate
   end
end
method Ballistic_Eq'evaluate
    let D.X(BALLISTIC) = …
end
process BALLISTIC
   let D.X = 1200
   suspend
end
```