```
process AIRPLANE
     call TOWER giving GATE yielding RUNWAY
     work TAXI.TIME (GATE, RUNWAY) minutes
     request 1 RUNWAY
     work TAKEOFF.TIME (AIRPLANE) minutes
     relinquish 1 RUNWAY
end " process AIRPLANE
```


```
process AIRPLANE
     call TOWER giving GATE yielding RUNWAY
     work TAXI.TIME (GATE, RUNWAY) minutes
     request 1 RUNWAY
     work TAKEOFF.TI
     relinquish 1
end " process AI
```


Since 1962

SIMSCRIPT II.5
User's Manual

CACI

Updated November 2002

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
1011 Camino Del Rio South, suite 230
San Diego, California 92108
Telephone: (619) 542-5224
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
1011 Camino Del Rio South #230
San Diego, CA 92108

Telephone: (619) 542-5224

simscript@caci.com

TABLE OF CONTENTS

PREFACE.......... a

Introduction.......... 1

1 Developing Simulation Models with Simstudio.......... 3

1.1 Simstudio Overview.......... 4

1.2 Creating a New Project.......... 6

1.3 Adding Source code to a Project.......... 7

1.3.1 Creating a New File with the Text Editor.......... 7

1.3.2 Adding a Directory or a File Using Project Window.......... 8

1.3.3 Adding Multiple Directories and Files.......... 10

1.3.4 Adding Graphical Elements to a Project.......... 11

1.4 Opening an Existing Project.......... 12

1.5 Building a Project.......... 13

1.5.1 Building a Project for Debugging.......... 14

1.5.2 Building a Project for Release.......... 16

1.5.3 Compiler Listings.......... 16

1.6 Executing a Model.......... 17

1.6.1 Passing Command-Line Arguments.......... 17

1.6.2 Running the Executable with the Symbolic Debugger.......... 18

1.7 Closing the Project.......... 19

1.8 Setting Simstudio Preferences.......... 20

1.9 On-line Help.......... 20

1.10 Advanced Compiler/Link Options.......... 22

2 Developing Simulation Models Using Command-Line Interface.......... 23

2.1 Preparing Source Files.......... 23

2.2 Compiling.......... 23

2.3 Recompiling.......... 28

2.4 Linking.......... 29

2.5 Executing.......... 31

2.6 Profiling.......... 32

2.7 Makefiles.......... 34

2.7.1 Compilation Sequence.......... 35

2.7.2 Make Description File Format 36
2.7.3 Transformation Rules 36
2.7.4 Special Notes 37
2.7.5 Sample Makefile 37

2.8 Obtaining Online Help 39

2.9 Example Program 39

3 SIMSCRIPT II.5 Language Considerations 42

3.1 Input and Output 42

3.2 Modes and Packing Codes 44
3.2.1 Alignment of Values 45
3.3.1 Calling C Routines 45
3.3.2 Calling FORTRAN Routines 46

4 SimDebug Symbolic Debugger 50

4.1 Compiling for Debug and Invoking SimDebug 50
4.1.1 Compiling for Debug 50
4.1.2 Invoking SimDebug 51

4.2 A Quick Tour of SimDebug 52
4.2.1 Tour 1: Showing the Stack and Variables 52
4.2.2 Tour 2: Breakpoints and Single Stepping 55
4.2.3 Tour 3: Pointer Handling: Entity / Set Display 58

4.4 Advanced Topics 72
4.4.1 Batchtrace.v 72
4.4.2 Signal Handling / External Events 73
4.4.3 Reserved Names 73
4.4.4 Displaying Arrays 73
4.4.5 Permanent Entities and System Owned Variables/Sets 73
4.4.6 Conditional Breakpoints 74
4.4.7 Continuous Variables 74
4.4.8 Unsupported SIMSCRIPT Features 74

Appendix A Compiler Warning and Error Messages 77

Appendix B Runtime Error Messages 91

B.1 Runtime Error Messages 91

Appendix C Standard SIMSCRIPT II.5 Names 101

C.1 Functions and Routines 101

C.2 Global Variables 116

C.3 Attributes **120**
C.4 Constants **121**
Appendix D ASCII Character Set **123**

FIGURES

Figure 1-1 Opened Project in Simstudio with source and graphics windows 5
Figure 1-2 Project tree 7
Figure 1-3 Creating a new source file.. 8
Figure 1-4. Creating a new folder in the project tree 9
Figure 1-5 Project tree with hierarchical organization of source code 10
Figure 1-6 Adding a new Icon in Simstudio 11
Figure 1-7 Selecting Project Options 14
Figure 1-8 Selecting debugging options in Simstudio 15
Figure 1-9 Selecting Release options in Simstudio 16
Figure 1-10 Defining command line for model execution...................................... 18
Figure 1-11 SIMSCRIPT Symbolic Debugger window 19
Figure 1-12 Simstudio on-line help window 21

PREFACE

This document contains information on the use of CACI's SIMSCRIPT II.5 compiler for developing simulation models. Development can be done either using SIMSCRIPT II.5 Development Studio (Simstudio) or Command-line interface.

CACI publishes a series of Manuals and text books that describe the SIMSCRIPT II.5 language and SIMSCRIPTII.5 Simulation Graphics, Development environment, Data Base connectivity, Combined Discrete-Continuous Simulation, etc. All documentation is available on SIMSCRIPT II.5 WEB site http://www.caciasl.com/products/simscript.cfm

- *SIMSCRIPT II.5 User's Manual* — This Manual – A detailed description of the SIMSCRIPT II.5 development environment: usage of SIMSCRIPT II.5 Compiler and the symbolic debugger from the SIMSCRIPT Development studio, Simstudio and from the Command-line interface.

- *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 II.5 language and its implementations are proprietary program products of the CACI Products Company. Distribution, maintenance, and documentation of the SIMSCRIPT II.5 language and compilers are available exclusively from CACI.

Free Trial Offer

SIMSCRIPT II.5 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 II.5 are scheduled on a recurring basis in the following locations:

La Jolla, 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
1011 Camino Del Rio South, suite 230
San Diego, California 92108
Telephone: (619) 542-5228
www.caciasl.com

Introduction

As an aid to making important decisions, the use of computer simulation has grown at an astonishing rate since its introduction. Simulation was first used occasionally in manufacturing, military, nuclear, and a few other pioneering applications. More recently, its use has expanded to many other areas of need. The growing list of successful applications includes models relating to urban growth, hydroelectric planning, transportation systems, election redistricting, cancer and tuberculosis studies, hospital planning, communications, and multi-computer networks. SIMSCRIPT II.5 has been used world wide for building high-fidelity simulation models.

SIMSCRIPT II.5 is a language designed specifically for simulation. It is the most efficient and effective program development technique for simulation. This is due to the following:

- **Portability**. SIMSCRIPT II.5 development environment, which includes SIMSCRIPT II.5 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 II.5 provides constructs designed especially for simulation (e.g., processes, resources, events, attributes, 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 II.5 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 II.5 were designed to make it possible to manage a complicated simulation model.

- **Error Detection.** SIMSCRIPT II.5 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 II.5 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 II.5 includes the **`accumulate`** and **`tally`** statements that allow the model builder to collect statistics on key variables in his model.

• **Report Generator.** A formatted report generator with headings and page numbering, along with the `print` statement, is part of the SIMSCRIPT II.5 language.

• **Simulation Graphics.** Brings interactive animated and display graphics to new and existing SIMSCRIPT II.5 models. Graphical entities can be easily tied to program entities, 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 II.5 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 II.5 Application Program Interfaces (API's) to Operating System Services facilitating portable models across all SIMSCRIPT II.5 supported computer platforms.

• **Open System.** SIMSCRIPT II.5 provides possibility to call non-simscript routines/functions from a SIMSCRIPT model. This facilitates usage of libraries written in C/C++ or FORTRAN from SIMSCRIPT models.

• **Complete Methodology.** The SIMSCRIPT II.5 approach to simulation model development provides the complete set of capabilities needed to develop a simulation model. A simulation model developed in the SIMSCRIPT II.5 programming language is readable by the analyst familiar with the system under study.

• **Support.** CACI provides SIMSCRIPT II.5 software, documentation, training and technical support. Model development services are also available from CACI.

1 Developing Simulation Models with Simstudio

Developing a SIMSCRIPT II.5 model typically involves the following steps:

1. Preparing one or more SIMSCRIPT II.5 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 II.5 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 II.5 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 II.5 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 cmd window. 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 your own favorite text editor edit, vi, etc to create SIMSCRIPT source files. To create graphical elements for your model, you have to use Simstudio graphicals editors. 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 II.5 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.
>
> **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 model build and other temporary files. Contents of this directory are not important to developers.

This project directory structure helps you during development and deployment of your model. Subdirectory sources contain the current version of the model source code, directory executable contains all components necessary for model execution.

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.

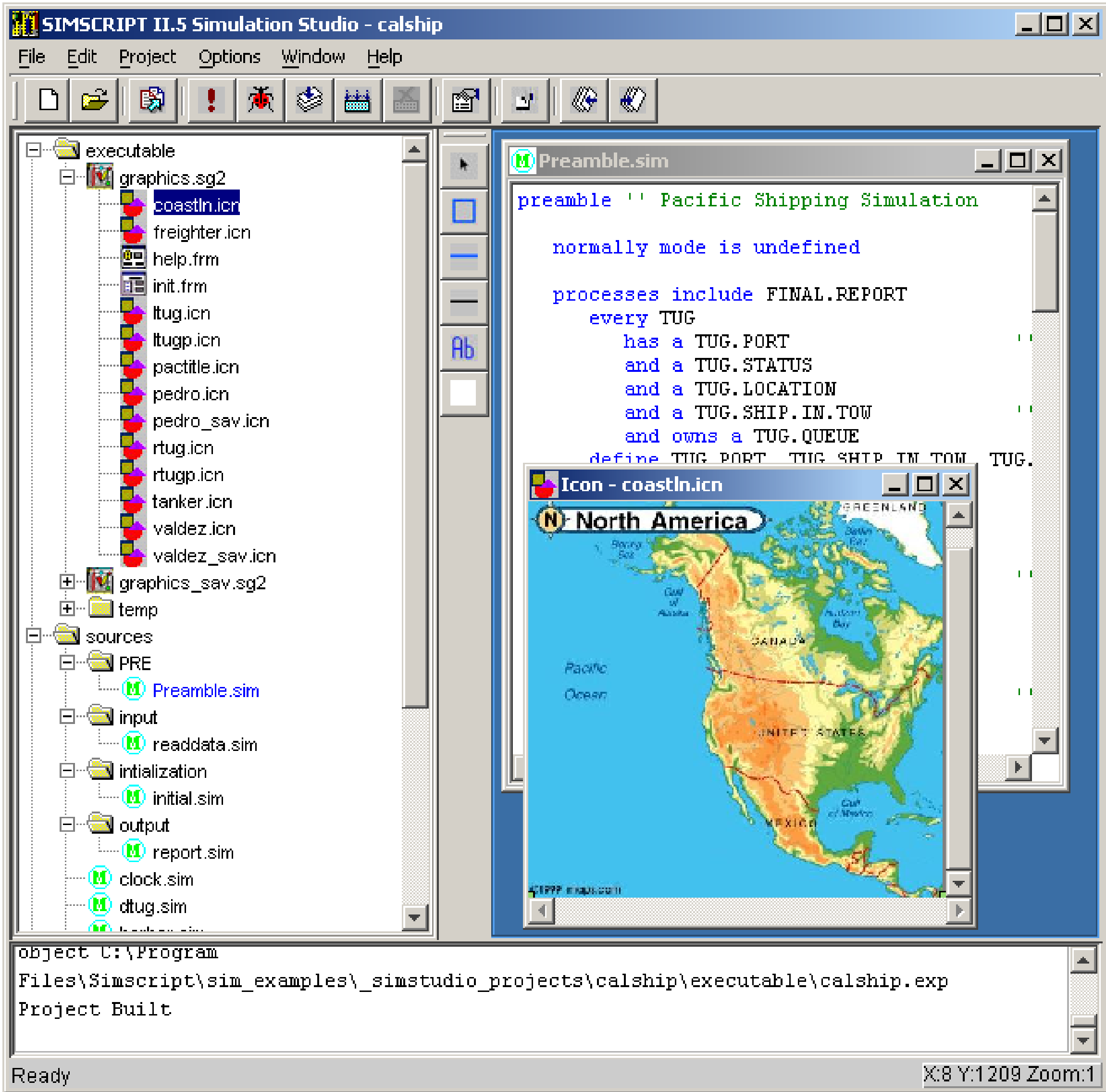


Figure 1-1 Opened Project in Simstudio with source and graphics windows

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.

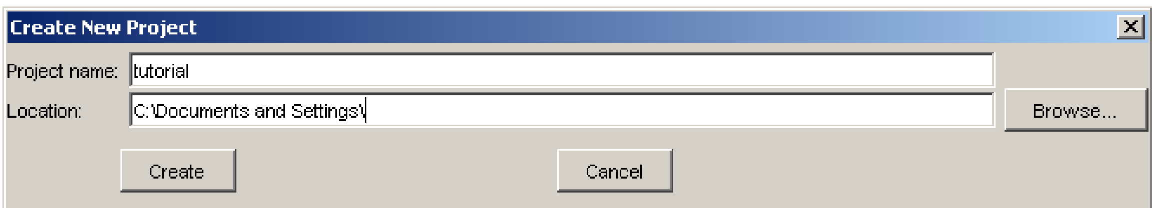
The project tree is composed of source code files with the extension '.sim' in the directory sources. The graphics.sg2 file contains the following graphical elements: icons with extension '.icn', forms with extension '.frm' , and graphs with extension 'grf'. These can be found in the directory executable.

Simstudio incorporates SIMSCRIPT II.5 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.

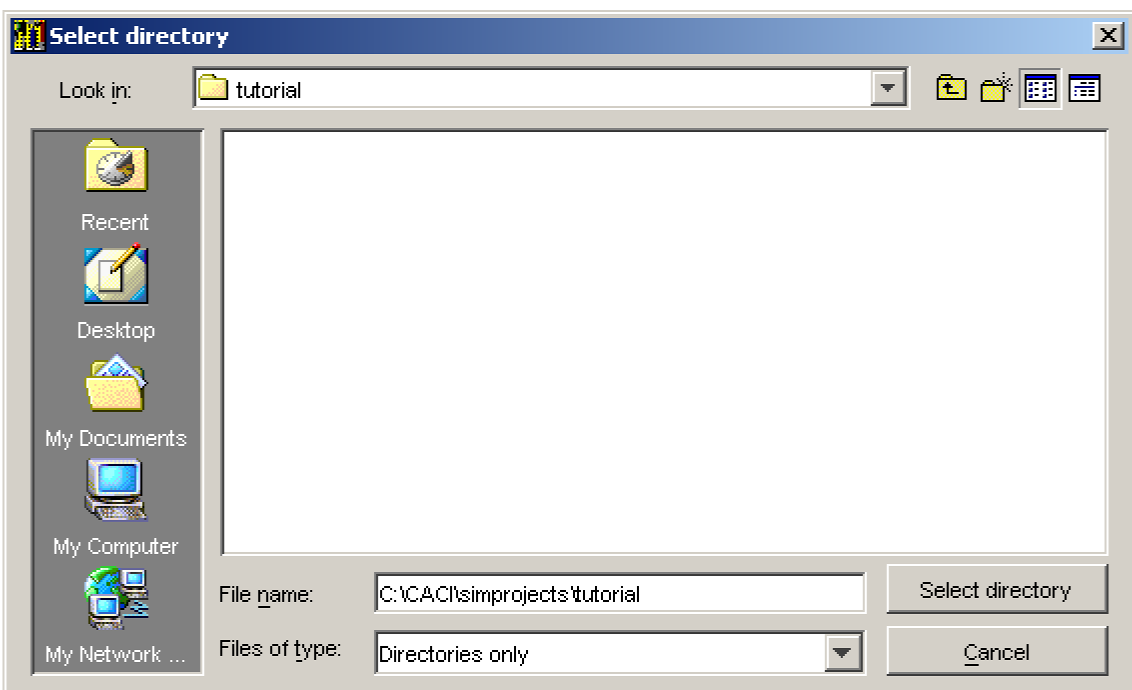
The following sections will describe how a to create projects, add source code and graphical elements, and build and execute the model.

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.



Type in the project name, Click **Browse…**



Go to the directory where the new project will be located, click on **Select Directory** and click **Create**

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

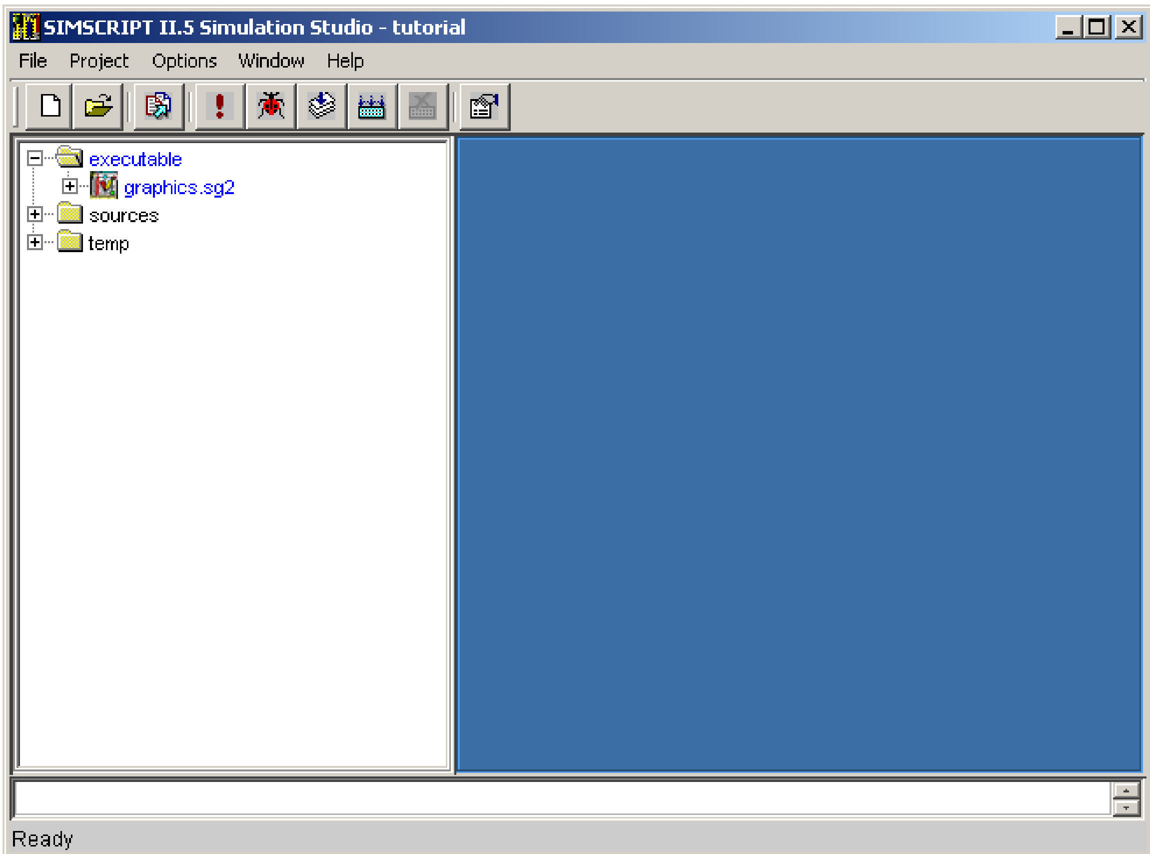


Figure 1-2 Project tree

1.3 Adding Source code to a Project

Source code for projects are 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.

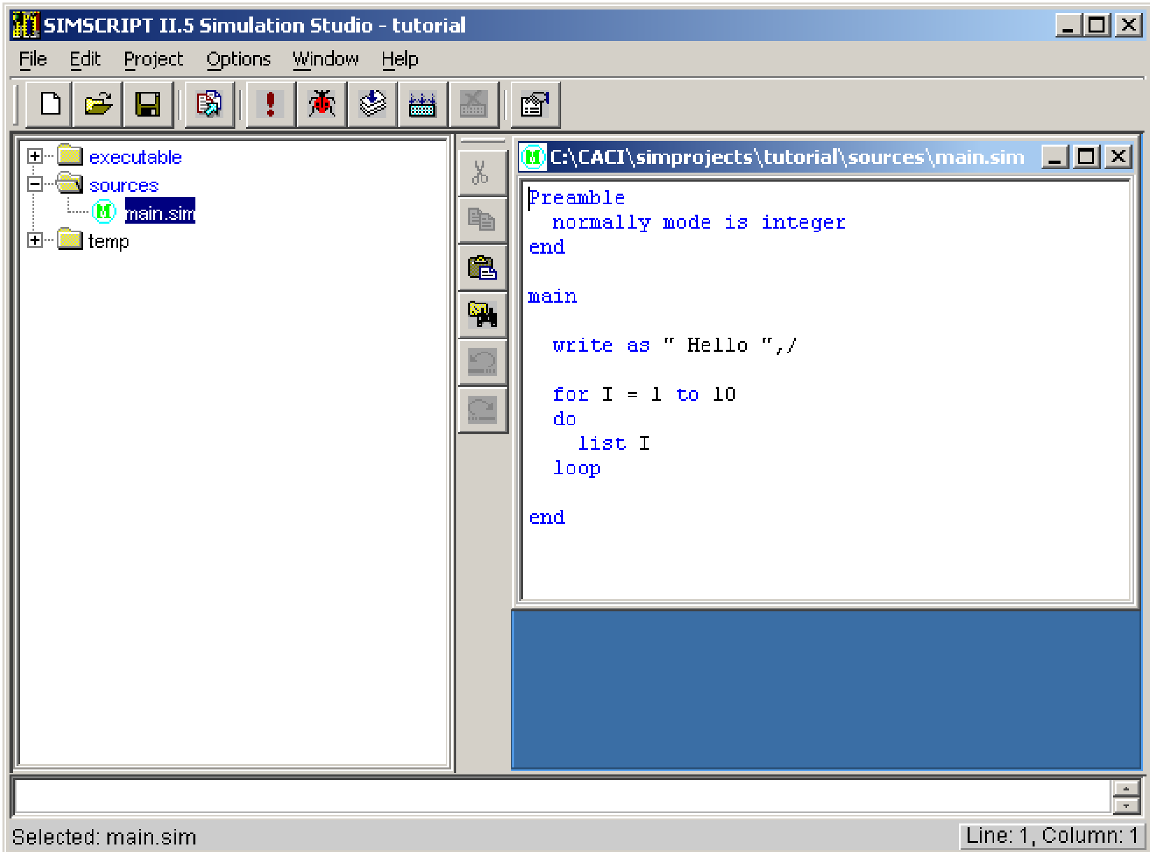


Figure 1-3 Creating a new source file

To open this file again, click on its name twice with the left mouse-button in the project tree. When you open a text file for editing the menu option **Edit, the** 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 mouse click on the directory sources. It will bring a pop-up menu with options: **add files**, **new folder** and **delete**. Chose **new folder**.

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.

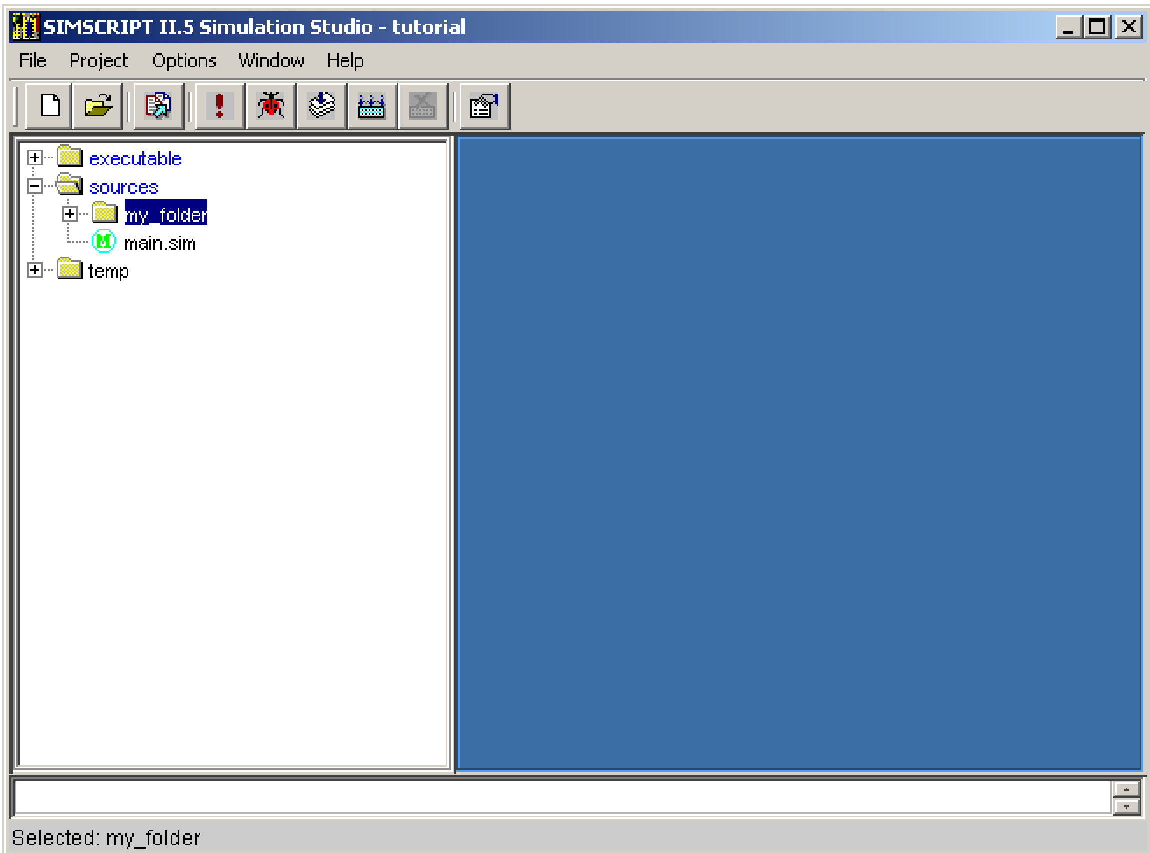


Figure 1-4. Creating a new folder in the project tree

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 the operating system tools to the project sources directory. Use **Project-> Update Project Tree** to include all directories and files for the project and project tree.

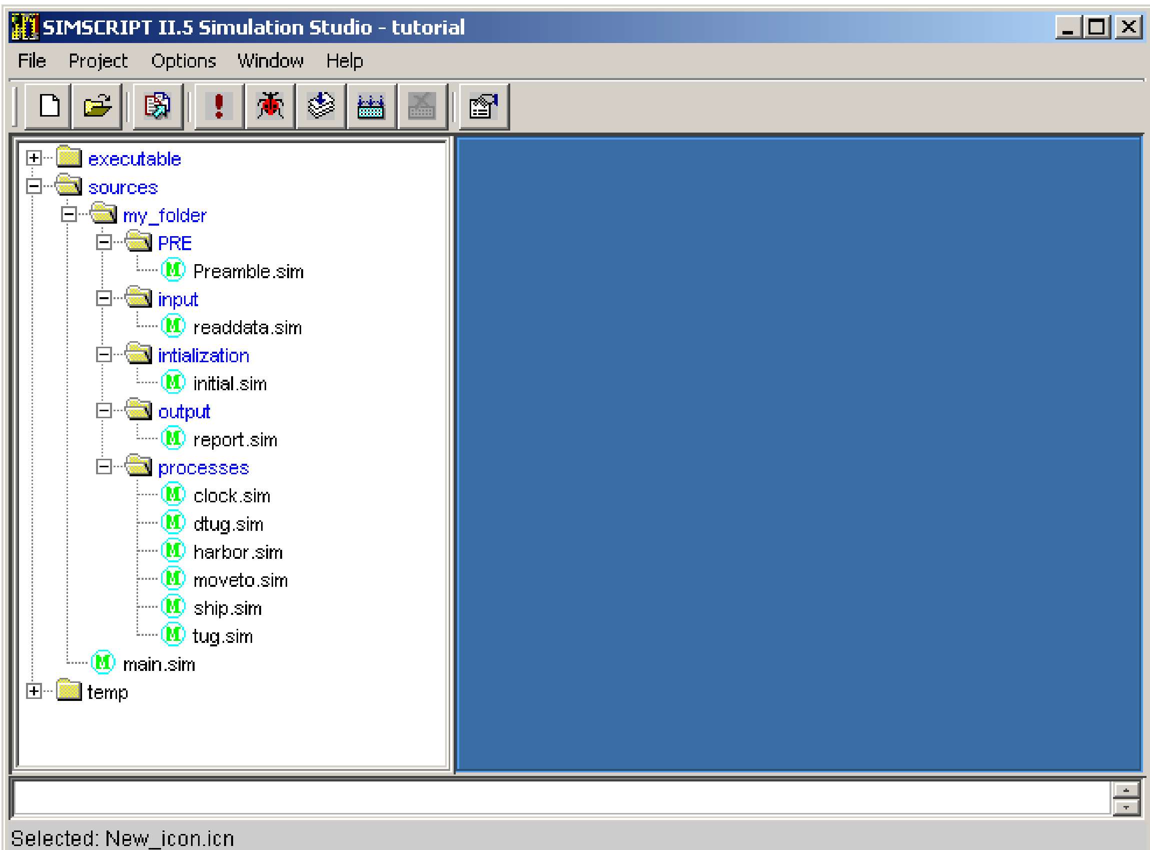


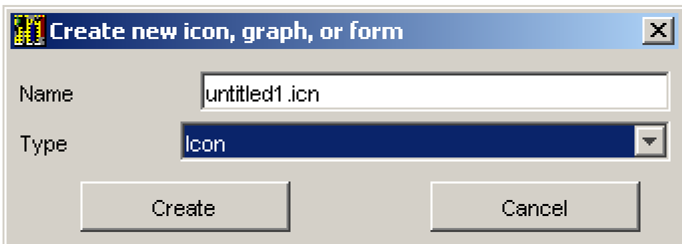
Figure 1-5 Project tree with hierarchical organization of source code

Right mouse click in the project window to bring up the pop-up menu with **update project tree** option. This can be used to add a file or hierarchical files to the project.

1.3.4 Adding Graphical Elements to a Project

Graphical elements for your model are located in the **graphics.sg2** in directory **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.



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.

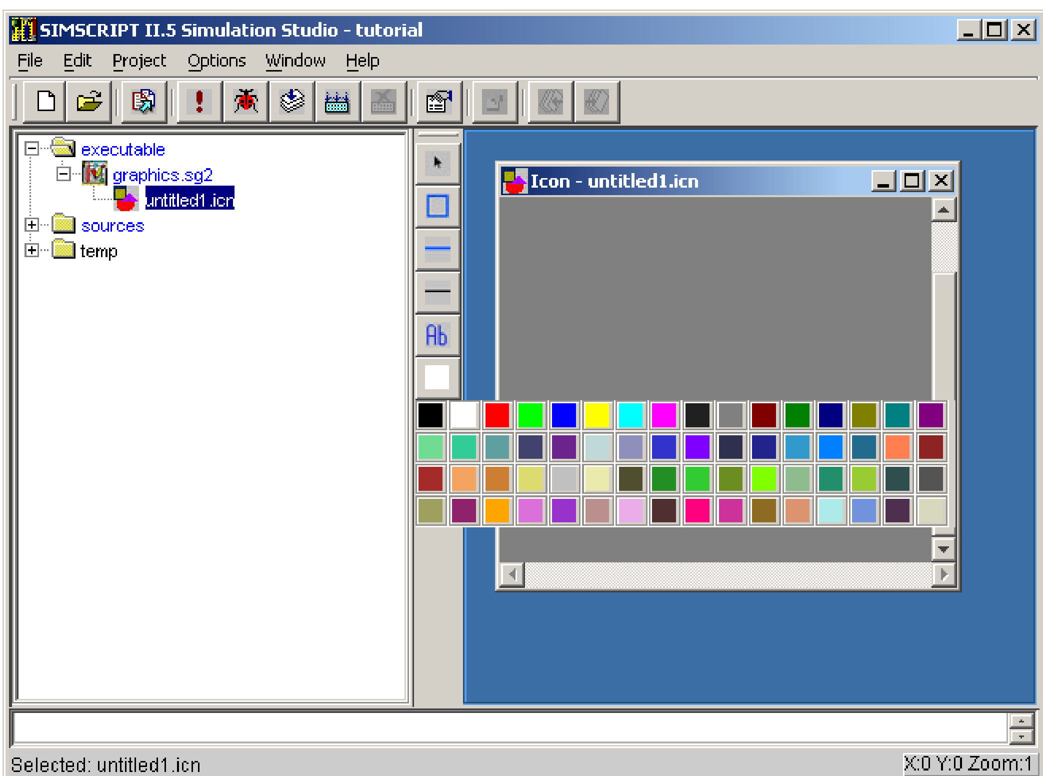
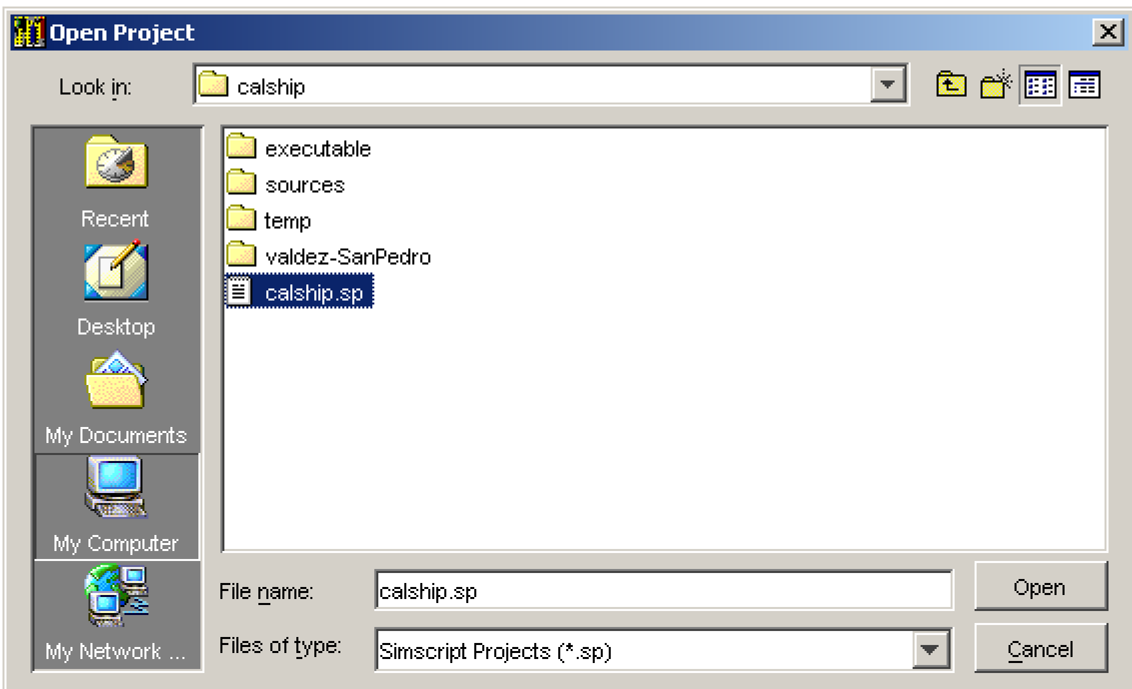


Figure 1-6 Adding a new Icon in Simstudio

A detailed explanation on how to create and use graphical elements in SIMSCRIPT II.5 models can be found in the **SIMSCRIPT II.5 Graphics User Manual**.

1.4 Opening an Existing Project

To open existing projects use the **Project->Open** menu option. The dialog box **Open Project** will appear allowing you to browse to the project directory.



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**, it will recompile all the project source files and re-link the model. When you use **Project->Build** only the modules changed after the previous build will be recompiled and the model will be re-linked.

In SIMSCRIPT II.5 when the preamble is changed, **Project->Build** will function as **Project->Rebuild All**, meaning all the source files will be recompiled and the model re-linked.

You can influence the model building procedure using menu option **Options->Project**. It will bring up a dialog box **Project options**, where you can define what you want your model to be built for **Release** or **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.

Linking phase can also be defined. Your model can be linked **With Graphics** libraries or **Without Graphics** libraries. 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. Static link is convenient when you want to link your model to be transferred to another computer for execution.

The name of the executable, by-default is **project_name,** but typing the desired name in the binary text box can change it.

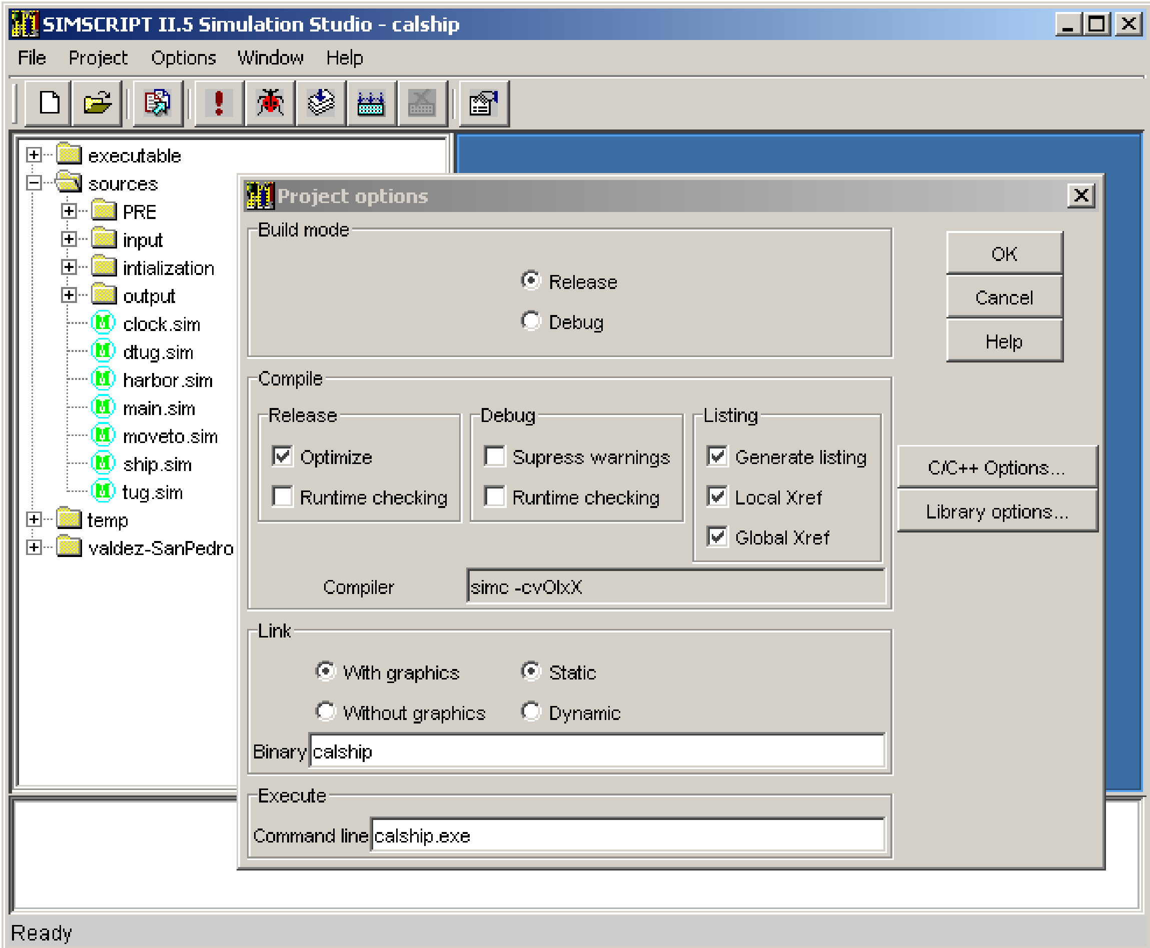


Figure 1-7 Selecting Project Options

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 you request run-time checking to be performed – this will involve entity attribute access checking and array index checking and will generate run-time error in case of incorrect access. These features will speed-up the testing phase.

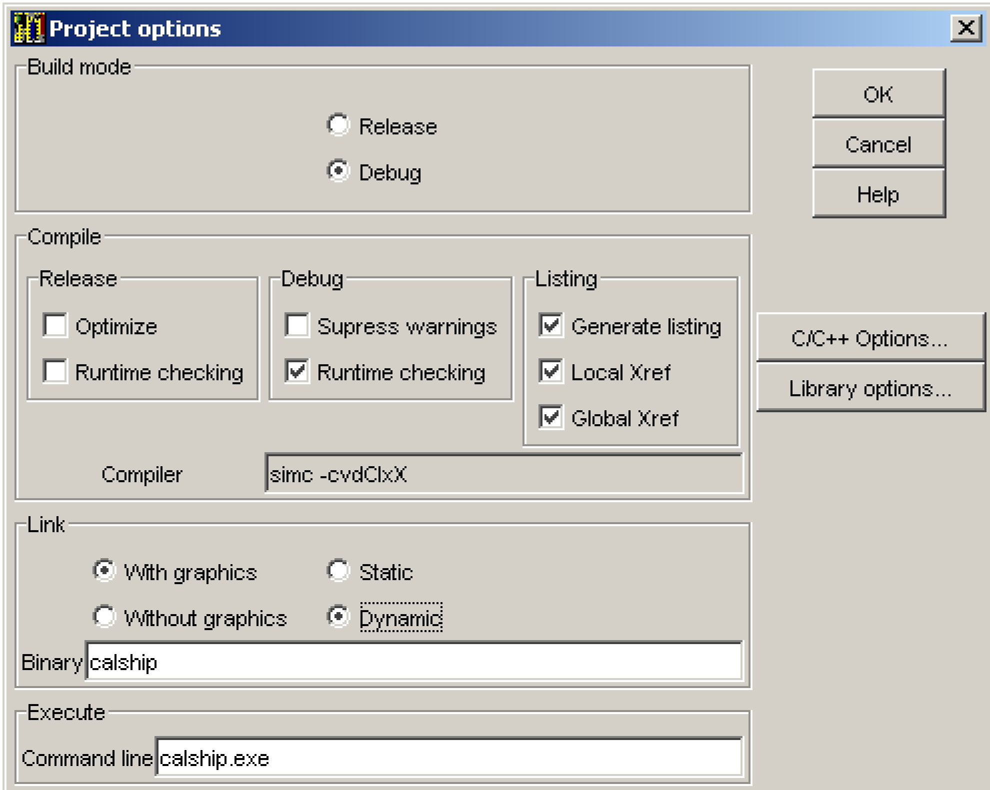


Figure 1-8 Selecting debugging options in Simstudio

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

When you finish debugging and your model is ready for the exploitation phase, you may want to choose to build your model in **Release** mode.

Use **Options->Project** to bring up the Project options dialog box and check **Build** mode **Debug.** This time choose **Optimization** to reduce model size and increase speed. If you are sure that your model is fully debugged you may exclude run-time checking. This will further increase execution speed.

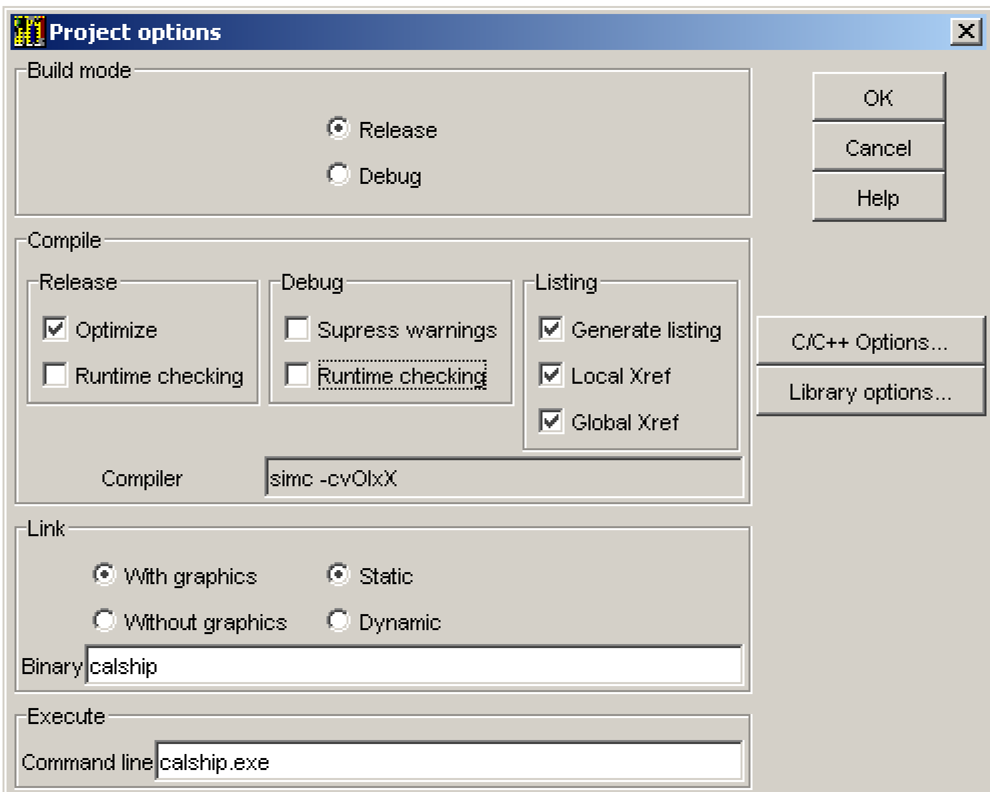


Figure 1-9 Selecting Release options in Simstudio

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** generatesone of the following: a compiler listing, a compiler listing with local cross-reference, or a full 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 Executing a Model

After building, the model executable is located in the project directory **executable**. To run it use the menu option **Project->Execute**. This is the most common way to execute a model built in Release mode.

The directory executable will also contain graphics.sg2 with graphical elements. All input data necessary for a model run should be placed in this directory.

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

1.6.1 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 write the command.

Project_name.exe – arg1 –arg2 …

Here is an example of the redirection of output of the model ed106.exe to a file ed.out.

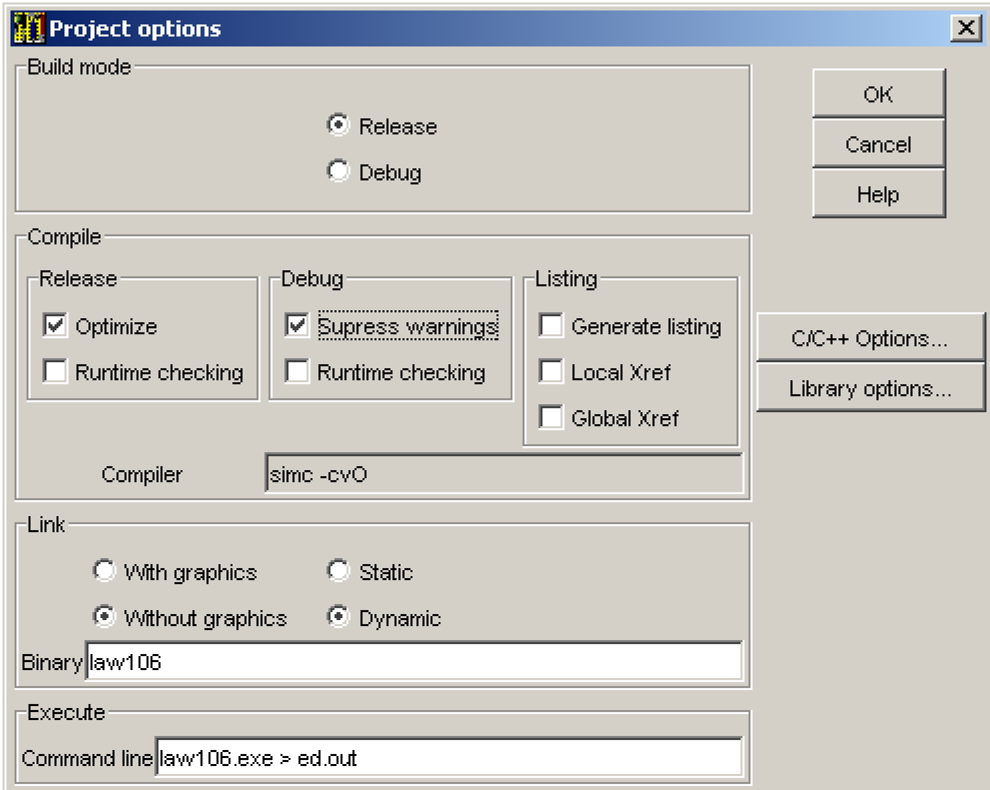


Figure 1-10 Defining command line for model execution

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.

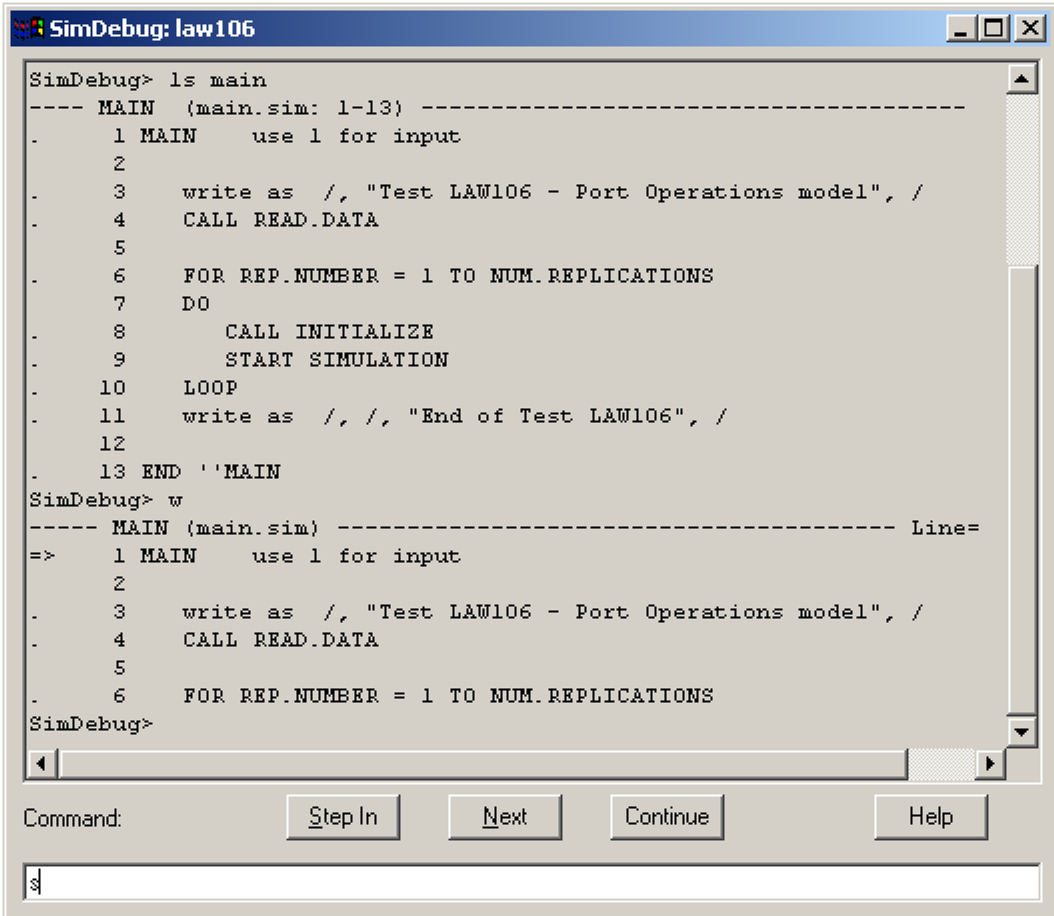


Figure 1-11 SIMSCRIPT Symbolic Debugger window

1.7 Closing the Project

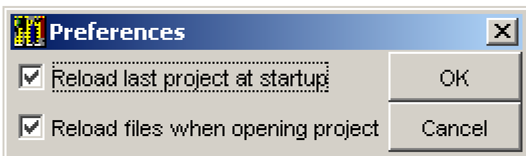
Close an open project before opening another project. To close a project use menu option **Project-> Close**.

1.8 Setting Simstudio Preferences

If you work on a project and close Simstudio without closing the project it will not remember the last project you worked on.

If you want Simstudio to open and reload the last project you worked on, when launched again, you can change its behavior.

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



Marking check box "Reload files when opening project", allows the user to request Simstudio to always update project tree when a project is opened.

1.9 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.

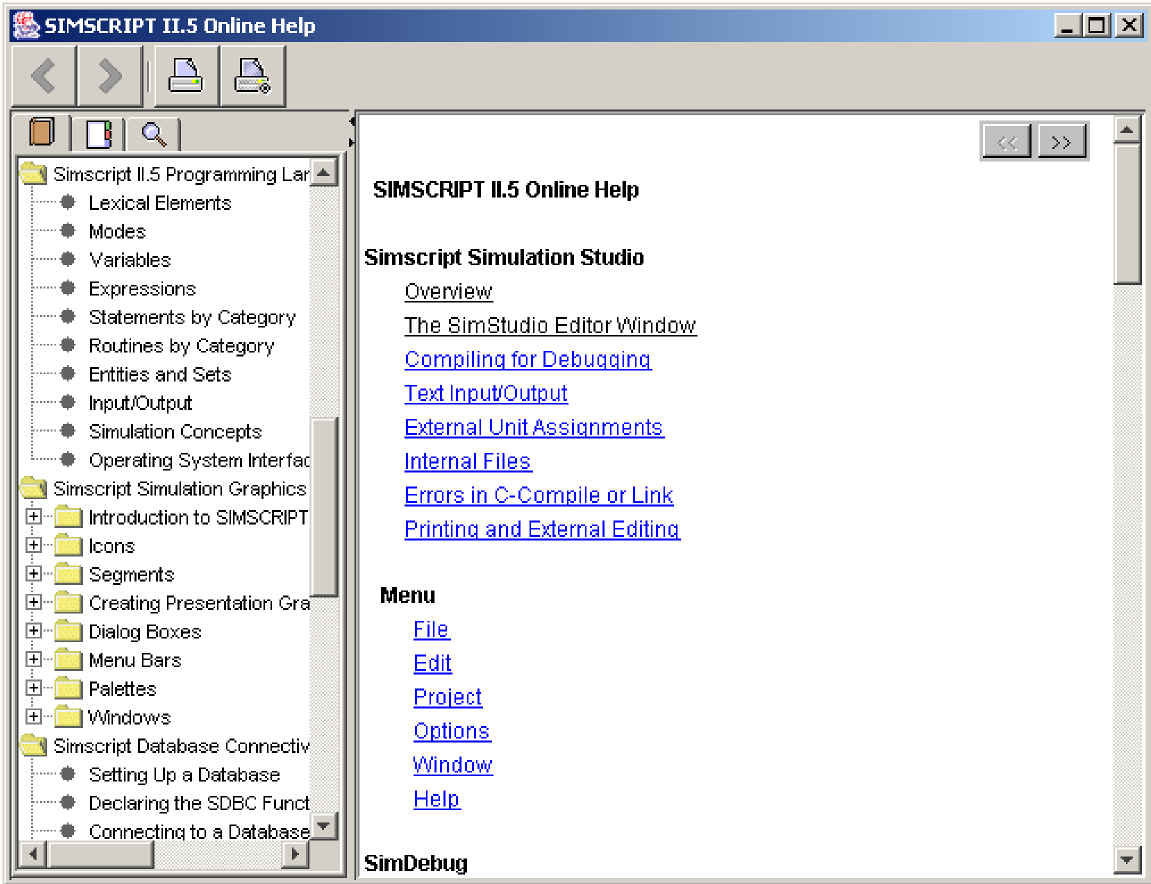


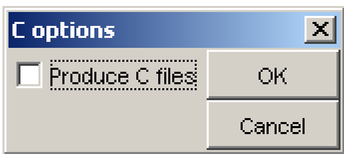
Figure 1-12 Simstudio on-line help window

1.10 Advanced Compiler/Link Options

Project options dialog box has two buttons very seldom used in more advanced model developments.

C/C++ Options…

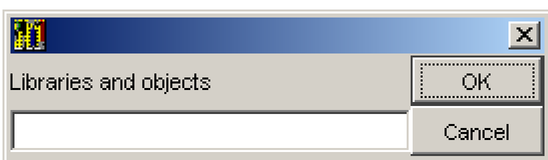
Facilitates preserving C files generated by the SIMSCRIPT compiler.



If you check this option, you will only generate C files. If you want to build the model do not check this option.

Library options …

Facilitates linking executable with objects from external additional libraries.



2 Developing Simulation Models Using Command-Line Interface

Developing a SIMSCRIPT II.5 program using command-line interface commands typically involves the following steps:

1. Preparing one or more SIMSCRIPT II.5 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 II.5 program may be prepared using **`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 store each routine in a file of its own. Files containing SIMSCRIPT II.5 source code must be given names that end with **`.sim`** or **`.SIM`**.

Although not a requirement, it is easier to compile and link a SIMSCRIPT II.5 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 II.5 compiler translates a program written in the SIMSCRIPT II.5 programming language into one or more object files. The compiler uses C as an intermediate language, but this is transparent to you, the SIMSCRIPT II.5 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 II.5 compiler and linker. Its general form is:

```
% simc [option] file.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** will be linked producing an executable file named **a.out**.

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 UNIX 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.

-l Display a routine-by-routine program listing.

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

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

-w Do not report any compiler warnings.

-x Display a local cross-reference listing for each routine.

Below is a complete list of the options available in the SIMSCRIPT II.5 compiler:

-a For each routine the compiler will produce a file containing the generated source code for the routine 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.

-b N This compiler switch is seldom used. The SIMSCRIPT II.5 compiler generates C code as an intermediate step during the compilation. This is transparent to you. On some platforms C compilers cannot compile source files with a large number of C code lines, because of the static allocation of the symbol table. By default, the SIMSCRIPT II.5 compiler will generate the intermediate C code into one file. To enforce splitting of generated intermediate C code into files with a defined (maximum) number of lines, you should invoke the SIMSCRIPT II.5 compiler with the optional compiler switch **-b N "** (break C code after N number of lines). For example:

```
simc -b 3000 big.sim
```

File **big.sim** will be transformed into **big-1.c**, **big-2.c,** etc.
Subsequently generated object modules will be **big-1.o**, **big-2.o**, etc.

-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.
Also, in the event of a runtime error, a more elaborate traceback will be provided. This option allows SIMSCRIPT II.5 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.

As of release 1.8 this option has been enhanced in the following way: When an **entity is removed from a set**, SIMSCRIPT now checks if this entity is indeed part of the given set. This is accomplished by changing the contents of the **M.setname** attribute of the entity, which not only indicates that this entity is a *member of some set*, but also indicates of *which set*.

When the list is owned by a permanent entity, the field **M.setname** now contains the *index* (integer) to the head of the list. When the list is owned by a temporary entity, **M.setname** now contains a *pointer* to the owner entity. This means that source code that checks **M.setname** for **1**, should check for `<> 0`.

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

-d Selects 'compiling for debug'. The compiler is fully integrated with the SIMSCRIPT II.5 symbolic debugger. After linking, the program can be activated with the command line switch **-debug** to provide interactive dialog with the debugger. The SIMSCRIPT II.5 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 given routine, or in an active SIMSCRIPT process instance
- Single stepping one source line at a time
- Viewing source code
- Displaying of local, global variables and 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.

-F This compiler switch is seldom used. SIMSCRIPT II.5 provides an interface toNON-SIMSCRIPT and FORTRAN routines. FORTRAN routines are invoked from SIMSCRIPT II.5 programs without appending an underscore to the FORTRAN routine name. In some computer environments this is necessary. To generate calls with the appended underscore, the SIMSCRIPT II.5 compiler should be invoked with the optional compiler switch **-F.** For example:

```
simc -F prog.sim
```

-g The compiler will provide a detailed traceback listing without enabling runtime checking. Routines compiled with **-g** will be shown with the 'current line number' and all their local variables in a traceback.

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

-l The compiler will write a listing to 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).

-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 The **C** compiler's optimizer will be involved when compiling. This option will increase compile time, but will reduce model runtime. On very rare occasions, some optimizers may produce incorrect code, resulting in incorrect behavior of your program. If this is suspected, try compiling without optimization. The following command will create an optimized executable called **prog** after compiling **filename.sim**.

simc -O -o prog filename.sim

-p Compile using profiling code. See **prof(1)** and **cc(1)** in the man pages for details. This must be specified at link-time, either through **simc** or **simld.** See paragraph 2.6. This option may not be provided on all computer platforms.

-S Create only a **.c** file. Do not produce **.o** or link.

-temp=dir

Specify the location of compiler temporary files. The default is **/tmp**. This does not affect where the **C** compiler places its own temporary files.

-v This option means a VERY OLD PREAMBLE. It is used during re-compilation of some 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

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

-x The compiler will write to the listing a local cross-reference for each routine. A local cross-reference shows the line number of every reference made to each name in the routine.

-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, which references each globally defined name.

-1 The compiler will not generate code. It is sometimes desirable to quickly check the syntax of a program and/or produce a listing without generating any object files. Note, this is a "one" not an "ell".

The following command compiles a program consisting of 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 compiler expects to find the preamble of the program at the beginning of the first file specified. Thus, if the program in the above examples contains a preamble, it must be located at the beginning of file **abc.sim** or compilation errors will result.

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. Since the compiler expects to find the PREAMBLE in the first file it encounters, it is necessary that the file containing the PREAMBLE be given a name, which precedes all others in sorted order. Since upper-case names precede lower-case names, one convention, which may be followed, is to store the PREAMBLE in a file named **PREAMBLE.sim** and to name the rest of the files using all lower-case characters.

2.3 Recompiling

Whenever a change is made to the PREAMBLE of a program, it is necessary to re-compile the entire program. If a change is made only to routines of the program, only those routines that have been modified 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**.

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**. The **simld** command is used to link a SIMSCRIPT II.5 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. 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 SIMGRAPHICS programs. For example, to link a SIMGRAPHICS II program and name the executable file **animate**, enter

```
% simgld -o animate *.o
```

It is possible to create a library of SIMSCRIPT II.5 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 II.5 is installed. For more details of the **SIMHOME**, see the Installation Notes for the current

SIMSCRIPT II.5 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 II.5 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 II.5 compiler.

SIMSCRIPT II.5 supports two graphics systems SIMGRAPHICS I and SIMGRAPHICS II. As of Release 1.9, SIMGRAPHICS II is the default SIMGRAPHICS in SIMSCRIPT II.5 systems. Compiler switch **-G** will link graphical models with SIMGRAPHICS II libraries.

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

Also **simgld** will automatically link with SIMGRAPHICS II libraries. If you want to use SIMGRAPHICS I, you must compile your model with the **-c** option and use **simgld1** as follows:

```
simc -c *.sim
simgld1 *.o
```

SIMSCRIPT II.5 runtime libraries as well as SIMGRAPHICS 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, simgld1** and **simgld2** 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 II.5, or does not have SIMSCRIPT II.5 installed at all, your model must be linked statically. This means that you have to perform static link or "total link". In other words, your executable has to include all object modules in itself.
SIMSCRIPT II.5 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 default with SIMGRAPHICS II
`tsimgld1`	- static link of graphical models with SIMGRAPHICS I
`tsimgld2`	- static link of graphical models with SIMGRAPHICS II

2.5 Executing

A SIMSCRIPT II.5 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 II.5 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 II.5 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 bespecified 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 II.5, 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** and **-g** will provide routine names with local variables and line numbers. If none of these switches are used, only routine names will be written, without other debugging information.

In the event that a runtime error goes undetected by SIMSCRIPT II.5 and a program aborts with a core dump, it is possible to analyze the core file using the UNIX debugger, **adb**.

Any SIMSCRIPT II.5 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 Profiling

Profiling is useful when analyzing the performance of a program. Profiling helps determine where most of the execution time in a program is spent. In the typical program, execution time is confined to a relatively few sections of code. It may be profitable to concentrate on improving coding efficiency in only those sections.

Profiling is platform specific, and may not be available on all UNIX platforms. We will describe a common approach, using the **prof** command for profiling a SIMSCRIPT II.5 model.

The **prof** command produces an execution profile of a program. The profile data is taken from the profile file, which is created by programs compiled with the **-p** option. That option also links in versions of the library routines, which are compiled for profiling.

When a program is profiled, the results appear in a file called **mon.out** (default filename) at the end of the run. Every time the program is run, a new **mon.out** file is created overwriting the old version. The profiled program must exit or return normally for the profiling information to be saved in the **mon.out** file. The **prof** command is then used to interpret the results of the profile.

prof Displays the following information for each routine:

%time Percentage of the total time of the program, that was consumed by this routine.

cumsecs A running sum of the number of seconds accounted for by this function and those listed above it.

#call The number of times this routine was called.

ms/call How many milliseconds this routine consumed each time it was called.

name The name of the routine.

To obtain a profile of a SIMSCRIPT II.5 program, it is necessary to link the program using the **-p** option. To tally the number of calls to a routine, the file that contains the routine must be compiled with the **-p** option.

Compile the modules you want profiled with the **-p** flag:

```
% simc -c -p file1.sim file2.sim
% simc -c file3.sim
```

To link the program, type:

```
% simld -p file1.o file2.o file3.o
```

Or simply:

```
% simc -p file1.o file2.o file3.o
```

Run your program:

```
% a.out
```

Now use **prof** to write an execution profile to standard output:

```
% prof a.out
```

The following is some sample profile data created by **prof**. Routines that begin with **_H** are SIMSCRIPT library routines. Routines that begin with **_R** were generated by the SIMSCRIPT compiler or are user routines. Routines that begin with **_Q** are SIMSCRIPT or user left routines. Other routines are C library routines.

Note: The symbol **`mcount`** is a side effect of profiling, and indicates the overhead incurred by profiling.

%time	cumsecs	#call	ms/call	name
21.4	25.66			_HP_SUSPEND_R
18.6	47.91			_HP_RESUME_R
11.9	62.12			mcount
6.9	70.34	220716	0.04	_HTIM0_R
3.9	75.01	11755	0.40	_RJOB
3.5	79.16	165643	0.03	_HT_EV_S
2.3	81.93	110445	0.03	_HRANDOM_F
2.2	84.62	110419	0.02	_QS_N_X_TRANSPORTER
2.0	87.00	208985	0.01	_HPRQ_R
1.7	89.08	86922	0.02	_QS_N_X_WORK_STATION
1.6	91.00 1	30610	0.01	_calloc
1.5	92.84	86922	0.02	_QS_WS_NUM_MACH_WORKING
1.4	94.51	56318	0.03	_QS_N_Q_WORK_STATION
1.3	96.11	208959	0.01	_HPSU_R
1.3	97.68	98664	0.02	_HRNQ_R
1.2	99.12 5	5303	0.03	_log
1.2	100.52	28165	0.05	_RT_Q_WORK_STATION
1.1	101.84	1	1320.00	_HTIME_R
1.1	103.12	98689	0.01	_HREQ_R
0.9	104.22	220716	0.00	_HPCALL_R
0.9	105.28	208985	0.01	_HPSUSP_R
0.8	106.29	220716	0.00	_HTIM1_R
0.8	107.21	130716	0.01	_malloc
0.7	108.08	241264	0.00	.mul
0.7	108.90	429904	0.00	_HDIM_F
0.7	109.71	55210	0.01	_RT_X_TRANSPORTER
0.6	110.46	43467	0.02	_QS_WS_DELAY_IN_QUEUE
0.6	111.19	43467	0.02	_HERLANG_F
0.6	111.86	55209	0.01	_RZ_X_TRANSPORTER

See the man page for **`prof(1)`** for more information.

2.7 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.

2.7.1 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.

2.7.2 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.

2.7.3 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).

2.7.4 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.

2.7.5 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
```

2.8 Obtaining Online Help

Online documentation regarding the use of the SIMSCRIPT II.5 compiler can be obtained by using the **simhelp** command, e.g.

```
% simhelp simc
```

Simhelp by itself lists all topics for which help is available.

2.9 Example Program

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

```
      m 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

  PREAMBLE

     RESOURCES INCLUDE CPU AND MEMORY
     PROCESSES INCLUDE GENERATOR AND STOP.SIM
        EVERY JOB HAS A JB.PRIORITY
                   AND A JB.MEMORY.REQUIREMENT
        DEFINE JB.PRIORITY AND JB.MEMORY.REQUIREMENT
                   AS INTEGER VARIABLES
        DEFINE JOB.DELAY.TIME AS A REAL VARIABLE
     EXTERNAL PROCESS IS JOB
     EXTERNAL PROCESS UNIT IS 1
     DEFINE SMALL.JOB.INTERARRIVAL.TIME,
       MEAN.SMALL.JOB.PROCESSING.TIME, RUN.LENGTH
           AND STOP.TIME AS REAL VARIABLES
     DEFINE NO.CPU AND MAX.MEMORY AS INTEGER VARIABLES
     DEFINE MAX.MEMORY.QUEUE TO MEAN 1MAX.MEMORY.QUEUE

     ACCUMULATE CPU.UTILIZATION AS THE AVG OF N.X.CPU
     ACCUMULATE MEMORY.UTILIZATION AS THE AVERAGE
        OF N.X.MEMORY
     ACCUMULATE AVG.CPU.QUEUE AS THE AVG AND
        MAX.CPU.QUEUE AS THE MAXIMUM OF N.Q.CPU
     ACCUMULATE AVG.MEMORY.QUEUE AS THE AVG
        AND MAX.MEMORY.QUEUE AS THE MAXIMUM OF N.Q.MEMORY
   TALLY AVG.JOB.TIME AS THE AVERAGE AND NO.JOBS.PROCESSED
AS
           THE NUMBER OF JOB.DELAY.TIME

     DEFINE HOURS TO MEAN UNITS

   END ''PREAMBLE
```

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

PROCESS GENERATOR

         UNTIL TIME.V >= STOP.TIME
         DO
           ACTIVATE A JOB NOW
           LET JB.PRIORITY.. = RANDI.F(1,10,1)
         LET JB.MEMORY.REQUIREMENT.. = RANDI.F(1,MAX.MEMORY,2)
         WAIT EXPONENTIAL.F(SMALL.JOB.INTERARRIVAL.TIME,3) MINUTES
         LOOP

END
```

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

PROCESS JOB

   DEFINE ARRIVAL.TIME AND PROCESSING.TIME
     AS REAL VARIABLES
   IF PROCESS IS EXTERNAL
     READ JB.PRIORITY..,JB.MEMORY.REQUIREMENT.. AND
         PROCESSING.TIME
   ELSE
     LET PROCESSING.TIME = MIN.F(EXPONENTIAL.F
         (MEAN.SMALL.JOB.PROCESSING.TIME,4),2 *
         MEAN.SMALL.JOB.PROCESSING.TIME)
   ALWAYS
   LET ARRIVAL.TIME = TIME.V
   REQUEST JB.MEMORY.REQUIREMENT.. UNITS OF MEMORY(1)
      WITH PRIORITY JB.PRIORITY..
   REQUEST 1 CPU(1) WITH PRIORITY JB.PRIORITY..
   WORK PROCESSING.TIME MINUTES
   RELINQUISH JB.MEMORY.REQUIREMENT.. UNITS OF MEMORY(1)
   RELINQUISH 1 CPU(1)
   LET JOB.DELAY.TIME = TIME.V - ARRIVAL.TIME

END
```

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

MAIN

   WRITE AS /, "A COMPUTER CENTER STUDY", /, /

   Open unit 1 for input

   LET HOURS.V = 1
   CREATE EVERY CPU(1) AND MEMORY(1)
   Let U.CPU(1) = 1
   Let U.MEMORY(1) = 6
   LET NO.CPU = U.CPU(1)
   LET MAX.MEMORY = U.MEMORY(1)

   Let SMALL.JOB.INTERARRIVAL.TIME = 2.0
   Let MEAN.SMALL.JOB.PROCESSING.TIME = 0.8
   Let RUN.LENGTH = 12.0
```

```
      LET STOP.TIME = RUN.LENGTH / HOURS.V

      PRINT 6 LINES WITH U.CPU(1), U.MEMORY(1),
         60/SMALL.JOB.INTERARRIVAL.TIME,
         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

      ACTIVATE A GENERATOR NOW
      ACTIVATE A STOP.SIM IN STOP.TIME HOURS
      START SIMULATION

   END ''MAIN
                                                         5
CACI SIMSCRIPT II.5 (R) v2.0                  6/26/1997 15:23:42

  PROCESS STOP.SIM

      SKIP 6 LINES
    PRINT 9 LINES WITH TIME.V, CPU.UTILIZATION(1)*100/NO.CPU,
           MEMORY.UTILIZATION(1)*100/MAX.MEMORY,
           AVG.MEMORY.QUEUE(1), MAX.MEMORY.QUEUE(1),
           AVG.CPU.QUEUE(1), MAX.CPU.QUEUE(1),
           NO.JOBS.PROCESSED AND AVG.JOB.TIME * MINUTES.V
           THUS
A F T E R **.** HOURS
THE CPU UTILIZATION WAS           *.** %
THE MEMORY UTILIZATION WAS        *.** %
THE AVG QUEUE FOR MEMORY WAS      *.** JOBS
THE MAX QUEUE FOR MEMORY WAS      *.** JOBS
THE AVG QUEUE FOR A CPU WAS  *.** JOBS
THE MAX QUEUE FOR A CPU WAS  *.** JOBS
THE TOTAL NUMBER OF JOBS COMPLETED WAS  ***
WITH AN AVERAGE PROCESSING TIME OF *.*** MINUTES

   STOP

   END
```

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 |
   scratch |
   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 and Packing Codes

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)
`Integer`	A signed integer of at least 32 bits
`Real`	A floating-point number of at least 32 bits
`Double`	A floating-point number of at least 64 bits
`Pointer`	An address
`Subprogram`	An address of a routine
`Text`	An address of a character string

Bit packing is supported. For example, on 32 bit machines, any packing code (a-b) is allowed provided that:

$$1 \le a \le b \le 32$$

Examples: (1-4), (12-12), (21-22)

The following shows each of the available field-packing codes together with its equivalent bit-packing code:

(1/2)	(1-16)
(2/2)	(17-32)
(1/4)	(1-8)
(2/4)	(9-16)
(3/4)	(17-24)
(4/4)	(25-32)

Intrapacking codes, (*/2) and (*/4), are also supported.

3.2.1 Alignment of Values

Some machines require strict alignment of double-precision floating point values on a double word boundary. For maximum portability to these systems, variables and permanent attributes of mode **`double`** should be assigned to odd array numbers. Similarly, **`double`** temporary attributes should be assigned to odd word numbers or left for automatic definition.

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:

```
sub(inarg,outarg)
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
```

When calling this subroutine, the first argument should evaluate to `integer` since this is the SIMSCRIPT II.5 mode, which corresponds to the C type **long**. The second argument must be a pointer to an **integer**. This can be accomplished by passing a pointer to an `integer`array. For example:

```
define IN.ARG as an integer variable
define OUT.ARG as a 1-dim integer array
write as "Enter the input value:", /
read IN.ARG
reserve OUT.ARG(*) as 1
call sub(IN.ARG, OUT.ARG(*))
write OUT.ARG(1) as "The output value is ", I 10, /
```

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

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

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

```
define FUNC as an integer nonsimscript function
```

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 II.5 mode of each argument and function matches its C type. Here is a list of C types and the corresponding SIMSCRIPT II.5 modes:

unsigned char	`alpha`
unsigned short	`integer2`
shortsigned	`integer2`
long	`integer`
float	`real`
double	`double`

If an argument is a pointer to a null-terminated character string, pass a `text` value.

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 II.5 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 II.5 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. Sincethe 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 options are **`-g`** for traceback, and **`-d`** for full debug.

To be able to look at entities, system variables and global variables you must compile the

`PREAMBLE` with debugging or traceback, i.e. with the **`-d`** or **`-g`** 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 II.5 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!

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 **`-g`** and 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.

`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.

4.4.8 Unsupported SIMSCRIPT Features

SimDebug Release 1.0 supports all SIMSCRIPT features, with the exception of **packed permanent entities.** However, packed *temporary* entities are supported.

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 unsubscripted `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 unsubscripted 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
`Histogram`s 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 doubleprecision floating point numbers be aligned on a double-word boundary. This requires that unsubscripted `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 II.5 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 II.5'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 deallocated 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 imlicit 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

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.

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

Appendix C Standard SIMSCRIPT II.5 Names

C.1 Functions and Routines

Function `abs.f (arg)`

Arguments:

`arg` An **`integer`** or **`double`** value

Description: Returns the absolute value of **`arg`**.

Mode: The mode of **`arg`**.

Function `and.f (arg1, arg2)`

Arguments:

`arg1` An **`integer`** value.
`arg2` An **`integer`** value.

Description: Returns the logical product of **`arg1`** and **`arg2`**.

Mode: **`Integer`**

Function `arccos.f (arg)`

Arguments:

`arg` A **`double`** value between `-1` and `+1`.

Description: Returns the arc cosine of **`arg`**.

Mode: **`Double`**

Function `arcsin.f (arg)`

Arguments:

`arg` A **`double`** value between `-1` and `+1`.

Description: Returns the arc sine of **`arg`**.

Mode: **`Double`**

Function `arctan.f (arg1, arg2)`

Arguments:

`arg1` A **`double`** value
`arg2` A **`double`** value

Description: Returns the arc tangent of **`arg1/arg2`**.

Mode: **Double**

Function atot.f (arg)

Arguments:
arg An **alpha** value.

Description: Returns a **text** value of length 1 containing **arg.**

Mode: **Text**

Function beta.f (k1, k2, stream)

Arguments:
k1 A **double** value greater than zero specifying the power of X.
k2 A **double** value greater than zero specifying the power of (1-X).
stream An **integer** value specifying the random number stream.

Description: Returns a random sample from a beta distribution.

Mode: **Double**

Function binomial.f (n, p, stream)

Arguments:
n An **integer** value specifying the number of trials.
P A **double** value specifying the probability of success.
stream An **integer** value specifying the random number stream.

Description: Returns a random sample from a binomial distribution.

Mode: **Integer**

Function concat.f (arg1, arg2, ...)

Arguments:
arg1,
arg2, ... Two or more **text** values.

Description: Returns the concatenation of **arg1, arg2, ...**
Mode: **Text**
Function cos.f (arg)

Arguments:
arg A **double** value specifying an angle in radians.

Description: Returns the cosine of **arg.**

Mode: **Double**

Function date.f (month, day, year)

Arguments:

`month`	An **`integer`** value specifying the month.
`day`	An **`integer`** value specifying the day within the month.
`year`	An **`integer`** value specifying the year.

Description: Returns the cumulative simulation time for the given calendar date based on values given to **`origin.r.`**

Mode: **`Integer`**

Routine `date.r yielding date, time`

Arguments:

`date`	A **`text`** value containing the current date in the form **`MM/DD/YYYY.`**
`time`	A **`text`** value containing the current time in the form **`HH:MM:SS.`**

Description: Returns the current date and time.

Function `day.f (time)`

Arguments:

`time`	A **`double`** value specifying a cumulative simulation time.

Description: Returns the day portion corresponding to the simulation time based on values given to **`origin.r.`**

Mode: **`Integer`**

Function `descr.f (string)`

Arguments:

`string`	A text value, **`text`** variable or expression.

Description: Indicates an argument to a NONSIMSCRIPT routine is passed by descriptor. Used for VMS, ignored by UNIX systems.

Mode: n.a.

Function `dim.f (array(*))`

Arguments:

`array(*)`	An array pointer.

Description: Returns the number of elements in the array.

Mode: **`Integer`**

Function `div.f (arg1, arg2)`

Arguments:

`arg1`	An **`integer`** value.
`arg2`	An **`integer`** value not equal to zero.

Description: Returns the truncated value of **arg1/arg2.**

Mode: **Integer**

Function efield.f

Arguments: None

Description: Returns the ending column of the next data field to be read by a free-form **read** statement. Returns zero if there are no more data fields.

Mode: **Integer**

Function erlang.f (mu, k, stream)

Arguments:
- **mu** A **double** value greater than zero specifying the mean.
- **k** An **integer** value greater than zero specifying the number of stages.
- **stream** An **integer** value specifying the random number stream.

Description: Returns a random sample from an Erlang distribution.

Mode: **Double**

Routine exit.r (status)

Arguments:
- **status** An **integer** value specifying an exit status.

Description: Terminates program execution passing the exit status to the command level.

Function exp.f (arg)

Arguments:
- **arg** A **double** value.

Description: Returns "e to the **arg**".

Mode: **Double**

Function exponential.f (mu, stream)

Arguments:
- **mu** A **double** value greater than zero specifying the mean.
- **stream** An **integer** value specifying the random number stream.

Description: Returns a random sample from an exponential distribution.

Mode: **Double**

Function `fixed.f (txt, len)`

Arguments:

`txt`	A **`text`** value.
`len`	A non-negative **`integer`** value.

Description: Returns a copy of **`txt`**, which is either space-padded or truncated so that its length is **`len`**.

Mode: **`Text`**

Function `frac.f (arg)`

Arguments:

`arg`	A **`double`** value.

Description: Returns the fractional part of **`arg`**.

Mode: **`Double`**

Function `gamma.f (mu, k, stream)`

Arguments:

`mu`	A **`double`** value greater than zero specifying the mean.
`k`	A **`double`** value greater than zero specifying the shape.
`stream`	An **`integer`** value specifying the random number stream.

Description: Returns a random sample from a gamma distribution.

Mode: **`Double`**

Function `hour.f (time)`

Arguments:

`time`	A **`double`** value specifying a cumulative event time.

Description: Returns the hour portion corresponding to the event time.

Mode: **`Integer`**

Function `int.f (arg)`

Arguments:

`arg`	A **`double`** value.

Description: Returns **`arg`** rounded to the nearest integer.

Mode: **`Integer`**

Function `itoa.f (arg)`

Arguments:
arg An **integer** value in the range 0 to 9.

Description: Returns an **alpha** value containing the ASCII representation of the given digit.

Mode: **alpha**

Function itot.f (arg)

Arguments:
arg An **integer** value.

Description: Returns a **text** value containing the ASCII representation of the given value.

Mode: **Text**

Function length.f (arg)

Arguments:
arg A **text** value.

Description: Returns the number of characters in **arg.**

Mode: **Integer**

Function log.e.f (arg)

Arguments:
arg A **double** value greater than zero.

Description: Returns the natural logarithm of **arg.**

Mode: **Double**

Function log.normal.f (mu, sigma, stream)

Arguments:
mu A **double** value greater than zero specifying the mean.
sigma A **double** value greater than zero specifying the standard deviation.
stream An **integer** value specifying the random number stream.

Description: Returns a random sample from a log normal distribution.

Mode: **Double**

Function log.10.f (arg)

Arguments:
arg A **double** value greater than zero.

Description: Returns the base 10 logarithm of **`arg.`**

Mode: **`Double`**

Function `lower.f (arg)`

Arguments:
`arg` A **`text`** value.

Description: Returns a copy of **`arg`** with each upper-case character converted to lowercase.

Mode: **`Text`**

Function `match.f (string, pattern, offset)`

Arguments:
`string` A **`text`** value.
`pattern` A **`text`** value.
`offset` A non-negative **`integer`** value.

Description: Returns the position within **`string`** of the first occurrence of **`pattern`**, or zero if there is no such occurrence. The search begins after skipping the first **`offset`** characters of **`string.`**

Mode: **`Integer`**

Function `max.f (arg1, arg2, ...)`

Arguments:
`arg1,`
`arg2, ...` Any combination of two or more **`integer`** or **`double`** values.

Description: Returns the maximum of **`arg1, arg2, ....`**

Mode: **`Integer`** if each of the arguments is **`integer`**. Otherwise, **`double.`**

Function `min.f (arg1, arg2, ...)`

Arguments:
`arg1,`
`arg2, ...` Any combination of two or more **`integer`** or **`double`** values.

Description: Returns the minimum of **`arg1, arg2, ....`**

Mode: **`Integer`** if each of the arguments is **`integer`**. Otherwise, **`double.`**

Function `minute.f (time)`

Arguments:
`time` A **`double`** value specifying a cumulative event time.

Description: Returns the minute portion corresponding to the event time.

Mode: **Integer**

Function mod.f (arg1, arg2)

Arguments:

arg1 An **integer** or **double** value.
arg2 An **integer** or **double** value not equal to zero.

Description: Returns a remainder computed as:

```
arg1 - (trunc.f(arg1/arg2) * arg2)
```

Mode: **Integer** if both arguments are **integer**. Otherwise, **double.**

Function month.f (time)

Arguments:

time A **double** value specifying a cumulative simulation time.

Description: Returns the month portion corresponding to the simulation time based on values given to **origin.r.**

Mode: **Integer**

Function nday.f (time)

Arguments:

time A **double** value specifying a cumulative event time.

Description: Returns the day portion corresponding to the event time.

Mode: **Integer**

Function normal.f (mu, sigma, stream)

Arguments:

mu A **double** value specifying the mean.
sigma A **double** value greater than zero specifying the standard deviation.
stream An **integer** value specifying the random number stream.

Description: Returns a random sample from a normal distribution.

Mode: **Double**

Function or.f (arg1, arg2)

Arguments:

arg1 An **integer** value.
arg2 An **integer** value.

Description: Returns the logical sum of **arg1** and **arg2.**

Mode: **Integer**

Routine origin.r (month, day, year)

Arguments:

month	An **integer** value specifying the month.
day	An **integer** value specifying the day within the month.
year	An **integer** value specifying the year.

Description: Defines the calendar date of the start of simulation.

Right function out.f (column)

Arguments:

column	An **integer** value specifying a column position.

Description: Returns the character in the specified column of the current record of the current output unit.

Mode: **Alpha**

Left function out.f (column)

Arguments:

column	An **integer** value specifying a column position.

Enter with: An **alpha** value.
Description: Stores the assigned character in the specified column of the current record of the current output unit.

Function poisson.f (mu, stream)

Arguments:

mu	A **double** value greater than zero specifying the mean.
stream	An **integer** value specifying the random number stream.

Description: Returns a random sample from a Poisson distribution.

Mode: **Integer**

Function randi.f (low, high, stream)

Arguments:

low	An **integer** value specifying the beginning value.
high	An **integer** value specifying the ending value.
stream	An **integer** value specifying the random number stream.

Description: Returns a random sample uniformly distributed between **low** and **high** inclusive.

Mode: **Integer**

Function random.f (stream)

Arguments:

stream An **integer** value specifying the random number stream.

Description: Returns a pseudo-random number between 0 and 1.

Mode: **Double**

Function real.f (arg)

Arguments:

arg An **integer** value.

Description: Returns **arg** as a **double** value.

Mode: **Double**

Function ref.f (any)

Arguments:

Description: Indicates an argument to a NONSIMSCRIPT routine is passed by reference.

Mode: n.a

Function repeat.f (txt, count)

Arguments:

txt A **text** value.

count A non-negative **integer** value.

Description: Returns a **text** value which is the concatenation of **count** copies of **txt**.

Mode: **Text**

Function sfield.f

Arguments: None

Description: Returns the starting column of the next data field to be read by a free-form **read** statement. Returns zero if there are no more data fields.

Mode: **Integer**

Function shl.f (arg1, arg2)

Arguments:
arg1 An **integer** value.
arg2 An **integer** value.

Description: Returns the value of **arg1** shifted left **arg2** bit positions.

Mode: **Integer**

Function shr.f (arg1, arg2)
Arguments:
arg1 An **integer** value.
arg2 An **integer** value.

Description: Returns the value of **arg1** shifted right **arg2** bit positions.

Mode: **Integer**

Function sign.f (arg)

Arguments:
arg A **double** value.

Description: Returns **+1** if **arg** is positive, **-1** if **arg** is negative, and **0** if **arg** is zero.

Mode: **Integer**

Function sin.f (arg)

Arguments:
arg A **double** value specifying an angle in radians.

Description: Returns the sine of **arg.**

Mode: **Double**

Routine sleep.r (time)

Arguments:

time A **double** value specifying time in seconds.

Description: Suspends execution of your program for a specified time period. Implemented on VMS platforms only.

Routine snap.r

Arguments: None

Description: User-supplied snapshot routine that is called when a runtime error is detected.

Function `sqrt.f (arg)`

Arguments:
`arg` A non-negative **`double`** value.

Description: Returns the square root of **`arg.`**

Mode: **`Double`**

Right function `substr.f (txt, pos, len)`

Arguments:
`txt` A **`text`** value.
`pos` An **`integer`** value greater than zero.
`len` A non-negative **`integer`** value.

Description: Returns the substring of **`txt`** of length **`len`** starting at position **`pos.`**

Mode: **`Text`**

Left function `substr.f (txt, pos, len)`

Arguments:
`txt` A **`text`** value.
`pos` An **`integer`** value greater than zero.
`LEN` A non-negative **`integer`** value.

Enter with: A **`text`** value.

Description: Replaces the substring of **`txt`** of length **`len`** starting at position **`pos`** with the assigned **`text`** value.

Routine `system.r (command, status)`

Arguments:
`command` A **`text`** value specifying command string.
`status` An **`integer`** value specifying VMS return status.

Description: Implemented on VMS platforms only. Executes VMS DCL command.

Function `tan.f (arg)`

Arguments:
`arg` A **`double`** value specifying an angle in radians.

Description: Returns the tangent of **`arg.`**

Mode: **Double**

Function triang.f (min, mu, max, stream)

Arguments:

min	A **double** value specifying the minimum.
mu	A **double** value specifying the mean.
max	A **double** value specifying the maximum.
stream	An **integer** value specifying the random number stream.

Description: Returns a random sample from a triangular distribution.

Mode: **Double**

Function trim.f (txt, flag)

Arguments:

txt	A **text** value.
flag	An **integer** value.

Description: Returns a copy of **txt** which has leading and/or trailing blanks removed. If **flag** $\leq$ 0, leading blanks are removed; if **flag** $\geq$ **0**, trailing blanks are removed.

Mode: **Text**

Function trunc.f (arg)

Arguments:
arg A **double** value.

Description: Returns the truncated value of **arg.**

Mode: **Integer**

Function ttoa.f (arg)

Arguments:

arg	A **text** value.

Description: Returns the first character of **arg.**

Mode: **Alpha**

Function uniform.f (low, high, stream)

Arguments:

low A **double** value specifying the beginning value.
high A **double** value specifying the ending value.
stream An **integer** value specifying the random number stream.

Description: Returns a random sample uniformly-distributed between **low** and **high.**

Mode: **Double**

Function upper.f (arg)

Arguments:
arg A **text** value.

Description: Returns a copy of **arg** with each lower-case character converted to uppercase.

Mode: **Text**

Function val.f (any)

Arguments:

Description: Indicates an argument to a FORTRAN routine is passed by value.

Mode: n.a.

Function weekday.f (time)

Arguments:
time A **double** value specifying a cumulative event time.

Description: Returns the weekday portion corresponding to the event time.

Mode: **Integer**

Function weibull.f (shape, scale, stream)

Arguments:
shape A **double** value greater than zero specifying the shape.
scale A **double** value greater than zero specifying the scale .
stream An **integer** value specifying the random number stream.

Description: Returns a random sample from a Weibull distribution.

Mode: **Double**

Function xor.f (arg1, arg2)

Arguments:

arg1 An **integer** value.
arg2 An **integer** value.

Description: Returns the logical difference of **arg1** and **arg2.**

Mode: **Integer**

Function year.f (time)

Arguments:

time A **double** value specifying a cumulative simulation time.

Description: Returns the year portion corresponding to the simulation time based on values given to **origin.r.**

Mode: **Integer**

C.2 Global Variables

between.v

Description: If non-zero, specifies a routine which is called before each event or process is executed. The default is zero.

Mode: **subprogram**

buffer.v

Description: Specifies the length of **the buffer**. The default is 132.

Mode: **Integer**

dir.name.v

Description: Contains the directory the program was run from.

Mode: **Text**

eof.v

Description: For the current input unit, specifies, the action to take when end-of-file is encountered. If **eof.v** = 0 (the default), the program is aborted with a runtime error. If **eof.v** = 1, the program is not aborted and **eof.v** is set to 2.

Mode: **Integer**

event.v

Description: Contains the event/process class of the event or process to occur next.

Mode: **Integer**

events.v

Description: Contains the number of event/process classes.

Mode: **Integer**

f.ev.s(i)

Description: Contains the first-in-set pointer of the event set, **ev.s**, for event/process class "**i**".

Mode: **Pointer**

heading.v

Description: If non-zero, specifies for the current output unit a page-heading routine which is called for each new page. The default is zero.

Mode: **Subprogram**

hours.v

Description: Specifies the number of hours per simulated day. The default is 24.0.

Mode: **Double**

l.ev.s(i)

Description: Contains the last-in-set pointer of the event set, **ev.s**, for event/process class "**i**".

Mode: **Pointer**

line.v

Description: Contains, for the current output unit, the line number of the current line within the current page.

Mode: **Integer**

lines.v

Description: Specifies whether pagination is enabled for the current output unit. If **lines.v**=0 (the default), pagination is disabled. If **lines.v**>0, pagination is enabled and **lines.v** specifies the number of lines per page.

Mode: **Integer**

mark.v

Description: Specifies the termination character for external event data and random variable data. The default is "*".

Mode: **Alpha**

minutes.v

Description: Specifies the number of minutes per simulated hour. The default is 60.0.

Mode: **Double**

n.ev.s(i)

Description: Contains the number of events or processes of event/process class "**i**" in the event set, **ev.s.**

Mode : **Integer**

page.v

Description: For the current output unit, contains the page number of the current page.

Mode: **Integer**

pagecol.v

Description: Specifies for the current output unit whether a line containing the page number should be written automatically as the first line of each page. If **pagecol.v** > 0, this feature is enabled and **pagecol.v** specifies the starting column of the phrase, "**PAGE nnnn**". If **pagecol.v** = 0 (the default), this feature is disabled.

Mode: **Integer**

parm.v(i)

Description: Contains the "**i**"th command-line parameter.

Mode: **Text**

process.v

Description: If non-zero, contains a pointer to the process notice of the currently-executing process. If zero, no process is executing.

Mode: **Pointer**

prog.name.v

Description: Contains program name. Any directory information is removed.

Mode: **Text**

prompt.v

Description: The string of characters to be output when reading an input from terminal.

Default is "".

Mode: **Text**

rcolumn.v

Description: For the current input unit, contains the column number of the last character read from the current record, or zero if no character has been read from the current record.

Mode: **Integer**

read.v

Description: Contains the unit number of the current input unit.

Mode: **Integer**

record.v(i)

Description: Contains the number of records read from or written to unit number “**i**”.

Mode: **Integer**

ropenerr.v

Description: If non-zero indicates that an error occurred opening the current input unit.

Mode: **Integer**

rreclen.v

Description: For the current input unit, contains the length of the current record.

Mode: **Integer**

rrecord.v

Description: Contains the number of records read from the current input unit.

Mode: **Integer**

seed.v(i)

Description: Contains the seed value used to generate a random number from stream “**i**”.

Mode: **Integer**

time.v

Description: Contains the current simulated time.

Mode: **Double**

wcolumn.v

Description: For the current output unit, contains the column number of the last character written to the current record, or zero if no character has been written to the current record.

Mode: **Integer**

wopenerr.v

Description: If non-zero indicates that an error occurred opening the current output unit.

Mode: **Integer**

wrecord.v

Description: Contains the number of records written to the current output unit.

Mode: **Integer**

write.v

Description: Contains the unit number of the current output unit.

Mode: **Integer**

C.3 Attributes

The following attributes are automatically declared for an event or process notice:

eunit.a

Description: Contains zero for an endogenous event. Contains the unit number for an exogenous event.

Mode: **Integer**

m.ev.s

Description: Contains 1 if the notice is in the event set, **ev.s**. Contains 0 if it is not in the event set.

Mode: **Integer**

p.ev.s

Description: Contains a pointer to the event set predecessor.

Mode: **Pointer**

s.ev.s

Description: Contains a pointer to the event set successor.

Mode: **Pointer**

time.a

Description: Contains the simulated time at which the event or process is to occur, or for an **interrupt**ed process, the amount of time left to **work** or **wait.**

Mode: **Double**

The following attributes are automatically declared for a process notice only:

f.rs.s

Description: Contains the first-in-set pointer for the set of resources owned by the process.

Mode: **Pointer**

ipc.a

Description: Contains the process class corresponding to "**I.process**".

Mode: **Integer**

rsa.a

Description: Contains a pointer to the recursive storage save area for a suspended process.

Mode: **Pointer**

sta.a

Description: Contains the state of the process - 0 if passive (**wait**ing), 1 if active (**work**ing), 2 if **suspend**ed, or 3 if **interrupt**ed.

Mode: **Integer**

C.4 Constants

exp.c

Description: The value of "e", 2.718281828459045.

Mode: **Double**

inf.c

Description: The largest representable **integer** value.

Mode: **Integer**

pi.c

Description: The value of pi, 3.141592653589793.

Mode: **Double**

radian.c

Description: The number of degrees per radian, `57.29577951308232.`

Mode: **`Double`**

`rinf.c`

Description: The largest representable **`real`** value.

Mode: **`Double`**

Appendix D 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