Table of Contents

Preface .......................................................................................................................... ix

Overview of Chapters .................................................................................................. ix
  Part 1: User Guide .................................................................................................... ix
  Part 2: Reference ....................................................................................................... ix

Text Conventions ......................................................................................................... x
  Font Conventions ........................................................................................................ x

Variable Naming Rules ................................................................................................. x

Other Conventions ........................................................................................................ x
  Requests for Bug Fixes and Enhancements ................................................................. xi

Getting Help.................................................................................................................. xi

Part 1: User Guide ........................................................................................................ xii

  1 Overview ................................................................................................................ 1
    1.1 Introduction ........................................................................................................ 1
    1.2 Input Files .......................................................................................................... 1
    1.3 Object Orientation ............................................................................................. 2
    1.4 Air Breathing Thermodynamic Gas Properties .................................................. 2
    1.5 Model Definition ................................................................................................ 2
      1.5.1 Elements, Sockets, Subelements, Functions, and Tables ......................... 4
      1.5.2 Ports and Links ........................................................................................... 4
      1.5.3 Assemblies ................................................................................................. 5
      1.5.4 Executives .................................................................................................. 5
      1.5.5 The Solver .................................................................................................. 5
      1.5.6 Output Data Viewers .................................................................................. 6
    1.6 Distributed Modeling Via CORBA ....................................................................... 6
    1.7 Running the Model ............................................................................................. 6
      1.7.1 Batch Mode ................................................................................................. 6
      1.7.2 Interactive Mode .......................................................................................... 6

  2 Syntax Guide ............................................................................................................. 7
    2.1 Command Line Syntax ........................................................................................ 7
    2.2 General NPSS Input Syntax ................................................................................. 9
      2.2.1 Introduction to the Input ............................................................................ 9
      2.2.2 Some General Features of the Input ............................................................. 9
      2.2.3 Preprocessor Commands .......................................................................... 12
      2.2.4 Variables ................................................................................................. 15
      2.2.5 Programming Constructs .......................................................................... 34
      2.2.6 Functions ................................................................................................. 42
      2.2.7 Tables ....................................................................................................... 46
    2.3 Interactive Mode Syntax ...................................................................................... 55
      2.3.1 Introduction to Interactive Mode Syntax .................................................... 55
      2.3.2 Entering Interactive Mode ......................................................................... 56
      2.3.3 Breakpoints ............................................................................................... 57
      2.3.4 Interactive Debugging Commands ............................................................... 59

  3 Acquiring Components in NPSS .............................................................................. 61
    3.1 Interpreted Component Load on Demand (ICLOD) ........................................... 62
    3.2 Overview of Dynamically Loadable Modules (DLMs) ....................................... 62
    3.3 DLM Component Load on Demand (DCLOD) .................................................. 63
    3.4 How NPSS Uses CMFs ....................................................................................... 63
# Table of Contents

3.5 Component Creation Query Function findResource() ........................................ 64
3.6 Metadata .................................................................................................................. 64
4 Common Tasks ............................................................................................................. 65
4.1 Thermodynamics Packages ..................................................................................... 65
4.2 Elements ................................................................................................................... 66
4.3 Ports ......................................................................................................................... 68
  4.3.1 Creating Ports ...................................................................................................... 68
  4.3.2 Linking Ports ..................................................................................................... 70
  4.3.3 DataPorts ......................................................................................................... 72
  4.3.4 FilePorts .......................................................................................................... 73
4.4 Subelements .............................................................................................................. 74
4.5 Sockets ...................................................................................................................... 75
  4.5.1 Socket Attributes ............................................................................................. 76
  4.5.2 Determining Allowable Socket/Subelement Combinations ............................. 77
  4.5.3 Inserting Subelements into Sockets .................................................................. 77
  4.5.4 Inserting Functions and Tables into Sockets ..................................................... 78
  4.5.5 list() Function and Sockets ............................................................................. 79
4.6 Assemblies ................................................................................................................ 80
  4.6.1 Order of Execution in Assemblies ..................................................................... 81
4.7 Bleed Usage .............................................................................................................. 84
  4.7.1 Compressor Element Bleed Extraction ............................................................. 84
  4.7.2 Turbine Element Bleed Entry ............................................................................ 84
  4.7.3 Bleed Element Extraction and Entry ................................................................. 85
  4.7.4 Linking Bleed Flows ........................................................................................ 85
  4.7.5 Bleed User Functions .................................................................................... 85
4.8 Solver Setup ............................................................................................................. 87
  4.8.1 Basics .............................................................................................................. 87
  4.8.2 Critiquing Solver Setup .................................................................................... 88
  4.8.3 Adding to the Automatic Solver Setup ............................................................. 90
  4.8.4 Saving and Restoring A Solver Setup .............................................................. 92
  4.8.5 Solvers and Assemblies ................................................................................ 92
4.9 Input and Output ..................................................................................................... 93
  4.9.1 File Streams ................................................................................................... 93
  4.9.2 DataViewers .................................................................................................. 98
4.10 Basic Model Running ............................................................................................. 110
4.11 Moving and Deleting Objects ................................................................................. 112
4.12 Adding or Removing Elements from a Model ....................................................... 113
4.13 Prepass, Preexecute and Postexecute Functions .................................................. 115
4.14 Error Handling and Messages .............................................................................. 117
  4.14.1 ESO (Engineering Status Objects) ................................................................. 117
  4.14.2 ESORegistry ................................................................................................. 117
  4.14.3 ESORegistry: Identification Attributes (Unchangeable) ................................. 117
  4.14.4 ESORegistry: Changeable Attributes .......................................................... 118
  4.14.5 ESORegistry Functions ............................................................................... 119
  4.14.6 ESO Convenience Functions ...................................................................... 120
  4.14.7 errHandler Object Behavior ....................................................................... 120
  4.14.8 errHandler Variables .................................................................................. 121
  4.14.9 errHandler Functions .................................................................................. 122
  4.14.10 ESO: Examples ......................................................................................... 123
4.15 Data Matching and Analysis ............................................................................... 124
5 Building a Model ........................................................................................................ 126
  5.1 Study Executive File ............................................................................................ 127
5.1.1 Define Thermo Package .............................................................. 127
5.1.2 Include Standard NPSS Utilities and Components .................. 127
5.1.3 Include Special Utilities and Components ................................. 127
5.1.4 Include Model Definition .......................................................... 128
5.1.5 Model Case Description Files ................................................... 128
5.2 Model File .................................................................................... 128
  5.2.1 Header ..................................................................................... 128
  5.2.2 Instantiating Model Components .............................................. 129
  5.2.3 Linking Elements .................................................................... 137
  5.2.4 Solver Setup ............................................................................. 139
5.3 Case Input File ............................................................................. 140
  5.3.1 Design Point ............................................................................ 141
  5.3.2 Offdesign Steady-state Points .................................................. 141
  5.3.3 Transient Points ....................................................................... 142
5.4 fanjet Source Files ........................................................................ 143
  5.4.1 fanjet.run ................................................................................ 143
  5.4.2 Transient.view ........................................................................ 143
  5.4.3 fanjet.mdl ................................................................................. 144
  5.4.4 fanE3.map ............................................................................... 153
  5.4.5 IpcE3.map ............................................................................... 153
  5.4.6 hpcE3.map ............................................................................... 153
  5.4.7 hptE3.map ............................................................................... 153
  5.4.8 IptE3.map ............................................................................... 154
  5.4.9 Controls.cmp ........................................................................... 154
  5.4.10 desOD.case ........................................................................... 157
  5.4.11 viewOut ................................................................................... 158
  5.4.12 tout ......................................................................................... 161
6 Solver ............................................................................................... 162
  6.1 Introduction ................................................................................ 162
  6.2 Basic Theory of Solver Operation .............................................. 162
  6.3 General Structure of the NPSS Solver ...................................... 165
  6.4 Creating a Solver ........................................................................ 165
  6.5 Independent Variables ............................................................... 166
  6.6 Dependent Conditions ............................................................... 168
  6.7 Solver Setup ............................................................................... 171
  6.8 Constraint Handling ................................................................... 174
    6.8.1 Constraint Definition and Usage ........................................... 174
    6.8.2 Adding Constraints Dependents to Target Dependents .......... 176
    6.8.3 Constraint Groups ................................................................. 179
    6.8.4 Removing Constraints ......................................................... 180
    6.8.5 Temporarily Disabling Constraints ..................................... 180
    6.8.6 Resolving Constraint Slope Conflicts .................................. 181
    6.8.7 Resolving Min/Max Constraint Conflicts .............................. 182
7 Transient Simulation ........................................................................ 183
  7.1 Introduction ................................................................................ 183
  7.2 Running a Basic Transient Simulation ...................................... 187
    7.2.1 Basic Setup ............................................................................ 187
    7.2.2 Output .................................................................................... 187
    7.2.3 Initial Conditions ................................................................... 188
    7.2.4 Transient Executive .............................................................. 188
    7.2.5 Integration Method ............................................................... 189
    7.2.6 Setup of Transient Mode ...................................................... 189
    7.2.7 Time Step .............................................................................. 190
7.2.8 Termination Criteria ................................................................. 190
7.2.9 Running the Transient .............................................................. 191

7.3 Variable Time-Step Computation ............................................... 192
7.3.1 Adaptive Time Stepping ........................................................... 192
7.3.2 User-Defined Time-Step Computations .................................. 193

7.4 Controlling Termination of a Transient Run .............................. 194

7.5 TransientExecutive Attributes .................................................. 196

7.6 Integrators .................................................................................. 198
7.6.1 Implicit Integration ................................................................. 198
7.6.2 Explicit Integration ............................................................... 199
7.6.3 Switching Individual Integrators to Steady-State ..................... 199

8 Linear Model Generator ............................................................... 201
8.1 Introduction ............................................................................... 201
8.2 Basic Setup ............................................................................... 201
8.2.1 Creation and Setup of the Linear Model Generator ................. 202
8.2.2 Setup of the Solver During Linear Model Generation .......... 203
8.2.3 Modifying the Linear Model Setup ......................................... 203
8.2.4 Generating a Linear Model .................................................... 204
8.2.5 Printing/Saving a Linear Model ............................................. 204

8.3 Functions .................................................................................. 205
8.4 User Input Attributes ............................................................... 208
8.5 Output (Calculated) Attributes .................................................. 209

9 Hierarchical Data Format (HDF) Guide ....................................... 211
9.1 Overview ................................................................................ 211
9.2 Syntax .................................................................................... 211
9.3 User-accessible Functions ......................................................... 214

Part 2: Reference .............................................................................. 216
10 Assembly Attributes ................................................................. 217

11 Units ....................................................................................... 218
11.1 Units Functions ....................................................................... 218
11.2 Valid Unit Strings ................................................................... 221

12 Function Summary ..................................................................... 226
12.1 User-Accessible Member Functions ........................................ 226
12.1.1 Functions for All NPSS Objects (Elements, Subelements, Assemblies, DataViewers, Solver,
Ports, Stations, Sockets) and Variables ........................................ 226
12.1.2 Array Functions .................................................................. 234
12.1.3 Assembly Functions ............................................................ 234
12.1.4 BleedInPort/BleedOutPort Functions .................................. 235
12.1.5 DataViewer Functions ........................................................ 235
12.1.6 Dependent Functions .......................................................... 236
12.1.7 Element, Assembly, and Subelement Functions .................. 236
12.1.8 errHandler Functions .......................................................... 236
12.1.9 External Component Attributes and Functions .................... 237
12.1.10 FlowStation Functions ......................................................... 238
12.1.11 Independent Functions ........................................................ 246
12.1.12 InFileStream Functions ...................................................... 246
12.1.13 Linear Model Generator Functions ..................................... 247
12.1.14 Link Functions ................................................................... 247
12.1.15 msgHandler Functions ....................................................... 248
12.1.16 OutFileStream Functions .................................................... 248
12.1.17 Port Functions .................................................................... 249
16.1.3 The preconvergence Function ................................................................. 291
16.1.4 Solver User Input Attributes ................................................................. 292
16.1.5 Solver Output (Calculated) Attributes .................................................. 297
16.2 Independent (NCPIndependent Class) ....................................................... 299
16.2.1 Independent Definition Example ......................................................... 299
16.2.2 Independent Limits ............................................................................... 300
16.2.3 Independent Input/Commands ............................................................... 300
16.2.4 Independent User Input Attributes ...................................................... 301
16.2.5 Lock Attributes for Independents ......................................................... 302
16.2.6 Independent Output (Calculated) Attributes ........................................ 303
16.3 Dependent (NCPDependent class) ............................................................ 303
16.3.1 Examples of Dependent Definition ..................................................... 303
16.3.2 Dependent Input/Commands ................................................................. 305
16.3.3 Dependent User Input Attributes ......................................................... 306
16.3.4 Lock Attributes for Dependents ........................................................... 308
16.3.5 Dependent Output (Calculated) Attributes ......................................... 308
16.4 ConstraintGroups (ConstraintGroup class) .............................................. 309
16.4.1 ConstraintGroup Input/Commands ...................................................... 309
16.4.2 ConstraintGroup User Input Attributes .............................................. 309
16.4.3 ConstraintGroup Output (Calculated) Attributes .................................. 310
16.5 DiscreteStateVariable (DSV) ................................................................. 310
16.5.1 Nested vs. Simultaneous DSV Solution ............................................... 311
16.5.2 Controlling Movement through the Allowed State Values .................. 311
16.5.3 Controlling Repetition of Previously Failed Iterations ...................... 311
16.5.4 Example Definitions of DiscreteStateVariables ................................. 312
16.5.5 Using a DSV in a Model ...................................................................... 313
16.5.6 DSV Integer Associations .................................................................. 314
16.5.7 DSV allowedStrings and allowedInts .................................................. 315
16.5.8 DSV Input Commands ....................................................................... 315
16.5.9 DSV User Input Attributes ................................................................. 315
16.5.10 DSV Output (Calculated) Attributes ................................................ 317
16.6 Advanced Solver Member Functions ...................................................... 317
16.7 Solver Diagnostic Output ......................................................................... 320
16.7.1 Searching Diagnostic Output .............................................................. 321
16.7.2 Grepable Output .................................................................................. 321
16.7.3 Option Variables .................................................................................. 322
16.7.4 Interactive Debug Output Functions .................................................. 322
16.7.5 userReport: User-defined Solver Output Function ............................. 323
16.7.6 projectionReport: User-defined Solver Output Function .................. 325
16.7.7 dxLimitAll: User-defined Solver Output Function .............................. 325
16.8 Solver Diagnostic and Trouble-Shooting Reference .............................. 326
16.9 Solver Terms ......................................................................................... 327

17 Transient Reference Guide ........................................................................ 330
17.1 Transient (TransientExecutive class) .................................................... 330
17.1.1 Initial Model State Assumptions ......................................................... 330
17.1.2 Initialization of the Time Update Loop ............................................... 330
17.1.3 Time Update Loop ............................................................................. 331
17.1.4 Independent variable predictions ....................................................... 337
17.1.5 Time Step Computation ..................................................................... 338
17.1.6 Utility Functions .................................................................................. 340
17.1.7 TransientExecutive Functions ............................................................ 341
17.1.8 Transient User Input Attributes ........................................................ 342
17.1.9 Transient Output (Calculated) Attributes .......................................... 344
17.2 Integrator (NCPStateIntegrator class) ..................................................... 345
17.2.1 Creating an Integrator ................................................................. 345
17.2.2 Implicit Integration ................................................................. 345
17.2.3 Explicit Integration ................................................................. 347
17.2.4 Integrator Functions ............................................................... 347
17.2.5 Integrator Input Attributes ..................................................... 348
17.2.6 Locking Attributes for Integrator .......................................... 348
17.2.7 Integrator Output (Calculated) Attributes ............................. 349
17.2.8 State Substitution Variables .................................................. 349

17.3 Example Program .............................................................................. 350
17.3.1 Model File ................................................................................ 350
17.3.2 First Order Lag Test File ......................................................... 351
17.3.3 Test Output File ....................................................................... 359

17.4 Functions for Time-Discrete Model Objects ............................... 360
17.4.1 Time Discrete Example Test File ............................................. 361
17.4.2 Time Discrete Test Output File ................................................. 363

18 Glossary .............................................................................................. 364
Index ....................................................................................................... 369
List of Figures

Figure 1. Engine Model .................................................................................................................. 3
Figure 4. Time March Executive Event Sequence ........................................................................ 333
Figure 5. Time Sequence - Step 2 ............................................................................................... 334
Figure 6. Time Sequence - Step 3 ............................................................................................... 334
Figure 7. Time Sequence - Step 4 ............................................................................................... 335
Figure 8. Time Line ....................................................................................................................... 336
Preface

This User Guide is for engineers who will be using NPSS. This includes those who run the code, those who modify and edit its input files, and those who develop complete models. Part 1 is designed to provide complete explanations of the NPSS features used by most users for the majority of their performance-related work. Chapters 1-5 contain basic information about the system, its syntax, and the structure of a model. Chapters 6 and 16 contain more detailed information about the NPSS systems that solve steady-state and transient problems. Chapters 1-6 make occasional reference to information for more advanced users contained in the Developer Guide. The content of chapters 1-6 should be considered prerequisite for the material in the Developer Guide.

Chapters 10-18 constitute part 2 of the Guide. Part 2 is intended to give complete reference information on important parts of the NPSS system. For example, a discussion of the most commonly used solver features is given in Part 1 (Chapter 6), but a complete list of all features is given in Part 2 (Chapter 16).

Overview of Chapters

Part 1: User Guide

1 Overview introduces the model structure and presents basic information about the input file. It introduces terms and presents a general view of the system discussed in detail in later chapters.
2 Syntax Guide explains (1) the command line syntax, (2) the general NPSS syntax used to create input files, and (3) interactive mode syntax. It provides the basic fundamentals of how to run the system.
3 Acquiring Components in NPSS provides information on Creation Method Facilities (CMFs), which are important for using pre-written NPSS components—either supplied with NPSS or written at a particular site.
4 Common Tasks provides information on the practical tasks involved in using NPSS syntax to construct and use models.
5 Building a Model shows how to build a model using the concepts in Chapters 1–4. It presents a complete, working NPSS model of a turbofan gas turbine engine.
6 Solver introduces the basic theory and structure of the solver. It discusses setting up the solver for steady-state running and using constraints.
7 Transient Simulation provides information on setting up and running transient simulations.
8 Linear Model Generator provides introductory information on the Linear Model Generator and its setup. Functions and attributes are also included in this chapter.
9 The Hierarchical Data Format (HDF) Guide discusses a basic implementation of HDF file format for binary data storage.

Part 2: Reference

10 Assembly Attributes contains attributes specific to an assembly.
11 Units is a list of unit strings accepted by the system.
12 Function Summary provides the member functions for the various objects, global functions, interactive mode functions, suites of functions, and built-in math functions.
13 AutoDoc is a short chapter that explains how to run AutoDoc to generate html files on a specific class or a list of classes.
14 DataViewer Reference Sheets lists characteristics common to all data viewers and variables specific to certain types of data viewers.
15 Stream Reference Sheets provides attributes on input and output streams.
16 The NPSS Solver Reference contains all the commands and attributes for the solver classes, sections on output and troubleshooting, and solver terms. It contains commands for more advanced users.
17 Transient Reference Guide provides Transient Executive and Integrator attributes and functions, and functions for Time-Discrete Model Objects.
18 Glossary contains some basic terminology (excluding solver-specific terms, which are included in the solver reference chapter).
Text Conventions

Font Conventions

The font conventions below are followed in this document.

<table>
<thead>
<tr>
<th>Font</th>
<th>Used For</th>
</tr>
</thead>
<tbody>
<tr>
<td>Times New Roman</td>
<td>Body of the document</td>
</tr>
<tr>
<td>Times New Roman Bold</td>
<td>Emphasizes text</td>
</tr>
<tr>
<td>*Times New Roman Italic</td>
<td>Titles of Documents</td>
</tr>
<tr>
<td>*Times New Roman Italic</td>
<td>Emphasizes the introduction of new terms and concepts</td>
</tr>
<tr>
<td>Courier New (constant width)</td>
<td>Examples of code, file contents, and program output</td>
</tr>
<tr>
<td>*Courier New or Times New Roman Italic</td>
<td>Items for which the user must supply context-specific information. For example, the variable Var1 would be replaced by an actual variable name supplied by the user.</td>
</tr>
</tbody>
</table>

Variable Naming Rules

When you create variable names for the new components you want to add, please use the variable naming rules that were followed for the NPSS built-in components. These naming conventions are provided below.

Naming Conventions

- The first and second words in the name alternate case of the starting letter; remaining words all start with upper case followed by lower case.
- Names start with lower case letters except for common usage names: A, P, T, C, F, RNI
  - Axxx : Area associated with xxx
  - P: pressure
  - T: temperature
  - Cxxx: Coefficient of xxx
  - Fxxx: thrust
- Fluid Ports start with Fl_
- Fuel Ports start with Fu_
- Sockets start with S_
- Tables start with TB_
- Shafts start with Sh_
- adders start with a_
- Scalars start with s_
- Subscripts for "total" and "static" are always lower case: t, s
- Base added to the end of a variable name refers to a variable that is passed from a socket.
- Map added to the end of a variable name refers to variables used to read a table.
- hj added to the end of a variable is associated with heat transfer.
- All option switches begin with switch.
- Two letters abbreviate familiar ratios: PR, AR, TR (i.e., pressure ratio, area ratio, and temperature ratio)
- eff is always assumed adiabatic efficiency; if polytropic, label as effPoly.
- errXxx : error in Xxx
- expXxx : exponent on Xxx
- fracXxx : fraction of xxx

Other Conventions

As noted above italics are used in generic examples to indicate that the user must substitute a specific name. A generic example might be:
In real use, the user would supply specific names for the items in *italics*, such as:

```cpp
solver.addIndependent("HPSpeedIndep");
```

In general, examples are indented. However, some long sections of code have not been indented. Throughout this document the reader may notice occasional names that include the letters "NCP," for example, "NCPDependent." Earlier versions of NPSS were known as NCP so the system's code still contains references to it.

### Requests for Bug Fixes and Enhancements

If you have a question, a defect, or an enhancement you want addressed, check with your site representative first, since you may have a problem related to your site's installation of NPSS. If your problem is an NPSS problem that has not been addressed, your site representative can enter a change request (CR) in the Web Site Change Request Tracking System or contact the NPSS Software Configuration Manager (SCM) who will enter the request for them. If you have a question that your contact person cannot answer, that person is responsible for contacting the appropriate person at Wolverine Ventures, Inc. You may also receive help via email. Please see the following section.

### Getting Help

Check with your local expert, then if there are further question that requires assistance, you can send an e-mail to the following address:

```
support@wolverine-ventures.com
```

Your e-mail must include the following information. Please double space between these items.

1. The **category** of the request for help: question or problem
2. The **submitter** of the problem, bug, etc., so the person can be contacted when the issue is resolved.
3. The **version** of the system you are running: npss –v
4. The **platform** on which you are running the system.
5. A **detailed description** of the question/problem. Explain exactly what you were doing or what action you completed when the problem occurred. You may want to include the path where the problem occurred.
6. A copy of the **input file** you were working with when you encountered a problem. To expedite the process, please delete from the input file any information that does not pertain to your problem or question. Send the pared down copy of the input file with your email. You may send it as an attachment.
7. The **error message** you received. Please provide the exact wording of the message.
Part 1: User Guide
1 Overview

1.1 Introduction

NASA Glenn Research Center, in conjunction with the U.S. aeropropulsion industry and the Department of Defense, developed technologies capable of supporting detailed aerothermomechanical computer simulations of complete aircraft engines. This project is called the Numerical Propulsion System Simulation (NPSS). NPSS can realistically model the physical interactions that take place throughout an engine, accelerating the concept-to-production development time and reducing the need for expensive full-scale tests and experiments. Its architecture supports implementing one-, two-, and three-dimensional models of the engine flow field and structure, and reconciling them with zero-dimensional component-based models.

At its foundation, NPSS is a component-based object-oriented engine cycle simulator designed to perform cycle design, steady-state and transient off-design performance prediction, test data matching, and many other traditional tasks of engine cycle simulation codes. Like traditional codes, an NPSS engine model is assembled from a collection of interconnected components, and controlled through the implementation of an appropriate solution algorithm. Historically, limited computer resources restricted component representations used in these simulations to be simple characterizations of empirical test results, or the results of more sophisticated component models run separately. NPSS, however, is capable of calling upon more sophisticated component models directly. Using the computer industry's Common Object Request Broker Architecture (CORBA) communication standard, NPSS can interact with external codes running on other computers distributed across a network. The advanced system architecture designed into NPSS will allow the marriage of design tools at varying levels of dimensional fidelity across multiple technology disciplines.

The initial focus of NPSS was on the aerothermodynamic cycle simulation process, which includes the following:

- All model definition through input file(s)
- NIST (National Institute of Standards and Technology) compliant thermodynamic gas-properties package
- A sophisticated solver with auto-setup, constraints, and discontinuity handling
- Steady-state and transient system simulation
- Flexible report generation
- Built-in object-oriented programming language for user-definable components and functions
- Support for distributed running of external code(s) via CORBA
- Support for test data matching and analysis

This chapter presents a brief introduction to the system and its features.

1.2 Input Files

An NPSS engine model may be completely defined in one or more input files. No alteration of NPSS source code is required. Input files may be created and modified using any text editor. Input files contain commands and directives that do the following:

- Identify what thermodynamic gas-properties package to use.
- Define a model's components and component linkages, tell NPSS what calculations to perform, and specify what output to generate.
- Initiate the solution of specified cases.

NPSS input syntax is a full-featured programming language modeled after C++. It allows definition of real (floating point), integer, and string variables, up to three-dimensional arrays, "if...else if...else" (branching) constructs, looping controls, and conditional preprocessing directives. Floating point variables can be assigned units, and a facility exists to automatically convert variables and expressions from one set of units to another.

Refer to Chapter 2, Syntax Guide, and Chapter 4, Common Tasks, for detailed information on the NPSS input language, and how to use it to define a model. Chapter 5, Building a Model, presents a complete example.
1.3 Object Orientation

NPSS is an object-oriented program, and its input reflects this. Object orientation means that the program's structure revolves around objects of various types, each of which has a set of attributes accessible by other objects, and certain functions—a set of actions particular to the object that can be performed at the request of other objects. See 2.2.2.1 for more information on object orientation.

1.4 Air Breathing Thermodynamic Gas Properties

Several thermodynamic gas property packages are supplied with NPSS to support air breathing engine analysis (aircraft and industrial gas turbine engines). Their modular design allows a user to select the desired package at run time. One package ("Janaf") offers flexibility and matches the NIST standard (NIST–JANAF, Revision 3) at the expense of some computational speed. A second package ("GasTbl"), created by Pratt & Whitney and based on NASA's "Therm," includes humidity calculations as well as some chemical equilibrium capabilities. The "CEA" thermo is an implementation of the NASA chemical equilibrium code. "allFuel," from General Electric, contains both gas properties and fuel properties. It is generally consistent with the NASA TP-1908 thermo but specific agreement varies with the fuel and working fluid options selected. A fluid property table “FPT” package is also available. It allows users to define NPSS tables and/or functions to describe the thermodynamic properties of the fluid. Section 4.1 discusses declaration of thermodynamics packages.

1.5 Model Definition

Before using NPSS, one should be familiar with the structure of the model and with some terminology. Figure 1 graphically depicts a simple model using phrases described in this section. Briefly stated, a model will encapsulate a variety of component elements, some of which may have sockets connected to subelements, functions, or tables, and ports connected by links to other elements. Elements may be grouped into assemblies, and each assembly may have a dedicated solver. In addition, the model definition will include specifications for generating formatted output using one or more data viewers.
Figure 1. Engine Model

Simple Conceptual Engine Model
1.5.1 Elements, Sockets, Subelements, Functions, and Tables

Elements are the main building blocks of an engine model in NPSS and generally represent the major components of the engine (e.g., compressors, combustors, turbines). Elements contain variables, also called attributes. These represent quantities appropriate for a given component, such as physical characteristics, scale factors, and gas properties. Some element variables may be option variables, which have a limited set of values that may be assigned to them (e.g., "DESIGN" and "OFFDESIGN"). These are typically used to control the behavior of the element. Section 2.2.4 discusses variables in detail. The attributes associated with each element (and with other NPSS objects) are given in the NPSS Reference Sheets.

Some elements contain sockets. Sockets are a software vehicle to "plug in" calculations needed by the element. For example an element modeling a compressor might have a socket through which the compressor map information is provided. The most common socket plug-ins are subelements: element-like objects designed to support the object to which they are attached. In the example above, a subelement would perform the compressor map lookups to be supplied through a socket to the compressor element. Some subelements also have sockets that can accept other subelements. In general, wherever interchangeable computational approaches are required for an element, one or more subelements will be available. Each socket is of a distinct type, and each subelement must meet the requirements of that socket type in order to "fit." This checking prevents a turbine map subelement from being plugged into a compressor element, for example. Elements are discussed further in Section 4.2, and subelements in Section 4.4. Section 4.5 discusses sockets and their use.

NPSS comes with a standard set of elements and subelements that fulfill many modeling needs. Users may, however, create additional customized elements and subelements. This can be done entirely through normal input files if desired. For improved execution speed, it is also possible to run user-written elements and subelements through a separate conversion utility that changes the NPSS syntax into C++. The object definition can then be compiled and made available to any model as if it were a native part of NPSS. These topics are covered in the Developer Guide rather than in this document.

In addition to using components written in NPSS's native syntax, NPSS can use components written in other languages, such as FORTRAN or C. Compiled components can be added to the standard NPSS executable to produce a custom executable or a customer deck, or they can be compiled into Dynamically Loadable Modules (DLMs) and loaded at run time by an existing NPSS executable. NPSS can also use external components accessed through the Common Object Request Broker Architecture (CORBA). Information on loading components at run time is given in Chapter 3. Creating such components is discussed in the Developer's Guide.

Special calculations can also be specified through functions and tables. Functions are like program subroutines that can be called by direct command, or by objects such as elements, subelements, and other functions. Tables are special functions that perform table lookups. Many subelements either require or provide the opportunity for the user to supply information through tables. NPSS comes with many preprogrammed functions for various tasks, and the user can readily write new ones using NPSS syntax. Every element and subelement may have a user-supplied prePass function, preexecute function and postexecute function attached to them. These provide users with the ability to override variable values and compute extra attributes not computed by the element or subelement. Functions are discussed more fully in Section 2.2.6. The use of prePass and pre- and postexecute functions in particular is discussed in Section 4.13. Tables are discussed in Section 2.2.7. Sockets can also be designed to accept plug-in functions and tables, but this is covered in the Developer's Guide.

1.5.2 Ports and Links

Elements communicate with other elements through input and output ports that manage the flow of data through links. Elements may have zero to many ports, depending on their needs.

Ports can be categorized by the type of information communicated and the direction of information flow. NPSS ensures that a port is linked with another port of the same type and opposite direction. For example, a Compressor element could have three ports: a FluidInputPort linked to an Inlet element's FluidOutputPort, a FluidOutputPort...
linked to a Burner element's FluidInputPort, and a ShaftOutputPort linked to a Shaft element's ShaftInputPort. In addition, a Compressor element may have any number of InterStageBleedOutPorts.

A variety of port types is available. Fluid ports transfer primary-gas flow properties between elements (e.g., flow rate, total temperature and pressure, molecular constituents); special fluid ports are available for bleeds and leakages. Similarly, fuel ports are used to transfer fuel properties from a FuelStart element into a Burner. Shaft ports transfer mechanical properties such as torque and rotational speed between rotating components. DataPorts transfer a single numerical value, such as an engine measurement. FilePorts transfer one or more files. Ports are discussed in detail in Section 4.3.

1.5.3 Assemblies
A set of connected elements may be grouped as an assembly. The assembly is then treated by the rest of the model as if it were a single element. Selected element ports within the assembly are generally promoted to the assembly boundary so they may be connected with other elements and assemblies. Each assembly may have its own dedicated solver. An assembly can contain other assemblies. In fact, the entire model constitutes the top-level assembly: any other assemblies are contained within it. The top-level assembly is the only assembly that must have an Executive. The use of assemblies is discussed further in Section 4.6.

1.5.4 Executives
An object that may exist inside an Assembly or Element, which is responsible for the accounting for and running of executionSequences (and possible solving) is an Executive. The Executive class contains string array variables preExecutionSequence, executionSequence, and postExecutionSequence, which are explained further in Section 4.6.1. Classes derived from Executive may be solvers, but are not necessarily so. The NPSSSteadyStateSolver class and the NPSSTransientSolver class are both classes derived from Executive.

1.5.5 The Solver
The solver is the part of NPSS that drives the model to a valid solution. The top-level assembly always contains a solver, which is created for the user. This solver receives a run command and is responsible for iteratively adjusting the values of the model independent variables in order to satisfy the dependent conditions in the system. If convergence cannot be achieved within a specified number of iterations, an error is returned for that case. For transient simulations, the solver also controls the progression of time within the run, providing a converged solution at each point in time. More details about the solver's operation are presented in Chapter 6. Transient simulation is specifically discussed in Chapter 7.

Many Elements contain predefined independents and/or dependents. Should the user choose the "default" solver setup, this information is automatically used to solve the cycle. User-defined independents and dependents may also be added to, or used instead of, the defaults provided. Further, constraints may be imposed on the solver's solution. Basic solver setup is discussed in Section 4.8. The use of constraints is discussed in Section 6.8.

Any assembly in an NPSS model may contain its own solver. When an assembly with a solver is run, its solver attempts to converge that assembly by recursively calling any other assemblies it contains, and so on down to the bottom of the assembly tree. If an assembly does not have a solver, the independents and dependents in that assembly are handled by the solver in the assembly containing it. If an assembly has a solver, its independent variables are varied to satisfy its dependent conditions before control is returned to its parent assembly. A model may therefore contain a hierarchy of nested solvers, each responsible for solving a successively smaller portion of the model. The solver in the top level of the model is always responsible for controlling the time-step for transient simulations. The use of solvers in subassemblies is discussed in Section 4.8.5.

Each assembly may define one "pre-solve" sequence of objects that is executed once before every converged point, an "inner-loop" solve sequence that is run during the solution process, and a "post-solve" sequence that is run after the point is converged. If an assembly does not have a solver, the pre-, inner-, and post-solve sequences are executed sequentially, once, and the flow of control is returned to the calling level.
1.5.6 **Output Data Viewers**

*Data viewers* provide the mechanism for generating formatted output. Viewers are available for presenting model output in columnar format (i.e., one column per "case"), row format, full-page layout, and others. For each viewer, which data values are presented, how they are labeled, and where they appear is entirely under user control. Data "views" are defined in the model, and the viewers are called as needed while case running progresses. Information on constructing and using data viewers is found in Section 4.9.2.

A data viewer is connected to an output *stream*, which defines the output destination. Streams can be directed to a monitor, a disk file, or a connected printer. Output messages can also be generated and directed to a stream without the use of a data viewer. For example, progress messages can be directed to the monitor through the predefined stream "cout" (connected to the operating system's "standard output"), or user-generated warnings can be directed to "cerr" (connected to "standard error"). Input streams can also be set up to read information from sources outside NPSS. Input and output streams are discussed in Section 4.9.1.

1.6 **Distributed Modeling Via CORBA**

NPSS supports *distributed computing*. This means an NPSS model running on one computer can access components or complete NPSS models running on other computers. For example, an NPSS model running on a UNIX workstation could obtain its high-pressure compressor definition from a multi-stage 3D computational fluid dynamics program running on a supercomputer, accessed via the internet.

NPSS uses CORBA (Common Object Request Broker) as the agent to support distributed components and models. To aid in developing distributed components, a CORBA Component Developer's Kit is provided with the system.

1.7 **Running the Model**

After the model definition is complete, cases may be run. Option variables may be set, element variables adjusted, solver goals established, and a *run* command issued to process the point. After point convergence is achieved, output may be processed and a new setup established for the next case. More details can be found in Section 4.10.

1.7.1 **Batch Mode**

*Batch mode* operation is the default mode of running NPSS. Input files contain not only all model definition information, but also all case-running directives. When NPSS is executed, the model definition is processed and all cases are run. Output files collect the results of model running, as defined by the data viewers, and any appropriate errors and warnings are conveyed to the user. In this mode, execution stops when an *end-of-file* or an explicit *quit* command is encountered.

1.7.2 **Interactive Mode**

*Interactive* execution is also available in NPSS. In interactive mode the system presents the user with a command-line prompt. Individual commands may be typed directly into the system and executed immediately, after which the user is presented with another prompt. This mode can be triggered by a command-line option, or by commands inserted the input files.

All of the features available for batch mode running are available in interactive mode. Command-line editing capability is provided, as well as previous command recall. This mode is primarily intended for debugging purposes, but it is not limited to that task. It is discussed in detail in Section 2.3.
2 Syntax Guide

This chapter is divided into three major sections:

- Command Line Syntax: How to execute NPSS from a command line
- General NPSS Input Syntax: How to construct NPSS input files
- Interactive Mode Syntax: Special commands and features available in interactive mode

The middle section on General NPSS Input Syntax focuses on the basic building blocks of the NPSS input language: the preprocessor, variables, programming constructs, functions, and tables. To actually build and run a model, these items must be used, together with special NPSS commands, to create and connect more complex objects such as elements, subelements, and assemblies. These topics, and other topics of interest in constructing practical NPSS models, are discussed in Chapter 4, Common Tasks. Many of these concepts are illustrated in the discussion of a complete, working NPSS model in Chapter 5, Building a Model.

2.1 Command Line Syntax

NPSS is executed from a command line using the following syntax:

```sh
npss [-v] [-D varName=value] [-trace] [-debug] [-I dirName] [-i]
[-h] [-X customAssemblyType] [-E customExecutiveType] [-nosolver] file1.
```

where:
- `npss` is the name of the NPSS executable
- `file1`, `file2`, etc., are input files
- [] indicate optional items

The command line options are presented in Table 1 below. The meaning of some of these options is made clearer by material later in the User Guide.

<table>
<thead>
<tr>
<th>Command Line Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-v</code></td>
<td>Prints the NPSS program version and copyright information. Also reports whether this build is a DEBUG and/or OPTIMIZED build, and the CORBA implementation in use. Format is &quot;(ORB::orbName)&quot;, where <code>orbName</code> is either VisiBroker, Orbix, or MICO. If NPSS was not built with CORBA support, &quot;(CORBA disabled)&quot; is displayed.</td>
</tr>
</tbody>
</table>
| `-D varName=value`  | Defines a preprocessor variable named `varName` with an optional `value` (see Section 2.2.3.2). White space is optional between `-D` and `varName`. For example:

```sh
-D THERMO=Janaf
```

This defines preprocessor variable `THERMO` and assigns it the value "Janaf". No white space is allowed around the equals sign. Notice that the variable value is not quoted.

```sh
-DprintDiagnostics
```

This defines preprocessor variable `printDiagnostics` but assigns it no value.

<p>| <code>-trace</code>            | Causes the system to generate a trace output to the screen of all interpreted statements and function calls. Also see command <code>trace</code> in Section 2.3.4 and the <code>traceExecution()</code> global function. |</p>
<table>
<thead>
<tr>
<th>Command Line Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-debug</td>
<td>Runs in debug mode (breakPoints are active). See Section 2.3.3. Also enters interactive mode. (-i)</td>
</tr>
<tr>
<td>-I dirname</td>
<td>Adds the directory specified by dirname to the NPSS include path (see Section 2.2.3.1). For example: npss -I /NPSS/dev/AirBreathing/include -I../myDir myFile</td>
</tr>
<tr>
<td></td>
<td>A space after the -I is optional. Use a separate -I for each directory to be added to the include path.</td>
</tr>
<tr>
<td></td>
<td>The NPSS include path can also be defined using the NPSS_PATH environment variable, defined by a command before NPSS is executed.</td>
</tr>
<tr>
<td></td>
<td>In UNIX, directory pathnames in the NPSS_PATH variable are separated by colons. Different UNIX shells use different syntax for defining environment variables. Under the C shell, the syntax is: setenv NPSS_PATH &quot;/NPSS/dev/AirBreathing/include:../myDir&quot;</td>
</tr>
<tr>
<td></td>
<td>Under the Bourne shell, the syntax is: NPSS_PATH=&quot;/NPSS/dev/AirBreathing/include:../myDir&quot; export NPSS_PATH</td>
</tr>
<tr>
<td></td>
<td>In Windows, directory pathnames in the NPSS_PATH variable are separated by semicolons. To set an environment variable the syntax is: set NPSS_PATH=\NPSS\dev\AirBreathing\include;../myDir</td>
</tr>
<tr>
<td></td>
<td>If paths are defined in NPSS_PATH and more paths are defined on the command line using -I, the command line paths are added before those in NPSS_PATH, and in the order given on the command line. Chapter 3 discusses other environment variables used by NPSS and their relationship to NPSS_PATH.</td>
</tr>
<tr>
<td>-i</td>
<td>Enters interactive mode after reading and executing all input files specified (see Section 2.3).</td>
</tr>
<tr>
<td>-l dlmModule</td>
<td>Dynamically loads the specified DLM module on startup (see Section 3.2).</td>
</tr>
<tr>
<td>-log</td>
<td>Creates a log file of interactive commands (see Section 2.3). The file is named npss.log, and is placed in the same directory from which NPSS was launched.</td>
</tr>
<tr>
<td>-corba</td>
<td>Tells NPSS to check for CORBA elements when creating objects (see Sections Error! Reference source not found. and 3.4).</td>
</tr>
<tr>
<td>-ns nsRef</td>
<td>Specifies the server to use for NameService when looking for external components (see Section Error! Reference source not found.). The nsRef argument is an Interoperable Object Reference (IOR) either supplied directly on the command line or contained in a file, e.g.: npss -corba -ns IOR_filename</td>
</tr>
<tr>
<td></td>
<td>If nsRef is &quot;none&quot;, then no attempt is made to contact a name server.</td>
</tr>
<tr>
<td>-iclodfirst</td>
<td>Tells the NPSS to use ICLOD (Interpreted Component Load on Demand) before DCLOD (DLM Component Load on Demand) when searching for components. When both ICLOD and DCLOD are active (the default), DCLOD will search for the component first unless this option is used (see Section 3.4).</td>
</tr>
<tr>
<td>-nodclod</td>
<td>Disables automatic use of the DCLOD (DLM Component Load on Demand) facility (see Sections 3.3 and 3.4)</td>
</tr>
<tr>
<td>Command Line Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>-noiclod</td>
<td>Disables automatic use of the ICLOD (Interpreted Component Load on Demand) facility (see Sections 3.1 and 3.4)</td>
</tr>
<tr>
<td>-h</td>
<td>Displays a summary of these command line arguments.</td>
</tr>
<tr>
<td>-w</td>
<td>Allow the use of DLMs built with previous versions of NPSS. WARNING: This feature is dangerous and unpredictable for the novice user and its use is only intended for short term testing of dynamically linked modules. <em>Use of this option at your own risk.</em></td>
</tr>
<tr>
<td>-X assemblyName</td>
<td>Runs NPSS with an instance of the user-specified type <code>assemblyName</code> as the top-level object.</td>
</tr>
<tr>
<td>-nosolver</td>
<td>Runs NPSS without its default NPSSSteadyStateSolver and NPSSTransientSolver, and instead places a default Executive instance named “executive” in the top-level Assembly. This Executive contains and is able to run all of the execution sequences.</td>
</tr>
<tr>
<td>-E executiveName</td>
<td>Runs NPSS with an instance of the user-specified Executive <code>executiveName</code> as the Executive in the top-level Assembly. The instance is named “solver” by default.</td>
</tr>
</tbody>
</table>

Each time an NPSS execution ends, an exit status is returned. If any errors or exceptions occurred during execution, the status will be set to −1; otherwise, it will be 0.

## 2.2 General NPSS Input Syntax

### 2.2.1 Introduction to the Input

NPSS is usually executed with at least one input file. NPSS input files are plain ASCII files that can be prepared with standard text editors such as `vi` or `xemacs`. The input file or files define the engine model, the calculations that are to be performed, and any output that is to be generated. This section describes the syntax used to create these files. Later chapters (such as Chapter 4 and Chapter 5) will show more specifically how this syntax is used to define and run a simulation.

NPSS input is actually a programming language. Readers familiar with C++ will notice many similarities between that language and the language of NPSS. No prior knowledge of C++ is necessary, however, to understand this or any other chapter in the *User Guide*.

### 2.2.2 Some General Features of the Input

Five general features affect all NPSS input:

- object orientation
- case sensitivity
- statement terminators
- blocks
- comments

#### 2.2.2.1 Object Orientation

As mentioned in Section 1.3, object orientation means that the program's structure revolves around objects of various types that have attributes accessible by other objects and functions that can be performed at the request of other objects. An object may be a simple entity, such as a variable, or a complex entity, such as a solver. An object *type* is a pattern or blueprint defining the attributes and functions that all objects of that type will have.
The NPSS program has been written by defining various object types, and creating specific objects of those types so that they can interact with each other. Likewise, the user builds a model in NPSS and controls what it does by creating specific instances of NPSS object types, and causing them to interact with one another and with other NPSS objects created for the user. Creating a specific example or instance of an object type is called instantiation.

In addition to instantiating objects of predefined object types, and using objects instantiated automatically, the user can also utilize the NPSS language to define new object types having whatever attributes and functions are useful. Objects can be instantiated from user-defined types just as objects of predefined types can be. Whenever an object is instantiated, it is given a unique name by which it is referenced thereafter.

The attributes and functions of an object are generally accessed using "dot" notation. For example, suppose thrust is an instance of an NPSS variable (an NPSS object type), and therefore has an attribute (as will be discussed in Section 2.2.4.1) called units. One could set the units attribute of variable thrust by using the following notation:

```
thrust.units = "lbf";
```

Similarly, suppose ncpView is an object instantiated from the NPSS DataViewer type. As discussed in Section 4.9.2, NPSS DataViewer objects have a function called display(). To instruct object ncpView to execute its display() function, use the following notation:

```
ncpView.display();
```

NPSS objects are hierarchical: it is common for one object to be contained inside another, which may be contained inside another, and so on. "Dot" notation is again used to specify the path to a specific object. Suppose CmpH is an object instantiated from an NPSS Compressor type. The NPSS Compressor type is an example of a kind of NPSS types called elements. Elements can contain subelements. The Compressor element in particular can contain a subelement object of type CompressorRlineMap. In our example, suppose the CmpH element contains a CompressorRlineMap subelement named S_map. An object of type CompressorRlineMap automatically contains a variable named RlineMapDes that specifies the design Rline of the unscaled compressor map. One could set the value of this variable using the following notation:

```
CmpH.S_map.RlineMapDes = 2.0;
```

Another NPSS Compressor element named CmpL might also contain a CompressorRlineMap subelement named S_map which itself also contains a variable named RlineMapDes. This variable would be set using the following notation:

```
CmpL.S_map.RlineMapDes = 1.8;
```

Both subelements and both variables have the same name, but they are not confused with each other because they are in different places in the NPSS object hierarchy (they have different paths). As will be discussed later, it is not always necessary to specify the complete path to an NPSS object. It is important to understand at this point, however, the hierarchical nature of NPSS objects.

The simplest path is a dot placed before the name of an object to indicate an object directly under the top-level assembly (see Section 1.5.3). As will be discussed further in Section 2.2.4.3, this is often called global scope.

An object that contains another object can be referred to as that object's parent. In the first example above, Compressor element CmpH is the parent of CompressorRlineMap subelement S_map. Similarly, subelement S_map could be described as the child of Compressor element CmpH. Subelement S_map is also the parent of variable RlineMapDes, and variable RlineMapDes is the child of subelement S_map.

### 2.2.2.2 Case Sensitivity

All NPSS input is case sensitive. This means that name1 and Name1 refer to two distinct objects. This gives the user more flexibility in naming objects. It also means, however, that the user must carefully note the case of each letter in predefined NPSS objects and commands.

To simplify matters for the user, NPSS commands, keywords, and predefined variables follow certain rules regarding case:
• Basic commands (such as if and else), basic data types (such as int and real), and other keywords (such as class and extends) are in lower case.
• Complex data types (such as Assembly and Element) begin with a capital letter.
• If a name is a concatenation of multiple words, all words after the first begin with a capital letter. An example is Viewer, which begins with a capital because it is a complex data type. Most predefined variable names begin with a lower case letter, such as eRamBase.

2.2.2.3 Statement Terminators
NPSS input is composed of statements: syntactical units that instruct NPSS to instantiate an object, access an attribute of an existing object, or execute a function in an existing object. In general, each statement must end with the NPSS statement terminator which is the semi-colon (;). Thus,

\[ A = B + C \]

would not be recognized as an NPSS statement, whereas

\[ A = B + C; \]

would be. Although it is customary to place one statement per line, the user must remember that the line feed/carriage return does not terminate an NPSS statement. (Interactive mode provides an exception to this rule – see Section 2.3.1.1.) In NPSS input files, the semi-colon is necessary. The semi-colon as a statement terminator allows the user to put two or more statements on a single line. Placing one statement per line, however, usually makes for input that is easier to understand and more pleasing to the eye. The user also has the freedom to break a single long statement into two or more lines in order to make it easier to read.

2.2.2.4 Blocks
A block is a set of statements that are grouped together as a unit. Certain features of the NPSS language, such as conditional branching, require blocks. The statements constituting a block are always enclosed within curly braces: { }. Curly braces must always be used in pairs; a common error is to open a block with a left "curly" but fail to close it with a right "curly." As the examples in the User Guide will show, consistent use of indentation can help the user keep curly braces properly matched in pairs.

In many respects, a block is treated as a single, compound statement. Syntactically, however, a block does not need to be terminated by a semi-colon; the closing curly brace is sufficient. Individual statements within the block, however, do require the semi-colon to terminate them. For example:

```npss
Element Inlet Inl {
  eRamBase = 0.995;
} // END Inl
```

The first line instantiates an object of Element type Inlet, which is named Inl (see Section 2.2.2.1 for more information on instantiating elements). The left curly brace opens a block. It is not the end of a statement, so no semi-colon appears. A single statement appears within the block, assigning the value of 0.995 to the variable eRamBase. This statement must be terminated by a semi-colon as shown. The statement is indented simply to remind the reader that it is part of a block. The block is closed by a right curly brace, after which no semi-colon is necessary. The closing "curly" is placed on its own line with the same indentation as the line that opened the block to assist the reader in identifying the extent of the block. This practice also helps the user remember the closing brace. The text "// END Inl" is an optional comment, as discussed in the next section (2.2.2.5).

When objects are instantiated, it is always possible to place a block (enlosed in curly braces) after the object name to immediately define items pertaining to the object. This block, such as the one in the example above, is called an object's instantiation block. After an object's instantiation, another block can be associated with it by giving the object's name followed by the block (again, enlosed in curly braces). For example:

```npss
Inl {
  eRamAud = 0.0;
  s_eRamAud = 1.0;
}
```
The examples above simply assign values to variables instantiated for the user by NPSS. It is also possible to instantiate objects such as variables (Section 2.2.4), functions (Section 2.2.6), tables (Section 2.2.7), and subelements (Section 4.4) within a block associated with an object. Objects instantiated within a block associated with another object become the children of that object. The importance of this will be more clearly seen as the hierarchy of objects in an NPSS model is discussed further (especially, for example, Sections 2.2.4.3, 2.2.6.2, and 0 through 4.6).

2.2.2.5 Comments

Every bit of text in an NPSS input file is not necessarily processed as an instruction by NPSS. The user can denote certain text as a comment, in which case, it is ignored by NPSS. Two devices are available for marking text as a comment. Any text following a pair of slashes, //, to the end of the line is treated as a comment. For example:

```
// Component design operating characteristics
Amb.W = 47.8194;   // CmpH WcDes = 124 lbm/sec
```

As in the first line above, if the two slashes are the first non-blank characters on the line, the entire line is a comment. As in the second line, comment text can also be added at the end of a line containing input to be interpreted by NPSS.

Another way to mark a segment of input as a comment is to enclose it between these delimiters: /* ... */. For example:

```
/*
Element Compressor CmpL {
    #include "CmpL.map"
} //END CmpL
*/
```

This "comments out" the entire object instantiation. This is a useful technique for removing items from the model that might be desired in a later run. Alternatively, double slashes could have been added at the start of each line. As shown in the example, the two comment methods can be combined. Multi-line comments delimited by /* ... */ can also be nested (unlike C and C++).

The user is encouraged to liberally introduce comments in NPSS input files to provide visual markers separating major sections of the input and to document important features of the input.

2.2.3 Preprocessor Commands

Before the instructions in the input files are processed by NPSS, they are scanned by a preprocessor. The user can insert commands to the preprocessor in the input files. Preprocessor commands always begin with the pound sign: #. These commands allow the user to include other files, define preprocessor variables, and perform conditional preprocessing. A preprocessor command can appear anywhere on a line, but must be the only thing on the line except for comment text. A preprocessor command need not be terminated with a semi-colon.

2.2.3.1 Including Other Files

#include "filename" or #include <filename> instructs the preprocessor to insert the entire contents of the named file in place of the #include command. For example:

```
Element Compressor CmpH {
    #include "hpc.map"
} //END CmpH
```

In the example, the "map" defining the operating characteristics of compressor CmpH is located in a separate file named hpc.map. Its contents are included within the block instantiating this compressor object just as if they had been typed there. By using #include commands, the user can divide the NPSS input among several files in a logical fashion.

If the file name appears between quotation marks, as above, the preprocessor first searches the current directory for the file. (Of course, filename could be a path to a file located in another directory.) If the file is not found, the
Preprocessor searches the NPSS include path (combination of NPSS_PATH and –I command line args). If the file name appears between angle brackets, <>, the preprocessor performs the same search but in the opposite sequence, searching the NPSS include path first and then the current directory. For example:

```
#include <ncp.view>  //default page viewer
```

This facilitates including standard pieces of input located in "library" directories, which are made part of the include path. See Section 2.1 for information on using the command line –I option and the NPSS_PATH environment variable to specify the include path.

If the file to include does not appear in the NPSS include path, then the file's location may be specified as part of the `#include` command. For example, if the file to be included resides in a different directory than the current directory, then the file may be loaded using the `#include` command as long as either the relative or absolute path is specified. The following examples show a variety of ways to include a file named "file2.int" that resides in a subdirectory (of the current directory) named "SubDir":

```
#include "SubDir/file2.int"
#include "SubDir/file2.int"
#include "SubDir/file2.int"
#include <SubDir/file2.int>
#include <SubDir/file2.int>
#include <SubDir/file2.int>
```

These statements work with NPSS on Windows:

```
#include "SubDir/file2.int"
#include "SubDir/file2.int"
#include <SubDir/file2.int>
#include <SubDir/file2.int>
#include <SubDir/file2.int>
```

These statements work with NPSS on Linux:

```
#include "SubDir/file2.int"
#include "SubDir/file2.int"
#include <SubDir/file2.int>
#include <SubDir/file2.int>
```

The following statement will fail on Windows because the argument is read in as a string, and the single backslash (\) will be treated as an escape character:

```
#include "SubDir\file2.int"  // will fail
```

A given file can be included any number of times by other NPSS input files. However, users should avoid inadvertently including a file more than once. A particularly bad error is causing a file to include itself. For example:

```
file_1:  #include "file_2"
file_2:  #include "file_3"
file_3:  #include "file_1"
```

This "self-inclusion" of a file is currently undetected in NPSS and will result in a crash once memory is exhausted. Conditional preprocessing using preprocessor variables can avoid this error. These are the topics of the next two sections. (Prevention of multiple inclusion is specifically discussed in Section 2.2.3.3.)

The contents of one file can also be processed from within another file by using the `parseFile()` function (see Section 12.2.4 for details). Preprocessor commands cause the contents of a file to be inserted before parsing begins. The file's contents are then parsed when encountered by the parser as it processes the entire input stream. The `parseFile()` function, on the other hand, parses the contents of the specified file immediately as it is read. The `parseFile()` function can be useful in interactive mode (discussed in Section 2.3) in which the user suspends normal NPSS input stream processing. Using `parseFile()` in interactive mode, the user can force immediate processing of a file's contents.
2.2.3.2 Defining Preprocessor Variables

#define variable_name value defines a preprocessor variable. The value is optional. There must not be any comment text on the same line as a #define statement (it will be taken as part of the variable's value). After its definition, prefixing the preprocessor variable name with a dollar sign, $, directs the preprocessor to substitute the variable's value, if there is one, in place of $variable_name. For example:

    #define THERMO Janaf
    ...
    setThermoPackage("$THERMO");

The last line is equivalent to setThermoPackage("Janaf");. (See Section 4.1 for more information on thermodynamics packages.) Assigning a value to a preprocessor variable is useful if that value appears several places in the input files. To change the value, the user need only change the definition of the preprocessor variable. The preprocessor takes care of placing the new value wherever it is referenced in the input files. Another use is to assign critical model values to preprocessor variables and collect the definition of these variables together in a convenient place (such as at the top of one of the first input files processed). The user can then change these critical values in a single location without having to find the places they are used in the input files.

As mentioned in Section 2.1, the values of preprocessor variables can also be defined on the command line using the -D option. Any preprocessor variable defined on the command line takes precedence in the following two ways over those defined using #define in an input file:

- Any attempt to #define a variable with a different value than that set on the command line will be ignored. The value set on the command line cannot be changed.
- Any attempt to #undef a variable (see below) defined on the command line will be ignored. Once defined on the command line, always defined.

These features can be used to force certain behavior on a given run of NPSS, overriding preprocessor variable definitions in the input files. In the example given above, an NPSS input file uses preprocessor variable THERMO to specify Janaf as the thermodynamics package. The value of preprocessor variable THERMO can be overridden, however, on the command line as follows:

    npss -D THERMO=GasTbl file1 file2 ...

This provides a convenient way to run the same model with any thermodynamics package without having to change any of the model's files.

Preprocessor variables can also be "undefined" as follows:

    #undef variable_name

As far as the preprocessor is concerned, variable_name no longer exits. The principal reason to "undefine" a preprocessor variable is in certain cases where it is used in a conditional preprocessor statement (discussed in the following section).

2.2.3.3 Conditional Preprocessing

Four conditional preprocessing commands are available:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#ifdef variable_name</td>
<td>(&quot;if defined&quot;) True if variable_name is defined.</td>
</tr>
<tr>
<td>#ifndef variable</td>
<td>(&quot;if not defined&quot;) True if variable_name is not defined.</td>
</tr>
<tr>
<td>#else</td>
<td>Used in logical constructs.</td>
</tr>
<tr>
<td>#endif</td>
<td>Used to end a logical construct.</td>
</tr>
</tbody>
</table>
#ifdef and #ifndef test only for the definition of a preprocessor variable (i.e. whether or not it exists), not if it has a value associated with it. For example:

```c
#ifdef PPVAR
  ...
#else
  ...
#endif
```

A #endif must always follow a #ifdef or a #ifndef. The #else section is optional. A #else can follow either a #ifdef or a #ifndef.

One use for such conditionals is to prevent a file from being included more than once. For example, an object type named EngPerf might be defined in a file named EngPerf.cmp. The definition of the object type in file EngPerf.cmp should be bracketed by conditional preprocessor commands as follows:

```c
#ifndef ENGPERF_CMP
#define ENGPERF_CMP
  ...
#endif
```

After the object type has first been defined in the model through a #include "EngPerf.cmp" statement, preprocessor variable ENGPERF_CMP is defined. The object will not be erroneously defined twice even if #include "EngPerf.cmp" appears elsewhere in the model.

### 2.2.4 Variables

The most basic objects in the NPSS language are variables. Their primary purpose is to hold data values–either numbers or text. Variables may be integers, real numbers, or text strings, and they may be either scalars or arrays. All in all, there are ten types of NPSS variables. These variable types and the data values they hold are:
### Table 3. Variable Types

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>scalar integer value</td>
</tr>
<tr>
<td>int[]</td>
<td>1D array of integer values</td>
</tr>
<tr>
<td>int[][]</td>
<td>2D array of integer values</td>
</tr>
<tr>
<td>real</td>
<td>scalar real (&quot;floating point&quot;) value</td>
</tr>
<tr>
<td>real[]</td>
<td>1D array of real values</td>
</tr>
<tr>
<td>real[][]</td>
<td>2D array of real values</td>
</tr>
<tr>
<td>real[][][]</td>
<td>3D array of real values</td>
</tr>
<tr>
<td>string</td>
<td>scalar text string</td>
</tr>
<tr>
<td>string[]</td>
<td>1D array of text strings</td>
</tr>
<tr>
<td>string[][]</td>
<td>2D array of text strings</td>
</tr>
<tr>
<td>Option</td>
<td>one of a set of allowable text strings</td>
</tr>
<tr>
<td>FunctVariable</td>
<td>references to a <code>getFunction</code> and <code>setFunction</code> (see Section 2.2.4.8)</td>
</tr>
<tr>
<td>Matrix</td>
<td>a matrix of real values</td>
</tr>
</tbody>
</table>

Ordinary scalar variable types (`int`, `real`, and `string`) are discussed in Section 2.2.4.4. Array types (`int[]`, `int[][]`, `real[]`, `real[][]`, `real[][][]`, and `string[]`) are discussed in Section 2.2.4.5. Some special features of string scalars and string arrays are covered in Section 2.2.4.6. Option variables are discussed in Section 2.2.4.7. Section 2.2.4.8 covers FunctVariables.

There will be occasional references later in this Guide to *Boolean integers*. These are not a separate NPSS variable type, but simply a way to interpret a variable of type `int`. When an integer variable has a value of zero, and is interpreted as a Boolean, it is interpreted to mean "false." When an integer, interpreted as a Boolean, has a nonzero value (either positive or negative), it is taken to indicate "true." Boolean integers are usually given the value "1" to indicate "true."

#### 2.2.4.1 Variable Attributes

Different types of variables have different properties and capabilities; however, all variable types allow the user to define the attributes shown in Table 4. Only the variable's `value` is required; setting the other attributes is optional.
### Table 4. Variable Attributes

<table>
<thead>
<tr>
<th>Variable Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The variable's data value. The variable's type (Table 3) determines the kind of information its value attribute holds. The default is zero or a null string. See the following information.</td>
</tr>
<tr>
<td>description</td>
<td>A short text string describing the variable's meaning (i.e., why it exists). The default is a null string. See the following information.</td>
</tr>
<tr>
<td>iDescription</td>
<td>A short text string describing the variable's value (i.e., why it has the value it does). The default is a null string. See the following information.</td>
</tr>
<tr>
<td>units</td>
<td>A text string identifying the variable's units. The units attribute can be set for any NPSS variable type; however, units are meaningful only for real variables. The default is a null string. See the following information.</td>
</tr>
<tr>
<td>unitsRule</td>
<td>Has two settings: &quot;CONFORM&quot;, &quot;DISPLAY&quot;. The default is: &quot;CONFORM&quot;. CONFORM - must conform to the units system as shown in displayConvTables(); DISPLAY - does not conform to units system and can be whatever.</td>
</tr>
<tr>
<td>IOstatus</td>
<td>A text string indicating the variable's current input/output status. Allowable values are given and discussed in the information that follows.</td>
</tr>
<tr>
<td>trigger</td>
<td>TRUE or FALSE, indicating whether a change in the variable is registered with its parent component. This is discussed further in information that follows.</td>
</tr>
</tbody>
</table>

**value**

A variable's value is set by default when the assignment operator (=) is used, as in

```plaintext
CmpH.PRdes = 25.0;
```

This is equivalent to `CmpH.PRdes.value = 25.0;`.

Variables of type real hold floating point numbers; variables of type int hold integers. Integer values can be assigned to real variables. If a real value is assigned to an int variable, the int variable will be assigned only the integral part of the real number (i.e. the truncation of the real number). The real number is not rounded.

Values of string variables must be enclosed in *double quotation marks*, as in

```plaintext
Brn.switchBurn = "TEMPERATURE";
```

Numerical values cannot be directly assigned to string variables, nor string values to variables of types real and int. Functions are provided, however, to accomplish these conversions (see Section 12.8.1).

**description and iDescription**

These attributes receive string values. For variables, attribute description is intended to explain why the variable exists. This is particularly useful for new variables created by the user, but can also explain the particular purpose to which a standard NPSS variable is applied. Attribute iDescription is intended to explain why the variable has been given its particular value. For example:

```plaintext
CmpH.effDes.description = "Development goal adiabatic efficiency";
CmpH.effDes.iDescription = "Value from rig test of similar machine";
```

Attributes description and iDescription also exist for more complex objects such as elements and subelements. For such objects, description is intended to explain the general purpose of the object type, and is usually given a predefined default value by the element writer. Attribute iDescription is intended to explain the purpose of a specific instance of a type. It can be supplied by the user to help document a model.
units
The units attribute receives a string value, as in the following example:

```java
thrust.units = "lbf";
```

NPSS recognizes certain units strings given in Chapter 11. These can be used to affect units conversions as discussed in Section 2.2.5.3.

unitsRule
The unitsRule attribute selects to either enforce valid units assignments (unitsRule = "CONFORM"), or to allow non-sense units for documentation purposes (unitsRule = "DISPLAY"). The default setting is to "CONFORM".

```java
real Pgage;  // guage pressure
Pgauge.unitsRule = "DISPLAY"; // allow units that do not conform
Pgauge.units = "psig"; // assign units that do not conform,
                  // i.e. not listed in displayConvTables()
```

IOstatus
The IOstatus attribute is a string value that the user may access but will normally not set when using NPSS components. The IOstatus attribute identifies the role of the variable in the current operating mode of the object that contains the variable. NPSS contains certain predefined Option variables that control the mode of operation of certain objects. (Option variables are discussed in more detail in Section 2.2.4.7.) For example, several element and subelement types include an Option variable named switchDes. When switchDes = "DESIGN", the object calculates scale factors to make its output match specified design values. The variables representing the design values are inputs to the object, and the scale factors are outputs. When switchDes = "OFFDESIGN", the object assumes the scale factors are already defined (they are now inputs), and it calculates its other output variables using them. The variables representing design values may be inactive in this case. The IOstatus attribute can take on the values shown in the following table.
Table 5. IOstatus Values

<table>
<thead>
<tr>
<th>IOstatus Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>The variable is an input to its parent component.</td>
</tr>
<tr>
<td>output</td>
<td>The variable is an output from its parent component.</td>
</tr>
<tr>
<td>const</td>
<td>The variable's value cannot be changed; it is a constant. See note below.</td>
</tr>
<tr>
<td>inactive</td>
<td>The variable is not used in the current operating mode.</td>
</tr>
<tr>
<td>unset</td>
<td>Either the variable's role has not been determined, or it is not calculated in the current operating mode. Most likely the variable is an output in some other operating mode. A variable's default IOstatus is unset.</td>
</tr>
<tr>
<td>readonly</td>
<td>The variable is an output from its parent component and can only be read, not modified.</td>
</tr>
</tbody>
</table>

If a variable is to be a constant, all its other attributes should be set before its IOstatus attribute is set to const. Once a variable's IOstatus attribute equals const, nothing about the variable can be changed. Even its IOstatus cannot be reassigned to another value.

**trigger**

The trigger attribute is an integer value interpreted as a Boolean (0 is false, nonzero is true). For clarity, NPSS provides named constant TRUE, which is a substitute for integer 1, and FALSE, which is a substitute for integer 0. If a variable's trigger attribute is nonzero (TRUE), then whenever the variable's value is set, the variable calls the variableChanged() function of its parent component. The function in the parent component can be written to take certain actions whenever critical variables within it change. See the Developer Guide for more information. Most variables have their trigger attribute equal to FALSE by default. Option variables (see Section 2.2.4.7) have their trigger attribute equal to TRUE by default.

### 2.2.4.2 Variable Naming

As with all NPSS objects, a variable must be given a name, and this name must be unique within the scope of the variable (see Section 2.2.4.3). Follow these rules when naming any NPSS object, including variables:

- Use only alphabetic characters, digits, and the underscore character, _. Names cannot include spaces, dots (periods), hyphens, or any other punctuation marks or special characters.
- Do not begin the name with a digit. It can begin with an alphabetic character or the underscore character.
- Make the name meaningful—there is no limit on name length.
- Remember that the name is case sensitive.
- Do not use an NPSS keyword (summarized below).

NPSS *keywords*, which cannot be used as object names, are as follows.
2.2.4.3 Variable Scope

A variable’s scope is that part of the input where the variable can be referenced by name only, without a path. The scoping rules for variables in NPSS are as follows:

- If the user creates a variable, the scope of that variable is limited to the block in which it is defined and any blocks or objects contained within that block.
- Similarly, variables created by NPSS for the user are scoped to the object to which they belong and to that object’s children.

This means that if a variable name appears without a path attached to it, NPSS first looks for the definition of that variable within the block in which the name appears, or within the object where the name is referenced. If it fails to find the variable there, it looks in the parent block or object, and so forth up the hierarchy until global scope is reached (the scope of the top-level assembly – see Section 1.5.3). If the variable cannot be found there, an error is reported. To access variables in the child of an object, a path must be given. That path, however, need not be an absolute path (beginning with objects in the top-level assembly). The path can begin with objects under that in which the variable is being referenced. Consider the following example.

```
real effGoal = 0.832;
CmpH {
    effDes = effGoal;
    real stallLine = 1.0;
    S_map.RlineStall = stallLine;
}
```

The first line instantiates or declares a new variable of type real named effGoal. The next lines constitute a block defining variables belonging to object CmpH which was instantiated earlier as an object of type Compressor (not evident from the code fragment given). Object CmpH already contains a variable named effDes and a subelement named S_map, which in turn contains a variable named RlineStall. To reference the effDes variable from within the block belonging to CmpH, the variable path need not be given because effDes belongs to the CmpH object. However, the variable effGoal does not exist within the block in which it is referenced, so NPSS seeks it
(and finds it) in the next level up and effDes is assigned the value of effGoal. Within the CmpH block, another new variable is declared, stallLine, and assigned a value so that it can in turn be used to set the value of the S_map object’s variable RlineStall. To reference RlineStall from within the CmpH block, a path must be given. The path does not, however, need to be CmpH.S_map.RlineStall because paths inside a block belonging to an object need only begin with children of that object. Consequently, the path is simply S_map.RlineStall, and this variable is assigned the value of stallLine. The variable stallLine exists within the block in which it is referenced (CmpH), so NPSS finds it at the current scope.

As mentioned in Section 2.2.2.1, the simplest path is a dot placed before the name of an object to indicate that it belongs directly to the top-level assembly (is in global scope). A variable without a path is sought in successively higher level scopes until global scope is reached. But if a variable by the same name as that desired in global scope exists in one of the intervening scopes, it will be found and used first. In cases where the user must be certain that the global variable is used, prepending a dot as a path insures safety. In the following example, element CmpH contains a subelement named S_map. Separate variables each named here are placed in global scope and in the scope of each object.

```plaintext
int here = 1;
CmpH {
  int here = 2;
  S_map {
    int here = 3;
    int a = here; // a = 3
    int c = .here; // c = 1
    int d = parent.here; // d = 2
  }
}
```

The assignment of here to "a" uses the variable named here in the local scope of the assignment. The reference to .here, however, refers to the variable bearing that name in global scope. Thus "c" is assigned the value 1. The variable named here belonging to object CmpH can only be reached from the scope of object S_map by using the path CmpH.here.

**The extern keyword**

Use the extern keyword to specify either the type of an out-of-scope variable or the return type of an out-of-scope function. When interpreted component files are converted, out-of-scope variables/functions are attributed to be from a parent, and no type is assumed. This can cause operational ambiguities in the C++ code that’s created by the converter. If, however, a user makes the variables/functions they are using external, then the variable type may be given and conversion problems will be reduced. For example:

```plaintext
extern int x;    //when x used, evaluate as an int
extern string y;   //when y used, evaluate as a string
extern int testFunct();  //call to testFunct() will evaluate to an int.
```

Extern is used only as a way to specify type ahead of time, so users may not attempt to initialize an extern. The following are examples that won’t work with extern:

```plaintext
extern real x { some init block;}    //must not attempt to init.
extern int x = 5;      //must not attempt to init
extern string testFunct() { return "Hi";}    //don't try to def. Funct body
```

See the Developer's Guide, Chapter 8, for information about the NPSS to C++ Converter and its -ext flag.

**2.2.4.4 Scalars**

A variable is instantiated (or declared) by first giving its type, then giving its name, then (optionally) initializing the variable with a value. A simple assignment to the variable name always sets the variable’s value attribute. Other attributes of the variable must be referenced explicitly. Some examples follow.
real var1;                 // single declaration without initialization
real var2, var3;           // multiple declaration without initialization
int counter = 2;           // declaration with value initialization
string label_1 = "xyz";

var3.IOstatus = "output";  // subsequent attribute assignment

real duct_length {         // declaration with full initialization
  value        = 10.3;
  units        = "in";
  description  = "Length of transition duct";
  iDescription = "From print 1234567-89, rev. B";
  IOstatus     = "input";
  trigger      = TRUE;
}

In the final example, notice that curly braces inclose a block (see Section 2.2.2.4) associated with variable duct_length, and that no semicolon is required after the closing curly brace of the block. A semicolon is required, however, after each statement in the block.

### 2.2.4.5 Arrays

As given in Table 3, the allowable array types in NPSS are:

- real (1D, 2D, and 3D)
- int (1D and 2D)
- string (1D and 2D)

A one-dimensional int array called ia can be created and initialized as follows:

```
int ia[] = { 1, 2, 3 };
```

In this table variable array types were indicated by placing a pair of square brackets after the variable type. This was only shorthand, however. When arrays are instantiated, the fact that the variable as an array is indicated by placing one or more pairs of square brackets after the variable name as in the example above. Array values are supplied between curly braces. Since three initial entries were supplied, the initial size of this array is three. Note that a semicolon is required after the closing curly brace. The initial array values do not constitute a block of statements. The initialization is a simple statement, and must be terminated with a semicolon as all simple statements must.

Entries of an array are addressed using an index number. The index number of the first entry in an array is zero, not one, as shown in the following example.

```
int i;
  i = ia[0];  // i = 1
  i = ia[1];  // i = 2
  i = ia[2];  // i = 3
  i = ia[3];  // Won't work! ia[] currently has only 3 entries.
```

Alternatively, the array could be created with an initial size, in which case it is automatically initialized to all zeros:

```
real ra1[3];
```

An array must either be given a set of initial entries or an initial size. The following example produces an error:

```
real ra2[];
ra2[0] = 1;  // Won't work!
```

An NPSS array will change size when reinitialized.

```
string sa[] = { "A", "B", "C" };  // sa[] has 3 entries.
```
NPSS array objects have some useful member functions. Calling a member function using dot notation was mentioned in Section 2.2.2.1. Functions are discussed in detail in Section 2.2.6.

To simply add entries to an array, use the append() function.

```
sa.append( "H" ); // sa[] now equals { "D", "E", "F", "G", "H" }
```

To find the number of entries currently in an array, use the entries() function, which returns an integer.

```
int e;
e = sa.entries(); // e = 5
```

Other variable attributes, such as description and units, pertain to the entire array. They cannot be set or accessed through individual entries.

```
real ra3[] = { 1.1, 2.2, 3.3 };  
ra3.units = "ft";
```

If one tried ra3[0].units = "ft"; an error would result.

As mentioned previously, real arrays can be two- or three-dimensional. A 2D array is a 1D array of 1D arrays.

```
real ra5[[]] = { { 0.0, 0.1, 0.2 }, { 1.0, 1.1 } };
```

Notice that the subarrays can differ in size. If the initial size of a multidimensional array is specified, its subarrays are initialized to the same size:

```
real ra6[2][3];
```

Array ra6 can be thought of as a 1D array of two 1D arrays, each of which contain three elements. Since no initial values were given, all the elements (2x3=6 total) are automatically initialized to zero.

Indices are evaluated from left to right with the leftmost index indicating the position in the topmost 1D array. Indices to the right indicate positions in the subarrays. Thus, ra5[0][1] corresponds to the first 1D array (index 0 indicates the first position), and the second entry of that array (1 indicates the second position). In the example above, ra5[0][1] equals 0.1. If the second index were omitted, the reference would be to an entire array:

```
real ra7[] = ra5[1]; // Array ra7 now equals { 1.0, 1.1 }
```

Three-dimensional arrays are handled similarly. A 3D array is a 1D array of 2D arrays, each of which is itself a 1D array of 1D arrays.

```
real ra8[][][] = { { { 000, 001 }, { 010, 011 } },  
{ { 100, 101 }, { 110, 111 } },  
{ { 200, 201 }, { 210, 211 } } };
```

Notice again that a semicolon is required to terminate the array initialization statement.

When array assignments are made separate from the array declaration, the terminating semicolon is not required.

Examples of array assignments where an ending semicolon is not required:

```
string myarray[]; // declare string array
myarray = {"joe", "john"} // ";" not required

Option switchType {
    allowedValues = { "CON_DIV=0", "CONIC=1" } // ";" not required
}

Element foo;
solverSequence = { "foo" } // ";" not required
```
Example of array assignment where an ending semicolon is required:

```
string myarray2[] = { "tom", "mark" }; // Needs ";" because array assignment
// is part of the array declaration
```

Function `append()` can be used to add arrays to multidimensional arrays. The function only appends to the top-level 1D array. Thus, when appending to a 3D array, `append()` requires a 2D array as its argument. When appending to a 2D array, `append()` requires a 1D array. These arrays must have been assigned to separate variable objects.

```
real app2[][] = { { 300, 301 }, { 310, 311 } }; // This works.
ra8.append( { { 300, 301 }, { 310, 311 } } ); // This fails.
```

As mentioned above function `append()` will only add an entity to the topmost 1D array.

```
real app1[] = { 120, 121, 122 }; // This fails.
ra8[1].append( app1 );
```

For convenience, nD array `append()` can also take an nD argument. This is treated as a sequence of normal `append()` calls with an (n-1)D argument.

The `entries()` function returns only the number of subarrays contained in the topmost 1D array. After the successful append illustrated earlier, `ra8.entries()` equals 4 (ra8[] contains four 2D subarrays).

**Range to Array Assignment**

A range of values may be assigned to an array. A range of values is a one-dimensional array containing a set of values the user would like to assign to an array starting at a predefined array index. The index may contain no more than one operator, i.e., `a[x + y]`. Using more than one operator `(+, -, *, /)` will result in an error. To use the returned integer value from a function as an index, the user will need to assign the returned value to an integer variable such as:

```
int myIndex = MyFunct();
```

where `MyFunct()` returns an integer value. The index would then look like this:

```
a[myIndex]
```

A range of strings and integers can be assigned to one-dimensional arrays only. Reals can be assigned to one-, two-, and three-dimensional arrays.

Examples of the syntax for range assignments are as follow:

**Strings:**

```
string str[10];
str[0] = {"one", "two", "three"};
returns: str = {"one", "two", "three", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "`,
```

**Integers:**

```
int MyInt1D[] = {0,1,2,3,4,5,6,7};
MyInt1D[2] = {9,8,7};
returns: MyInt1D = {0,1,9,8,7,5,6,7}
```

```
two dimensions:
int MyInt2D[][]={0,1,2},{3,4,5});
```
MyInt2D[0][1]=(5,3,1);
returns MyInt2D = {{0,5,3},{1,4,5}}

**Reals:**

*one dimension:*

real MyReal1D[] = {0., 1., 4.2, 5.0, 6.5};
MyReal1D[0]=(1., 2., 3.);
returns: MyReal1D = (1, 2, 3, 5, 6.5)
MyReal1D[1]=(2.2, 4.4);
returns: MyReal1D = (1, 2.2, 4.4, 5, 6.5)

*two dimensions:*

real MyReal2D[][] = {{0.,1.0,2.0},{3.0,4.0,5.0}};
MyReal2D[0][1]=(5.0,3.0,1.0);
returns MyReal2D = {{0.,5.0,3.0},{1.0,4.0,5.0}}

*three dimensions:*

real MyReal3D[][][] = {{1.,2.,3.},{4.,5.,6.},
{7.,8.,9.},{10.,11.,12.}},
{13.,14.,15.},{16.,17.,18.}};
MyReal3D[0][1][0]=(15., 14., 13., 12., 11., 10., 9., 8., 7., 6., 5., 4.);
returns MyReal3D = {{1.,2.,3.},{15.,14.,13.},
{12.,11.,10.},{9.,8.,7.}},
{6.,5.,4.},{16.,17.,18.}}

### 2.2.4.6 String Variables

A few special points pertain to string variables. Strings are always enclosed in double quotes for assignment to string variables.

```
string s = "This is a string.";
```

Sometimes strings contain commands or expressions (discussed in Section 2.2.5.2). Sometimes these expressions must themselves contain quotes. A common example is expressions using the cout object for writing information to the screen:

```
cout << "s = " << s << endl;
```

Briefly, however, the characters "<<" are used to send information to cout which displays it on the screen. If a string is sent, cout displays the string. If a number is sent, cout displays it with an appropriate format. If an array is sent (Section 2.2.4.5), cout displays the entire array in an appropriate format. In the example above, string "s = " is first sent, followed by string variable s itself, followed by predefined global variable endl (see Section 2.2.4.10) which produces a carriage return and line feed. The result is:

```
s = This is a string.
```

If the cout command given above needed to be stored as a string, the quotation marks that are part of the command must be separated from the quotation marks that enclose the string. This is done by escaping the quotation marks within the string by preceding them with a backslash (\"").

```
string command = "cout << \"s = \" << s << endl;";
```

Scalar string variables have a special member function named length() that returns the number of characters in the string. For example, s.length() returns the integer 17. The individual characters of a scalar string variable can be accessed with notation resembling that used for arrays. Each character is numbered, left to right beginning with zero, and is accessed by giving the string variable name followed by the character number enclosed in square brackets. For example, s[1] holds the character "h". Other functions for the manipulation of scalar strings are discussed in Section 12.8.
String arrays have five special member functions: contains(), insertAt(), insertByName(), replaceByName(), and remove() (see Section 12.1.2). Suppose variable sa is a string array defined as follows:

\[
\text{sa} = \{ \text{"A"}, \text{"B"}, \text{"D"}, \text{"13"}, \text{"E"} \};
\]

You can test if a string is contained in the array by:

\[
\text{int foundD} = \text{sa}.\text{contains(} \text{"D"} \text{)};
\]

A new string can be inserted to occupy index location 2 (the third element of the array) as follows:

\[
\text{sa}.\text{insertAt(} 2, \text{"C"} \text{)};
\]

String sa now equals \{ "A", "B", "C", "D", "13", "E" \}. Notice that no entries were lost, only shifted. Entries can be removed as follows:

\[
\text{sa}.\text{remove(} \text{"13"} \text{)};
\]

The user need not know the position in the array of the entry to be removed. Function remove() finds the entry, if it exists, removes it, and shifts the remaining entries as needed. After the command above, string sa equals \{ "A", "B", "C", "D", "E" \}.

Two-dimensional string arrays can be created in the following manner:

\[
\text{string sa2[][]} = \{ \{ \text{"A"}, \text{"B"} \}, \{ \text{"C"}, \text{"D"} \} \}
\]

When working with two-dimensional string arrays, remember that each entry in the array is a one-dimensional string array, and the operations on these arrays take that into account. So, when using the two-dimensional array above, one could, for instance, append a regular string array (as opposed to a single string) as follows:

\[
\text{string sa[]} = \{ \text{"E"}, \text{"F"} \};
\text{sa2.append(sa)};
\]

That would change sa2 to be \{ \{ "A", "B" \}, \{ "C", "D" \}, \{ "E", "F" \} \}. The other functions for two-dimensional string arrays tend to follow accordingly.

### 2.2.4.7 Option Variables

An option variable is a scalar string variable that can take on only certain values. For example, many NPSS objects of type Element contain an option variable named switchDes that has two allowable values: "DESIGN" and "OFFDESIGN".

The user can create new option variables using the Option keyword, as in the following example:

\[
\text{Option compressor_name;}\
\text{compressor_name.allowedValues} = \{ \text{"CmpFSec"}, \text{"CmpL"}, \text{"CmpH"} \};
\]

The allowable values for an option variable are defined by setting its allowedValues attribute. It is good practice to set this attribute immediately after declaring the variable. Once created, Option variables can be set just like ordinary string variables:

\[
\text{switchDes} = \text{"DESIGN"};\
\text{compressor_name} = \text{"CmpL"};
\]

If the user attempts to set the variable to a value that is not listed in its allowedValues attribute, an error results. However, The user may append a new value to the allowedValues list by using the append function:

\[
\text{compressor_name.append} = \{ \text{"BypassFan"} \};
\]

As noted in Table 4, all NPSS variables have an attribute named trigger that for option variables is set to TRUE by default. This means that changes to the variable's value generate a call to function variableChanged() in the variable's parent object. The variableChanged() functions built into NPSS are written to recognize changes to built-in NPSS option variables such as switchDes. These functions do not recognize new user-created option
variables such as compressor_name in the example above. Therefore changes to such variables will generate a non-fatal error that the variableChanged() function did not know how to handle the variable's change. To avoid these messages, set trigger to FALSE:

```c
compressor_name.trigger = FALSE;
```

**Integer Associations**

Each allowable string value is also associated with an integer value. By default, the allowable strings are associated from left to right with consecutive integers beginning with zero. Thus, in the example above, "CmpFSec" is associated with 0, "CmpL" with 1, and "CmpH" with 2.

The integer value associated with the current string value of an option variable is held in attribute `intValue`. Thus, in the example above, the assignment of string value "CmpL" to compressor_name caused `compressor_name.intValue` to take on the value 1. Setting attribute `intValue` also sets the string value of an option variable:

```c
compressor_name.intValue = 0;  // Sets compressor_name to "CmpFSec"
```

Thus integer associations provide a second way for the user to set the value of an option variable. An error results if the user attempts to assign `intValue` a value not associated with one of the allowable string values.

When an option variable is created, the user may assign integer values other than the defaults to the allowable string values of the variable. This is illustrated in the following example:

```c
Option days;
days.allowedValues = { "TU=3", "SU=1", "MO=2.6", "WE", "TH", "FR", "SA" };  
```

Any integer values can be assigned – they need neither be consecutive nor ascending. If a real value is provided, it is truncated to an integer ("MO" is associated with integer 2). If integer values are not explicitly assigned to some string values, consecutive default integers are assigned, left to right, beginning with one greater than the largest integer explicitly assigned. Thus "WE" is associated with 4, "TH" with 5, "FR" with 6, and "SA" with 7.

Note that the option variable itself is a string. Thus printing variable days will print a string, not an integer. The value of days can only be compared with operators `==` or `!=` (see Section 2.2.5.1) to another string, not to an integer.

**allowedStrings and allowedInts**

In addition to `allowedValues` and `intValue`, option variables have two other special attributes accessible to the user. Each of these is an array, and can only be read by the user (not written to).

<table>
<thead>
<tr>
<th><code>allowedStrings</code></th>
<th>String values to which the option variable may be set.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>allowedInts</code></td>
<td>Integer values associated with the allowable string values (and to which attribute <code>intValue</code> may be set).</td>
</tr>
</tbody>
</table>

In the example above, `days.allowedStrings = { "TU", "SU", "MO", "WE", "TH", "FR", "SA" }` and `days.allowedInts = { 3, 1, 2, 4, 5, 6, 7 }`.

**rewritableValues**
The rewritableValues attribute controls whether the allowedValues attribute may be rewritten. By default this attribute is TRUE. If set FALSE, the allowedValues attribute is constant. This will enable a converter optimization when comparing option variables with constant strings (the converter is a tool used to translate from NPSS syntax to C++, see the Developer’s Guide). Once set FALSE, you cannot reset the rewritableValues attribute back to TRUE.

Function setOption()

NPSS provides the function setOption() to set all option variables with a given name in and below the scope in which the setOption() function is called. The function can be called in the following two ways:

```plaintext
setOption( "option_variable_name", "value" );
setOption( "option_variable_name", intValue);
```

If setting the option variable's name to a string, the desired value must appear in quotes, as shown in the first line above.

Calling setOption() instructs NPSS to look throughout the scope from which setOption() was called for an option variable with the name specified. If one exists, NPSS will set its value as specified, assuming that the specification is one of the variable's allowable settings. NPSS will then look in all the scopes contained in the scope from which setOption() was called for other option variables with the name specified. All such variables found will also be set as specified. This process continues recursively through all scopes below that in which setOption() was originally called. For example, if the following appeared in global scope:

```plaintext
setOption( "switchDes", "OFFDESIGN" );
```

then all switchDes option variables in the entire NPSS model would be set to "OFFDESIGN.”

The setOption() function is a member function of all elements and subelements, and most other NPSS object types (see Section 12.1.1). Therefore, if CmpH is a Compressor object, then

```plaintext
CmpH.setOption("switchDes", "OFFDESIGN");
```

would set the switchDes option variable in just the CmpH object and its children to “OFFDESIGN”.

NPSS functions are discussed in more detail in Section 2.2.6.

2.2.4.8 FunctVariables

Note: This section requires knowledge of NPSS programming constructs (Section 2.2.5) and functions (Section 2.2.6). Upon first reading, the reader may wish to skip this section and return to it after covering the sections just mentioned. This section is located here, with the discussion of other variable types, to facilitate easier reference by experienced users.

A FunctVariable is used when other actions should always accompany the getting or setting of the variable's value. A FunctVariable has special attributes that associate it with two functions: a setFunction and a getFunction. The setFunction assigns the value of a variable associated with the FunctVariable, and can perform other actions as well (called side effects). The getFunction returns the value of a variable associated with the FunctVariable, and can also produce other side effects. A FunctVariable that lacks a setFunction is said to be read-only.

The user may create new FunctVariables using the FunctVariable keyword, as in the following example:

```plaintext
FunctVariable fv {
    setFunction = "fvSet";
    getFunction = "fvGet";
}
```

Notice that setFunction and getFunction are attributes of the FunctVariable. The functions themselves have the names specified by the attributes. If a value is assigned to attribute setFunction or getFunction, the user must implement the named function, and the function must be accessible from the scope in which the FunctVariable is
referenced. These functions may be implemented before or after the instantiation of the FunctVariable, but before it is referenced.

The setFunctions and getFunctions should normally be written according to the following rules:

- The setFunction should not return a value, and must receive exactly one argument representing the value to be returned by the getFunction, or from which that value can be calculated.
- The setFunction should assign a value to an existing variable accessible from the scope in which the FunctVariable is to be referenced.
- The getFunction must receive no arguments, and should return the value of the variable set by the setFunction.

An example of these functions appears below, using the definition of FunctVariable \( fv \) given above.

```c
real fvVal = 0.;
int fvAccess = 0;

void fvSet( real newval ) {
    fvAccess =fvAccess + 1;
    if ( newval >= 0. ) {
        fvVal = newval;
    } else {
        fvVal = 0.;
    }
}

real fvGet() {
    fvAccess = fvAccess + 1;
    return fvVal;
}
```

The setFunction above (\( \text{fvSet}() \)) insures that the value associated with FunctVariable \( fv \) is non-negative.

Writing setFunctions to insure valid values is one use of FunctVariables. In the example above, a side effect of the setFunction and getFunction is counting the number of times the FunctVariable is accessed.

A FunctVariable is accessed with syntax like that of an ordinary variable. For example:

```c
fv = 13.7;
```

NPSS effectively replaces this statement with:

```c
fvSet(13.7);
```

Likewise, the statement,

```c
var1 = fv;
```

is effectively replaced with:

```c
var1 = fvGet();
```

Note that the FunctVariable itself does not hold the value assigned to it. The setFunction assigns values to other variables, the getFunction accesses those variables. References to the FunctVariable simply trigger calls to those functions. Accessing the \text{value} attribute of a FunctVariable, as in

```c
var1 = fv.value;
fv.value = var1;
```

simply results in calls to the getFunction and the setFunction.

The value supplied to the setFunction is stored in attribute \text{setValue} of the FunctVariable. Thus in the example,

```c
fv = -10.2;
```
variable `var1` will have the value 0 (because of the way function `fvSet()` was written), but `fv.setValue` will have the value -10.2. Recall that `fv = -10.2` is equivalent to `fvSet(-10.2)`. The value supplied to `fvSet()` is copied into `fv.setValue` before `fvSet()` is executed. The user cannot directly assign a value to the `setValue` attribute.

**Note:** In the special case on the Windows platform, where a user defines a large number of FunctVariables and variables, the compiler may fail due to a stack overflow. If this occurs, the user will need to add the `/Zm` compiler option to the compiler command line. This option will increase the size of the memory used by the Windows compiler. The allowed values to the `/Zm` option are as follows: `/Zm10 = 5.0MB, /Zm100 = 50MB (default), /Zm200 = 100MB, /Zm1000 = 500MB, and /Zm2000 = 1000MB.

### 2.2.4.9 Matrix Variables

Matrix is another NPSS variable type. A Matrix can be declared in any of the following ways:

```plaintext
Matrix A, B;
```

real[] and real[][] expressions may be assigned to a Matrix directly without explicit conversion, for example:

```plaintext
Matrix A = {{1,2},{3,4}};
```

Another option for declaring a Matrix is:

```plaintext
Matrix A
{
    value = {{1,2},{3,4}};
    description = "a matrix!";
    units = "in";
}
```

A Matrix may also be the return type of a function or a function argument, for example:

```plaintext
Matrix matMult(Matrix A, Matrix B)
{
    return A*B;
}
```

The following standard operations are supported:

- `A*B`     // multiply
- `A+B`     // add
- `A-B`     // subtract
- `n*A` or `A*n`  // scalar multiply
- `A/n`     // scalar divide
- `A[1][2]`  // member access

**Matrix Functions**

- `Matrix A.transpose()`
  Returns a Matrix that is the transpose of `A`.

- `real A.determinant()`
  Returns the determinant of `A`. 

Matrix A.inverse()
Returns a Matrix that is the inverse of A.

Matrix A.eigenvalues()
Returns a diagonal matrix with eigenvalues along diagonal.

Matrix A.eigenvectors()
Returns a matrix with eigenvectors as column vectors.

real[] A.toR1D()
Returns a real[] representation of A (if A is a row or column vector).

real[][] A.toR2D()
Returns a real [][] representation of A (if A is a row or column vector).

Matrix A.getRow(int)
Returns a row vector (a Matrix) of the given row.

Matrix A.getCol(int)
Returns a column vector (a Matrix) of the given column.

string A.format()
Writes the matrix with fixed-width columns to a string, for example:

```
cout << A.format();
1.000000 2.000000
3.000000 4.000000
```

### 2.2.4.10 Automatically Created Global Variables

When launched, NPSS automatically creates the following variables, which exist in the top-level assembly (i.e., they have global scope).

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTHOR</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Available to hold the author's name.</td>
</tr>
<tr>
<td>background</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Available to document the top-level assembly. Not normally used.</td>
</tr>
<tr>
<td>baseType</td>
<td>string</td>
<td>&quot;Element&quot;</td>
<td>Identifies the base object type of the top-level assembly. Not normally used.</td>
</tr>
<tr>
<td>byteSwapIO</td>
<td>int</td>
<td>0</td>
<td>The default value used for the byteSwap attribute of a stream when created. Only affects binary files.</td>
</tr>
<tr>
<td>CASE</td>
<td>int</td>
<td>0</td>
<td>Available for use as a numerical case identifier. CASE is not automatically incremented; the user must set its new value at the beginning of each new case. See Section 4.10.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Initial Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cpuTime</td>
<td>real</td>
<td>(run dependent)</td>
<td>User and system CPU time accumulated by the process (seconds).</td>
</tr>
<tr>
<td>date</td>
<td>string</td>
<td>(run dependent)</td>
<td>The current date in format MM/DD/YY.</td>
</tr>
<tr>
<td>DCLODPATHS</td>
<td>string array</td>
<td>(run dependent)</td>
<td>Directories searched by DCLOD (Section 3.3).</td>
</tr>
<tr>
<td>defaultHideLevel</td>
<td>int</td>
<td>0</td>
<td>Sets the initial hide level for all new objects. It eliminates the need to call the hide() function explicitly on a large number of objects to hide them.</td>
</tr>
<tr>
<td>description</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Available to describe the top-level assembly. Not normally used.</td>
</tr>
<tr>
<td>E</td>
<td>real</td>
<td>2.71828</td>
<td>Base of natural logarithms (to approximately 20 digits).</td>
</tr>
<tr>
<td>endl</td>
<td>string</td>
<td>&quot;\n&quot;</td>
<td>Line feed string. Commonly used when producing output with cout or cerr. See Section 2.2.4.6.</td>
</tr>
<tr>
<td>FALSE</td>
<td>int</td>
<td>0</td>
<td>Used to make logical tests and Boolean arguments clearer.</td>
</tr>
<tr>
<td>ICLODPATHS</td>
<td>string array</td>
<td>(run dependent)</td>
<td>Directories searched by ICLOD (Section 3.1).</td>
</tr>
<tr>
<td>iDescription</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Available to describe the top-level assembly. Not normally used.</td>
</tr>
<tr>
<td>INCLUDEPATHS</td>
<td>string array</td>
<td>(run dependent)</td>
<td>Directories searched by the preprocessor, and, by default, Creation Method Facilities. See Sections 2.1, 2.2.3.1, 3.1, and 3.3.</td>
</tr>
<tr>
<td>inputFileList</td>
<td>string array</td>
<td>(run dependent)</td>
<td>List of all input files, in the order they were loaded.</td>
</tr>
<tr>
<td>MODELNAME</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Available to hold the model name.</td>
</tr>
<tr>
<td>NaN</td>
<td>real</td>
<td>NaN</td>
<td>Can be used to indicate the result of an invalid mathematical operation (&quot;Not a Number&quot;).</td>
</tr>
<tr>
<td>NPSSTerminationSequence</td>
<td>string array</td>
<td>{}</td>
<td>List of objects executed in order after NPSS exits. Items placed in this sequence would be for cleanup or for generating output. Functions must be no argument functions.</td>
</tr>
<tr>
<td>PI</td>
<td>real</td>
<td>3.14159</td>
<td>Ratio of a circle's circumference to its diameter (to approximately 20 digits).</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Initial Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>postsolverSequence</td>
<td>string array</td>
<td>{}</td>
<td>List of objects executed in order after the solver exits. Object names are commonly placed here by the user to cause them to be executed once after the solver converges. This is actually an alias to the current Executive’s postExecutionSequence, kept for backward compatibility.</td>
</tr>
<tr>
<td>presolverSequence</td>
<td>string array</td>
<td>{}</td>
<td>List of objects executed in order before the solver assumes control of execution. This is actually an alias to the current Executive’s preExecutionSequence, kept for backward compatibility.</td>
</tr>
<tr>
<td>stringSorting</td>
<td>string</td>
<td>“IgnoreCase”</td>
<td>Type of comparison used during sorting for operations like list(). Valid values are “Exact”, “IgnoreCase”, “AlphaNumeric”, and “AlphaNumericIgnoreCase”</td>
</tr>
<tr>
<td>solutionMode</td>
<td>string</td>
<td>&quot;ONE_PASS&quot;</td>
<td>Identifies the mode of operation of the top-level assembly. This variable is not very important since the mode of operation of the solver is determined by its local solutionMode attribute (e.g., solver.solutionMode). See Chapters 6 and 6. Allowable values are &quot;STEADY_STATE,&quot; &quot;TRANSIENT,&quot; and &quot;ONE_PASS.&quot; The initial mode for solver objects is &quot;STEADY_STATE.&quot; The value of this variable will determine which Executive object is used, either the one referenced by steadyStateExecutiveName or transientExecutiveName.</td>
</tr>
<tr>
<td>solverSequence</td>
<td>string array</td>
<td>{}</td>
<td>List of objects executed in order during each solver pass. This is actually an alias to the current Executive’s executionSequence, kept for backward compatibility.</td>
</tr>
<tr>
<td>switchDes</td>
<td>string</td>
<td>&quot;DESIGN&quot;</td>
<td>Option variable determining the mode of operation of the top-level assembly (but not necessarily the objects within it). Allowable values are &quot;DESIGN&quot; and &quot;OFFDESIGN&quot;. This switch is normally set for all the objects in a model using function setOption() called from global scope. See Section 4.8.1.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>switchCreatePermission</td>
<td>string</td>
<td>&quot;AUTO&quot;</td>
<td>Option variable determining what classes the user can instantiate in a customer deck. &quot;OFF&quot; means it is ok to instantiate any class (predefined and customer defined). &quot;CHECK&quot; means the user can only instantiate classes that have been granted create-permission for. &quot;AUTO&quot; means the user can instantiate classes that have been granted create-permission for and customer defined classes.</td>
</tr>
<tr>
<td>THERMPACKAGE</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>The currently selected thermodynamics package. This attribute should not be set directly by the user. The thermodynamics package is set using command setThermoPackage() (see Section 4.1).</td>
</tr>
<tr>
<td>time</td>
<td>real</td>
<td>0</td>
<td>The time in seconds during a transient run. See Chapter 6.</td>
</tr>
<tr>
<td>timeOfDay</td>
<td>string</td>
<td>(run dependent)</td>
<td>The current wall clock time (24 hour) in format HH:MM:SS.</td>
</tr>
<tr>
<td>timeStep</td>
<td>real</td>
<td>0.05</td>
<td>The current time step in seconds used in a transient run. See Chapter 6.</td>
</tr>
<tr>
<td>title</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Available to hold a title for the model.</td>
</tr>
<tr>
<td>TRUE</td>
<td>int</td>
<td>1</td>
<td>Used to make logical tests and Boolean arguments clearer.</td>
</tr>
<tr>
<td>UsageNotes</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Available to document the top-level assembly. Not normally used.</td>
</tr>
<tr>
<td>USER</td>
<td>string</td>
<td>(run dependent)</td>
<td>The user's UNIX full name, or login name, if no full name is defined.</td>
</tr>
<tr>
<td>VERSION</td>
<td>string</td>
<td>(run dependent)</td>
<td>The program version.</td>
</tr>
<tr>
<td>wallTime</td>
<td>real</td>
<td>(run dependent)</td>
<td>Time since some epoch (typically system boot time) with sub-second resolution.</td>
</tr>
</tbody>
</table>

### 2.2.5 Programming Constructs

The NPSS language provides operators with which the user can combine the values of variables and constants into a single value in constructs called expressions. Keywords also exist which allow construction of conditional branches and loops.
2.2.5.1 Operators

The following table gives the NPSS operators in the order in which they are executed. The "precedence" column indicates this order. If two operators appear in the same statement, the operator with the lower "precedence" value will be executed first. If two operators with equal precedence appear in the same statement, they are executed left to right. Unless otherwise stated, the operands are of type int or real.

<table>
<thead>
<tr>
<th>Symb</th>
<th>Name</th>
<th>Example</th>
<th>Prec</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-&gt;</td>
<td>indirect membership</td>
<td>A-&gt;B</td>
<td>0</td>
<td>B is an attribute, object, or function that belongs to the object whose name is given by the value of string A. See Section 2.2.6.5.</td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
<td>-A</td>
<td>1</td>
<td>Returns the value with the opposite sign of A. The value of A is not changed.</td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td>!A</td>
<td>1</td>
<td>= 0 if A is nonzero. = 1 if A is zero.</td>
</tr>
<tr>
<td>++</td>
<td>increment prefix</td>
<td>++A</td>
<td>1</td>
<td>First increases the value of A by 1, then returns this new value.</td>
</tr>
<tr>
<td>--</td>
<td>decrement prefix</td>
<td>--A</td>
<td>1</td>
<td>First decreases the value of A by 1, then returns this new value.</td>
</tr>
<tr>
<td>+</td>
<td>increment suffix</td>
<td>A++</td>
<td>2</td>
<td>First returns the value of A, then increases the value of A by 1.</td>
</tr>
<tr>
<td>-</td>
<td>decrement suffix</td>
<td>A--</td>
<td>2</td>
<td>First returns the value of A, then decreases the value of A by 1.</td>
</tr>
<tr>
<td>~</td>
<td>unary bitwise negation</td>
<td>~A</td>
<td>2</td>
<td>a.k.a. ones-compliment. The result is an integer in which all 1s in A are changed to 0 and vice versa.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>bitwise right shift</td>
<td>A :&gt; b</td>
<td>3</td>
<td>Returns an integer in which all bits in A are shift right by b places. Equivalent to A / (2**b), truncated (not rounded) to an integer.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bitwise left shift</td>
<td>A &lt;: b</td>
<td>3</td>
<td>Returns an integer in which all bits in A are shift left by b places. Equivalent to A * (2**b).</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>A &amp; B</td>
<td>3</td>
<td>For each bit, result is 1 if both operand bits are 1, 0 otherwise.</td>
</tr>
<tr>
<td>^</td>
<td>bitwise exclusive OR</td>
<td>A ^ B</td>
<td>3</td>
<td>For each bit, result is 1 if the one or the other operand bit is 1, but 0 if both bits are 1 or 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>A**B</td>
<td>4</td>
<td>Raises A to the B power. If B is real and has a fractional part, A must be non-negative. (C++ users will notice that this operator is an addition to the C++ operator set.)</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>A*B</td>
<td>5</td>
<td>Multiplies A and B.</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>A/B</td>
<td>5</td>
<td>Divides A by B. B must be nonzero. If both A and B are type int, a type int is returned by truncating any fractional part of the quotient.</td>
</tr>
<tr>
<td>Symb</td>
<td>Name</td>
<td>Example</td>
<td>Prec</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-----------------</td>
<td>----------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>A%B</td>
<td>5</td>
<td>Evaluates to the remainder of dividing A by B. Both A and B must be type int.</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>A+B</td>
<td>6</td>
<td>Adds A and B. If A and B are both type string, the result is the concatenation of A and B.</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>A-B</td>
<td>6</td>
<td>Subtracts B from A.</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>A&lt;B</td>
<td>8</td>
<td>Evaluates to 1 (TRUE) if A is less than B. Evaluates to 0 (FALSE) otherwise.</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>A&gt;B</td>
<td>8</td>
<td>Evaluates to 1 (TRUE) if A is greater than B. Evaluates to 0 (FALSE) otherwise.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>A&lt;=B</td>
<td>8</td>
<td>Evaluates to 1 (TRUE) if A is less than or equal to B. Evaluates to 0 (FALSE) otherwise.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>A&gt;=B</td>
<td>8</td>
<td>Evaluates to 1 (TRUE) if A is greater than or equal to B. Evaluates to 0 (FALSE) otherwise.</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
<td>A==B</td>
<td>9</td>
<td>Evaluates to 1 (TRUE) if A is equal to B. Evaluates to 0 (FALSE) otherwise. Can be used to compare strings as well as reals and integers.</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td>A!=B</td>
<td>9</td>
<td>Evaluates to 1 (TRUE) if A is not equal to B. Evaluates to 0 (FALSE) otherwise. Can be used to compare strings as well as reals and integers.</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>A&lt;B &amp;&amp; B&lt;C</td>
<td>13</td>
<td>Evaluates to 1 (TRUE) if both expressions evaluate to nonzero (TRUE). Evaluates to 0 (FALSE) otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical OR</td>
<td>A&lt;B</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
<td>A=B</td>
<td>14</td>
<td>Assigns the value of B to A.</td>
</tr>
<tr>
<td>+=</td>
<td>add and assign</td>
<td>A+=B</td>
<td>14</td>
<td>Equivalent to A = A + B. Just like the + operator, this operator performs a concatenation if A and B are both of type string.</td>
</tr>
<tr>
<td>-=</td>
<td>subtract and assign</td>
<td>A-=B</td>
<td>14</td>
<td>Equivalent to A = A - B</td>
</tr>
<tr>
<td>*=</td>
<td>multiply and assign</td>
<td>A*=B</td>
<td>14</td>
<td>Equivalent to A = A * B</td>
</tr>
<tr>
<td>/=</td>
<td>divide and assign</td>
<td>A/=B</td>
<td>14</td>
<td>Equivalent to A = A / B</td>
</tr>
<tr>
<td>%=</td>
<td>modulus and assign</td>
<td>A%=B</td>
<td>14</td>
<td>Equivalent to A = A % B</td>
</tr>
<tr>
<td>&amp;=</td>
<td>bitwise AND and assign</td>
<td>A&amp;=B</td>
<td>14</td>
<td>Equivalent to A = A &amp; B</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>bitwise OR and assign</td>
<td>A</td>
<td>=B</td>
</tr>
<tr>
<td>^=</td>
<td>bitwise XOR and assign</td>
<td>A^=B</td>
<td>14</td>
<td>Equivalent to A = A ^ B</td>
</tr>
</tbody>
</table>

As with other programming languages, parentheses can be used to give an operation higher precedence than it would otherwise have. For example:

```
var1 = 5 + 3 * 2;  // var1 equals 11
var1 = (5 + 3) * 2; // var1 equals 16
```
Use of these operators is illustrated in much of the material in the following sections. The indirect membership operator (\(\rightarrow\)) is discussed in more detail in Section 2.2.6.5.

These operators are evaluated in pairs from left to right, such that in the expression \(A \& B | C\), \(A \& B\) is evaluated first and the result ORed with \(C\). The operands are assumed to be INTEGER values.

### 2.2.5.2 Expressions

**Expressions** are formed by applying operators to variables or constants. An expression evaluates to a single value and can appear in a statement anywhere a variable or constant can appear. In the following example, \(\sin()\) is a function (Section 2.2.6) that requires a single \texttt{real} value within its parentheses representing an angle in radians. Variable \texttt{var2} represents an angle in degrees. Variable \texttt{PI} is an automatically created global variable (Table 7).

\[
\texttt{var1} = 2. * \sin(\texttt{var2} * \texttt{PI} / 180.);
\]

### 2.2.5.3 Expressions with Units

A string (enclosed in quotation marks) can follow an expression to indicate the units of the expression result. NPSS can perform some units conversions for the user. For example:

```plaintext
real \texttt{inletP} = 14.696;
real \texttt{dPoP} = 0.05;
real \texttt{exitP_SI} { \texttt{units} = "kpa"; }       // kilopascals
\texttt{exitP_SI} = \texttt{inletP} * (1-\texttt{dPoP}) "psia";  // pounds per square inch, absolute
```

In the example above, the result of multiplying \texttt{inletP} by the quantity \(1-\texttt{dPoP}\) is a value known to the user to have units of pounds per square inch. The user indicates this to NPSS by adding the string "psia" after the expression. The expression result is to be assigned to the variable \texttt{exitP_SI}. NPSS checks to see if the \texttt{units} attribute of \texttt{exitP_SI} is defined. In this case it is, and is set to "kpa". NPSS compares this unit to the indicated units of the expression. If they match, the simple assignment is made. If they do not match, as in the case above, NPSS checks to see if it knows how to convert "psia" into "kpa". Since these units are compatible, NPSS will perform the conversion so that the value stored in variable \texttt{exitP_SI} really does have units of "kpa". Had the units of the expression and the units of the assignment variable been incompatible, an error would have been issued.

A list of available units conversions can be displayed using the following function:

```plaintext
displayConvTables();
```

NPSS units are case sensitive. What this means is that "FT" is not a valid unit of measurement, but "ft" is valid. BTU and btu are not valid units but "Btu" is. Unitless parameters can be declared with their \texttt{units} attribute equal to "none". This prevents dimensional quantities being assigned to them. Valid units strings are given in Chapter 11.

Note that NPSS does not do units arithmetic. In the following example,

```plaintext
real \texttt{force} { \texttt{value} = 1000; \texttt{units} = "lbf"; }  // pounds force
real \texttt{area} { \texttt{value} = 10; \texttt{units} = "in2"; }  // square inches
real \texttt{pressure};
\texttt{pressure} = \texttt{force} / \texttt{area};
```

NPSS does not know that the units of the expression \texttt{force} / \texttt{area} are pounds per square inch. It must be told:

```plaintext
\texttt{pressure} = \texttt{force} / \texttt{area} "psia";
```

The specified units of the expression are \texttt{not} automatically copied into the \texttt{units} attribute of the variable on the left-hand side of the equals sign. In the example above, \texttt{pressure.units} is undefined.

Even in very simple cases, no units conversion will be performed unless NPSS is explicitly told the units of the right-hand-side of an assignment statement:

```plaintext
real \texttt{P_US} {\texttt{value} = 14.696; \texttt{units} = "psia"; }
```
real P_SI { units = "kpa"; }
P_SI = P_US; // no units conversion; the numerical value in P_SI is not in units of "kpa".
P_SI = P_US "psia"; // units conversion is performed.

An alternative method would be to use empty quotes to specify the default units associated with that variable:

real P_US {value = 14.696; units = "psia"; }
real P_SI { units = "kpa"; }  
P_SI = P_US ""; // the numerical value of P_SI is in units of "kpa"

### 2.2.5.4 Conditional Branches

A _conditional branch_ is a construct in which a test is made and a different set of statements executed depending on the outcome of the test. NPSS provides two keywords for constructing conditional branches: _if_ and _else_. An _if_ construct has the following form:

```plaintext
if ( conditional_expression ) {
    body_statements;
}
```

NPSS conditional expressions evaluate to numerical results that are interpreted as _Boolean values_ (see Section 2.2.4). A variable or expression whose value is nonzero (either positive or negative) is interpreted as "true." A variable or expression whose value is zero is interpreted as "false." Normally Boolean values are integers, but in this case real expressions are interpreted the same way. If the _conditional expression_ above evaluates to nonzero ("true"), the _body statements_ are executed. Any number of body statements can be given; they are identified by being grouped together in a block (enclosed within a pair of curly braces). The curly braces must be included even if there is only one body statement. If the conditional expression evaluates to zero ("false"), the body statements are skipped, and execution continues with whatever statement follows the body statement block.

The following examples are based on solving the quadratic equation in the form

\[ A*(\text{ans}^2) + B*\text{ans} + C = 0 \]

with A, B, C, and ans being NPSS real variables, and A being nonzero. An integer _soln_ will be used to indicate the number of unique real solutions (0, 1, or 2).

A simple _if_ construct can avoid the error due to a negative square root.

```plaintext
real ans1, ans2, det;  
int soln;  
det = B**2 - 4*A*C;  
if ( det > 0 ) {
    ans1 = ( -B + sqrt(det) ) / (2*A);  
    ans2 = ( -B - sqrt(det) ) / (2*A);  
    soln = 2;
}
```

Function _sqrt_() is a built-in NPSS math function. (See Section 12.7.)

The _else_ keyword can be used to provide a block of statements to be executed if the original conditional expression evaluates to 0 (FALSE). If the expression is _TRUE_ the _if_ block is executed; if it is _FALSE_ the _else_ block is executed.

```plaintext
if ( det > 0 ) {
    ans1 = ( -B + sqrt(det) ) / (2*A);  
    ans2 = ( -B - sqrt(det) ) / (2*A);  
    soln = 2;
}  
else {
    soln = 0;
}
```
The else keyword can also be followed by another if block.

```plaintext
if ( det > 0 ) {
    ans1 = ( -B + sqrt(det) ) / (2*A);
    ans2 = ( -B - sqrt(det) ) / (2*A);
    soln = 2;
} else if ( det == 0 ) {
    ans1 = -B / (2*A);
    ans2 = ans1;
    soln = 1;
} else {
    soln = 0;
}
```

As many `else if` constructs can be used as are necessary to handle the relevant cases. The final `else` block is optional. If included, it handles any cases not covered by the preceding `if` tests.

It is permissible to nest `if - else if - else` constructs inside others.

```plaintext
if ( det >= 0 ) {
    ans1 = ( -B + sqrt(det) ) / (2*A);
    if ( det > 0 ) {
        ans2 = ( -B - sqrt(det) ) / (2*A);
        soln = 2;
    } else {
        ans2 = ans1;
        soln = 1;
    }
} else {
    soln = 0;
}
```

The conditional expression can be a compound set of expressions using the `&&` and `||` operators, or it can be as simple as a test on the value of a single variable.

```plaintext
if ( !det ) {
    soln = 1;
}
```

### 2.2.5.5 Class Declarations and Instances in Conditional Statement Blocks

NPSS does not allow for class declarations in a conditional statement block. For example, the following:

```plaintext
int ynOPT = 1;
if (ynOPT == 1) {
    class fooo extends Element { } // attempt to declare a new class
}
```

would generate an error because a class is being declared within the conditional statement block.

However, unlike C/C++ NPSS does allow you to declare an instance of a component in a conditional statement block as long as the class is first declared outside of the block. Therefore, you can accomplish the same task in this manner:

```plaintext
class fooo extends Element { } // declare a new element class
```
int OPT = 1;
if (OPT == 1) {
    fooo MyFoo; // declare an instance of a class
}

2.2.5.6 Loops

A loop is a construct in which one or more statements are executed repeatedly until some terminating condition is met. Each repetition is called an iteration. Five keywords in the NPSS language allow the construction of loops: for, while, do, break, and continue.

The for loop is the most general and has the following form:

```
for ( initialization_statement; conditional_expression; increment_statement ) {
    body_statements;
}
```

The initialization statement is generally used to set a loop counter to an initial value. This statement is executed only once. The conditional expression is generally used to compare the loop counter to a limiting value. It is evaluated every iteration. If the expression evaluates to nonzero ("true"), the loop continues by first executing the body statements and then by executing the increment statement. If the expression evaluates to zero ("false"), the loop terminates and execution resumes with whatever statement follows the body statement block. The increment statement is generally used to increase or decrease the loop counter initialized by the initialization statement. The increment statement is executed once per iteration, and is the last thing executed before the conditional expression is evaluated again. The body statements constitute a block (enclosed in curly braces) and are executed once per iteration. In the following example, the model’s altitude (in feet) is set by assigning a value to variable alt belonging to element AMB (as in Section 5.2.2.1). Function run() executes the model.

```
real altitude[] = { 0, 10000, 20000, 30000 }; // Array of desired altitudes.
int ialt;
for ( ialt = 0; ialt < altitude.entries(); ialt++ ) {
    Amb.alt = altitude[ialt];   // Set the model’s ambient condition to each altitude in the array.
    CASE++;                       // Run the model at each specified altitude.
    run();                       // Run the model at each specified altitude.
}
```

Note that in a loop designed to access all the elements of an array, the array index ranges from 0 to \( I \) less than the number of entries (in the example, \( ialt \) ranges from 0 to 3 for an array with 4 entries). See Section 2.2.4.5 for details on array indexing.

C++ users should note that, unlike C++, variable declarations cannot be made in the initialization statement. Loop counters must be declared outside the loop as in the example above.

The variable used in the initialization statement, conditional expression, and increment statement need not be merely a counter. The same result accomplished by the loop above could be done as follows:

```
for ( Amb.alt = 0; Amb.alt <= 30001; Amb.alt+=10000 ) {
    CASE++;                   // Run the model at each specified altitude.
    run();
}
```

Variable AMB.alt is a real number, not an integer. Care should be exercised when indexing a real value since real arithmetic is not as precise as integer arithmetic. The conditional expression in the example above is written to allow for accumulated round-off error in the real variable.
The initialization statement and increment statement can be made into *compound statements* by separating individual statements with *commas*. In this way, a single for loop can change two or more variables at once. In the following example, the model's Mach number is set using variable AMB.MN.

```plaintext
for ( Amb.alt = 0,      Amb.MN = 0;
     Amb.alt <= 30001 && Amb.MN <= 0.61;
     Amb.alt+=10000,   Amb.MN+=0.20 ) {
     CASE++; run();
}
```

The line breaks in the example above are purely to improve readability.

The *for* statement is very flexible. It is not necessary for the same variable or variables to appear in the initialization statement, conditional expression, and increment statement. Nor is it necessary for the conditional expression to be the comparison of two or more values; it could be an arithmetic expression, or even a single variable. A successful loop results whenever each iteration changes something that eventually causes the conditional expression to evaluate to 0 ("false"). For more complicated examples, see texts on C or C++ programming.

A *while* loop is a simpler version of the *for* loop. Its syntax includes only the conditional expression. The user must handle initialization outside the loop and handle incrementing within the body statements. For example:

```plaintext
Amb.alt = 0;
while ( Amb.alt <= 30001 ) {
     CASE++; run();
     Amb.alt += 10000;
}
```

A loop constructed with the *do* statement, often called a "do-while" loop, is just like a while loop except the conditional expression comes last rather than first. This means the body statements are always executed at least once.

```plaintext
Amb.alt = 0;
do {
     CASE++; run();
     Amb.alt += 10000;
} while ( Amb.alt <= 30001 );
```

The final *while* statement in a do-while loop is a single statement and can appear on its own line. However, placing it on the same line as the close of the *do* block, as in the example above, improves readability by avoiding the appearance of the *while* statement initiating a while loop rather than completing a do-while loop.

The *break* statement causes execution to jump out of any of these loop constructs. In the following example, the ambient temperature in degrees Rankine is stored in variable AMB.Ts.

```plaintext
for ( Amb.alt = 0; Amb.alt <= 30001; Amb.alt+=10000 ) {
     CASE++; run();
     if ( Amb.Ts < 450. ) {
         break;
     }
}
```

Since the ambient temperature at 20000 ft altitude is less than 450°R, the loop terminates before the 30000 ft altitude condition is run.

The *continue* statement causes execution to skip the remainder of the body statements and start the next iteration.
for ( Amb.alt = 0; Amb.alt <= 30000; Amb.alt+=10000 ) {
    if ( Amb.alt >= 19999 && Amb.alt <= 20001 ) {
        continue;
    }
    CASE++;
    run();
}

This logic runs the model at altitudes of 0 and 10000, skips the run() statement when altitude is 20000, but continues the loop to run the model at an altitude of 30000. Again, notice allowance for imprecise indexing of real values.

2.2.6 Functions

As mentioned in Section 1.5.1, the user may define functions in the input file. An NPSS function accepts zero or more arguments, whose types must be specified, and returns a value, whose type must be also be specified. In addition to the types discussed in Section 2.2.4, two additional possibilities are available for use with functions:

- A function argument can be specified as type any, in which case, any of the allowable variable types is accepted.
- The function can be specified to return type void, in which case no value is returned.

Option variables cannot be returned by functions. An Option variable can be used as an argument to a function but is declared and treated as an ordinary string variable by the function.

The statements that make up the function (the function body) constitute a block (Section 2.2.2.4) and are therefore enclosed in curly braces, {}. The value returned by the function is specified by a return statement. Some examples follow.

```c
real Tstd(int alt) {  // This function is named Tstd, returns a
    // real value, and accepts a single int argument.
    real T;
    T = 518.67 - 0.0035662 * alt;
    return T;
}

real[] extract(real arr[][], int nsub) {
    // This function is named extract,
    // returns a single-dimensioned real array,
    // and accepts two arguments: the first is a double-
    // dimensioned real array, the second is an int.
    return arr[nsub];
}

string TF(any val) {  // This function is named TF, returns a
    // string value, and accepts a single
    // argument of any type.
    if (val) {
        return "True";
    } else {
        return "False";
    }
}

void setGuesses() {  // This function is named setGuesses,
    // returns no value, and accepts no arguments.
    CmpH.S_map.RlineMap = 2.0;
    ShH.Nmech = 9025.078;
}
```
A function can be defined with global scope or within the scope of any object other than another function. This means function definitions cannot be nested. Functions can, however, be called from within other functions. (See Function Summary in Section 12 for lists for functions.)

2.2.6.1 Using Functions

A function is called by supplying its name followed by a set of parentheses containing its arguments (or empty parentheses if there are no arguments). If the function returns a value, the function call itself is treated just like the value it returns and can be used in expressions or other statements. The examples that follow use the functions defined in Section 2.2.6 above.

```plaintext
real Ts;
Ts = Tstd(10000) + 36;

string det_check;
det_check = TF( B**2 - 4*A*C );

setGuesses();
```

If a function returns an array, the result of the function call should be assigned to an array variable in order to use the functionality of NPSS arrays. Function `entries()` in the example below was introduced in Section 2.2.4.5.

```plaintext
real matrix[] = { {00, 01}, {10, 11, 12, 13} };
real vector[];
real scalar;
int num;
vector = extract(matrix,0); // See the definition of function extract above.
scalar = vector[1]; // extract(matrix,0)[1] won't work.
um = vector.entries(); // extract(matrix,0).entries() won't work.
```

A function can only return a variable’s value, not its other attributes.

```plaintext
real setVane() {
real alpha {
    value       = 5.;
    units       = "deg";
    description = "Variable geometry vane angle";
}
return alpha;
}
real VaneAngle;
VaneAngle = setVane();
```

Any attempt to access `VaneAngle.units` or `VaneAngle.description` will fail. These attributes are not defined by the call to the function.

Function arguments become variables within the function body, and their scope is limited to the function itself. When a function is called, its arguments (if any) receive their initial values from the variables in the calling sequence (the variables within the parentheses following the function name). When the function terminates, however, the values of its arguments are not passed back to the variables in the calling sequence. This means that arguments changed in a function are not changed in the block from which the function was called. (Programmers recognize this as call by value rather than call by reference. NPSS functions are like C++ functions in this regard.)

```plaintext
void DoNothing(real arg) {
    arg += 1;
}
A = 5;
DoNothing(A);
// Variable A still equals 5.
```
Within function DoNothing, variable arg initially equals 5, then is changed to equal 6. In the calling block, however, after the function call, variable A still equals 5.

This means that functions can be used to change variable values in only two ways:

- Return the desired new value.
- Set a variable whose scope is beyond that of the function.

For example:

```c
real Bump_general(real arg) {
  return arg + 1;
}
A = 5;
A = Bump_general(A);
```

After calling Bump_general, A will equal 6.

```c
void Bump_specific() {
  outsideVar += 1;
}
real outsideVar = 5;
Bump_specific();
```

When function Bump_specific is called, NPSS will first look for variable outsideVar within the function. Not finding it there, it will look in the function's parent. Finding it there, it will increase its value by 1. Therefore, after the call to Bump_specific, the value of outsideVar will be 6. The setGuesses() function in Section 2.2.6 above is an example of using a function to set specific variable values whose path is specified.

Special syntax that allows a function to access variables in its parent is discussed in Section 2.2.6.3.

### 2.2.6.2 Function Scope

Functions and Variables in NPSS are scoped in the same fashion. A function name without a specified path is first sought in the block in which it is referenced and then successively in blocks enclosing that block until the top-level assembly (global scope) is reached.

### 2.2.6.3 Accessing a Function's Parent's Variables

A function can access its parent's objects (such as variables or functions) by placing "parent." in front of the object's name. The following example adds another function to subelement S_dP from Section 2.2.6.2. The cout object is used to display information to the screen (see Section 2.2.4.6).

```c
void Diffuser.S_dP.subFunct2() {
  string s = "Diffuser.S_dP. subFunct2()";
  cout << " s is in " << s << endl;
  cout << " parent.s is " << parent.s << endl;
  parent.subFunct();
}
```

The last line of the example above executes function subFunct2(). Its output is as follows:

- `s is in Diffuser.S_dP. subFunct2()`
- `parent.s is Diffuser.S_dP.s`

The reference to parent.subFunct() from within Diffuser.S_dP.subFunct2() correctly executes Diffuser.S_dP.subFunct(), but that function generates no output.
The "parent" feature can be used to access functions NPSS automatically creates for the user. For example, a function can obtain its parent's name using parent.getName(), or check for the existence of an object in its parent using parent.exists() (see Section 13.1.1 for more information on these functions).

It should be noted that the "parent." notation is a general syntax for accessing objects one level up in the hierarchy and works from the scope of any NPSS object. Combining multiple "parent."s together will allow access to objects more than one level up, for example:

```cpp
x.y.z {
    cout << "top description = " << parent.parent.parent.description << "\n";
}
```

### 2.2.6.4 Built-in Functions

NPSS provides a number of built-in functions which behave just like user-written functions (getName() and exists() were mentioned above). Some of the built-in functions have global scope; others are member functions of particular objects (they are part of the local scope of that object). In still other cases, functions with the same name are defined in global scope as well as in the local scope of each element (functions list() and setOption() are examples). The system's built-in functions are summarized in Chapter 12. The same scoping rules apply to built-in functions as to user-written functions.

### 2.2.6.5 Functions Using the Indirect Membership Operator

As noted in Table 8, the indirect membership operator, ->, is used as follows:

```cpp
A->B
```

where A is a string variable whose value is the name of an NPSS object, and B is an attribute or function of that object. The -> operator is used just like "dot" notation. When the name of the object is directly given, use "dot" notation; when the name of the object is indirectly given through the value of a string variable, use -> notation.

```cpp
CmpH.S_map.NcMapDes = 1.;                   // "dot" notation
string subelement_name = "CmpH.S_map";
subelement_name->NcMapDes = 1.;            // "->" notation
```

The indirect membership operator can be used only when its left operand is a string and its right operand is the actual name of an object. Consider the following example:

```cpp
VariableContainer foo {
    VariableContainer bar {
        int ii=-1;
    }
    string subelement_name = "bar";
}

string element_name = "foo";

element_name->subelement_name->ii = 3;
```

This dual indirect membership operation works because both "element_name" and "subelement_name" are strings, and "subelement_name" is resolvable in the scope of the object named in "element_name". Objects found outside of the enclosing scope where the -> operator appears are treated as read-only, so an error will be reported if the -> operation containing the out-of-scope object appears on the left-hand side of an expression, for example:

```cpp
VariableContainer jj {   // enclosing scope is jj
    element_name->subelement_name->ii = 1; // Won't work: 'foo' is read-only
}
```

When a path name must be assembled from the contents of several strings, the strings may also be concatenated together into a single string before using the indirect membership operator.
This method, like the method using multiple -> operators to indicate the path, is also limited by the restriction that out-of-scope objects are read-only, so the following will not work:

```cpp
VariableContainer jj {
  (element_name + subelement_name)->ii = 2;  // Won't work: 'foo' is read-only
}
```

Use of the indirect membership operator allows the user to write powerful functions. Consider the following example which iterates over an array of variable names, printing the name, description, iDescription, value, units, and IOstatus of each.

```cpp
void dumpVarArray(string names[]) {
  int i;
  for (i=0; i<names.entries(); i++) {
    cout << "name  = " << names[i] << endl;
    cout << " descr  = " << names[i]->description << endl;
    cout << " iDescr = " << names[i]->iDescription << endl;
    cout << " value = " << names[i]->value;
    cout << ", units = " << names[i]->units;
    cout << ", IOstatus = " << names[i]->IOstatus << endl;
  }
}
```

In the for statement, the number of entries in the actual array variable named "names" is required; therefore "dot" notation is used to access the array's entries() function: names.entries(). However, to print the attributes of the variables whose names are stored in array names, the indirect membership operator -> must be used, as in the last five cout statements above.

### 2.2.7 Tables

Tables return a real value or array that is interpolated from tabulated values involving one or more independent variables. Tables are actually a special kind of function, and can be defined and called anywhere a function can be defined or called. Whereas functions can optionally return a value, tables always return a value, and this value is always of type real. As with functions, the type of each table argument must be specified. Tables can be defined with a minimum of 1 and a maximum of 6 arguments.

#### 2.2.7.1 Defining Tables

An example of a one-dimensional table follows:

```cpp
Table Ratings1( real PLA ) {
  PLA = { 10.0, 30.0, 50.0 }
  BOT = { 1756.7, 2374.6, 2800.0 }
}
```

The single argument to this table is a real variable known to the table as PLA. The body of the table defines PLA as an array whose elements have one-to-one correspondence with the elements of another array, in this case BOT. The table definition thus establishes BOT as a function of PLA by defining several (in this case, 3) ordered pairs of PLA and BOT. PLA gives the independent values, and BOT gives the corresponding dependent values. When the table is called, a value of BOT will be determined by interpolating between these ordered pairs based on the value supplied for PLA. The default interpolation method is linear, but second- and third-order interpolation is also available (how to select interpolation methods will be discussed in Section 2.2.7.7). The array of independent variables (PLA in the example) must be monotonic.

In the example, the name of the dependent array (BOT) has a different name than the table itself (Ratings1). It is permissible for the same name to be specified for each, and it is often good practice to do so.

Tables can be more than one-dimensional. Suppose BOT is a function of both PLA and altitude:
For each value of \( \text{alt} \), a subtable was specified giving \( \text{BOT} \) as a function of \( \text{PLA} \) for that value of \( \text{alt} \). Notice that these subtables need not use the same values of \( \text{PLA} \), and can even have different numbers of ordered pairs. As many values of \( \text{alt} \) as desired can be given, but they must be specified in a monotonic order. Notice also that although tables always return real values, their arguments need not be real.

A three-dimensional table would be formed by adding a third independent parameter as an argument to the table and defining a two-dimensional subtable for each tabulated value of the third parameter. Each two-dimensional subtable, of course, would consist of a set of one-dimensional subtables. There is no limit to the number of dimensions a table can have.

If a multidimensional table is specified, and one or more of its subtables use the same independent array as the subtable preceding it, a shortened syntax is available, as follows:

```c
Table Ratings2a( real alt, int PLA ) {
    alt = 0. {
        PLA = { 10, 30, 50 }
        BOT = { 1756.7, 2374.6, 2800.0 }
    }
    alt = 10000. {
        PLA = { 10, 20, 30, 40, 50 }
        BOT = { 1702.0, 2066.0, 2317.6, 2545.8, 2797.9 }
    }
    alt = 20000. {
        PLA = *
        BOT = { 1578.9, 1916.8, 2151.0, 2362.5, 2597.8 }
    }
}
```

In the above example, the asterisk indicates that the PLA values of \{10.0, 20.0, 30.0, 40.0, 50.0\} be used for an alt of 20000.

A table’s arguments are not required to be in the same order as their arrays appear in the table definition. In the above example, the positions of real alt and real PLA could be transposed, despite the fact that alt comes before PLA in the table definition.

Additionally, if there is a need to pass into a table more arguments than are actually used, extraneous arguments will be ignored. Consider a version of the table above, only including an extra argument:

```c
Table Ratings2b( int PLA, string UNUSED, real alt ) {
    alt = 0. {
        PLA = { 10, 30, 50 }
        BOT = { 1756.7, 2374.6, 2800.0 }
    }
    alt = 10000. {
        PLA = { 10, 20, 30, 40, 50 }
        BOT = { 1702.0, 2066.0, 2317.6, 2545.8, 2797.9 }
    }
    alt = 20000. {
        PLA = *
        BOT = { 1578.9, 1916.8, 2151.0, 2362.5, 2597.8 }
    }
}
```
This would not produce any type of error, or calculation side effects, as \texttt{UNUSED} is dealt with internally and is essentially ignored.

### 2.2.7.2 Minimum Table Definition

A certain minimum number of independent variable values must be specified for each dimension of an NPSS table. This minimum number depends on the order of interpolation specified, which is discussed in Section 2.2.7.7. The default order of interpolation is linear, which requires at least two independent variable values. For a one-dimensional table, then, the following would be a minimum table definition:

```plaintext
Table Ratings0( int PLA ) {
  PLA = { 10, 50 }
  BOT = { 1756.7, 2800.0 }
}
```

Sometimes it is desirable to return a constant dependent value for all values of the independent variable. To accomplish this, the user must specify the same dependent value for at least two values of the independent variable, for example:

```plaintext
Table Ratings0a( int PLA ) {
  PLA = { 10, 50 }
  BOT = { 2800.0, 2800.0 }
}
```

An alternative is to define a function that returns a constant value:

```plaintext
real Ratings0b( int PLA ) {
  return 2800.0;
}
```

### 2.2.7.3 Calling Tables

Once a table has been defined, it may be called in the same way a function is called. The following example calls table \texttt{Ratings2a()} defined above in Section 2.2.7.1.

```plaintext
Amb.alt = 15000.;
real BrnTemp = Ratings2a( Amb.alt, 45 );
```

A table definition can be part of a function definition. This allows a function like the following, which makes a one-dimensional table look like a two-dimensional table for special cases when the extra independent variable is not needed.

```plaintext
real Ratings2c( real alt, int PLA ) {
  Table Ratings1( real PLA ) {
    PLA = { 10, 30, 50 }
    BOT = { 1756.7, 2374.6, 2800.0 }
  }
  return Ratings1(PLA);  // Variable alt is ignored.
}
```

Tables can also be called without arguments. The Table’s independents use the attribute \texttt{name} which holds the name of a valid numeric variable to be dereferenced. All independents must be assigned a valid reference name or an error will occur. The variable referenced must be of numeric type (real, integer) and follow normal scoping rules, if not an error will be generated.

Example:
// Define a table that requires no arguments when called
Table TB_Z( real x2, real x1 ) {
  x1.name   = "Amb.MN"; // x1 independent reference name
  x2.name   = "Amb.alt"; // x2 independent reference name

  x1 = 0.0 {
    x2 = { 0, 10000., 20000., 30000., 40000. }
    output = { 0., 20., 40., 60., 80. }
  }

  x1 = 1.0 {
    x2 = { 0, 10000., 20000., 30000., 40000. }
    output = { 50., 250., 450., 650., 850. }
  }
} // end TB_Z

Then, just call the Table with no arguments: real z = TB_Z();

2.2.7.4 Scaling Tables

Scaling Return Values

All tables have two attributes that can be used to scale the value returned by the table. The value resulting from the table lookup is first multiplied by \( s_{rtn} \), and then \( a_{rtn} \) is added to the result. The default values \( (s_{rtn} = 1.0, a_{rtn} = 0.0) \) perform no scaling. As with all objects belonging to a parent object, these variables can be set outside the table's definition block using "dot" notation.

Table Ratings1a( int PLA ) {
  PLA = { 10, 30, 50 }
  BOT = { 1756.7, 2374.6, 2800.0 }
}
real Rat_unscaled = Ratings1a(30); // Rat_unscaled will equal 2374.6
Ratings1a.s_rtn = 1.05;
Ratings1a.a_rtn = 14.2;
real Rat_scaled = Ratings1a(30); // Rat_scaled will equal 2507.53

Scaling Independent Values

All tables have two variables that can be used to scale the independent values in the table. During evaluation, each independent value is first multiplied by \( s \) (the scalar); then to that product, \( a \) (the adder) is added. This calculation will not overwrite the original table values; it only occurs during evaluation. The default values \( (s = 1.0, a = 0.0) \) perform no scaling. The value of \( s \) must be greater than zero.

If a user wants to actually change the independent values within the table, the function `applyIndepMods()` (see Section 12.1.20) will apply the scalar and adder to all of the independent entries, overwriting the previous values.

As with all objects belonging to a parent object, these variables can be set outside the table's definition block using "dot" notation.

2.2.7.5 Inverse Lookups

Tables can perform inverse lookups in which the dependent and independent variables are swapped. Do this by using member functions `evalYX()` and `evalYXiter()` as in the following examples:

```c
real P_angle = Ratings1a.evalYX(2700.); // P_angle will equal 45.2985
real P_angle = Ratings1a.evalYXiter(2700.);
```

The difference between `evalYX()` and `evalYXiter()` is that `evalYX()` reverses the independents and dependents and does a normal table read while `evalYXiter()` will do a normal table read while iterating on the independent value to force the dependent value to match. Due to differences in the interpolation, the values from
evalYX() will not give the same independent when put back into the table while evalYXiter() will. Consider the following example:

```c
real ind0 = 8100;
real Mline0 = 0.0824282;
real dep0 = TB_WcMap( ind0, Mline0 );
real ind1 = TB_WcMap.evalYX( dep0, Mline0 );
real ind2 = TB_WcMap.evalYXiter( dep0, Mline0 );
```

Since ind1 is determined from evalYX() it will not be 8100. It be slightly different to interpolating in reverse order. Since evalYXiter() maintains the interpolation order, it will be 8100.

Refer to the definition of table Ratings1a in Section 2.2.7.4 above, before scaling was applied. Notice that even though PLA in table Ratings1a is of type int, the answer returned is 45.2985. Functions evalYX() and evalYXiter() treat PLA as the dependent array, and internally promote its values to reals. Functions evalYX() and evalYXiter() only work when both the independent and dependent arrays are monotonic.

For multidimensional tables, evalYX() evalYXiter() swap the independent and dependent arrays in the lowest level of one-dimensional subtables. For example:

```c
Table Ratings2d( real alt, real PLA ) {
    alt = 0. {        
        PLA = { 10, 30, 50 }
        BOT = { 1756.7, 2374.6, 2800.0 }
    }
    alt = 10000. { 
        PLA = { 10, 20, 30, 40, 50 }
        BOT = { 1702.0, 2066.0, 2317.6, 2545.8, 2797.9 }
    }
}
```  

BrnTemp = Ratings2d(10000, 20); // Standard: BrnTemp = 2066.0    
P_angle = Ratings2d.evalYX(10000, 2317.6); // Inverse: P_angle = 30.

The altitude variable is an independent variable in either case; there is no way to use this table to find altitude as a function of PLA and BOT.

### 2.2.7.6 Stacked Tables

Tables can also be "stacked": they can return multiple dependent values as an array. To do this, first specify several uniquely named dependent arrays, all of which have the same number of values as their corresponding independent array. Then use member function evalStacked():

```c
Table Ratings1b( real PLA ) {
    PLA = { 10.0, 30.0, 50.0 }
    BOT = { 1756.7, 2374.6, 2800.0 }
    Fn = { 2232.3, 6697.5, 11162.0 }
}
real Rated_values[2];
Rated_values = Ratings1b.evalStacked( 40. );
```

Rated_values[0] will hold the interpolated value of BOT; Rated_values[1] will hold the interpolated value of Fn.

The following points pertain to stacked tables:

- Inverse lookups using evalYX() cannot be performed on stacked tables.
- The scale factors s_rtn and a_rtn are applied to each entry of the array returned by a call to a stacked table.
- If a stacked table is called as a normal table (i.e., without using evalStacked()), it returns a single value that is the result of interpolating the first dependent array given in the table.
2.2.7.7 Controlling Interpolation and Extrapolation

Each table independent variable has attributes which control the degree of interpolation and extrapolation. The interpolation attribute is accessed by:

\[ \text{indep\_variable\_name.interp} \]

The extrapolation attribute is accessed by:

\[ \text{indep\_variable\_name.extrap} \]

The allowable values for these variables are shown as follows, along with the minimum number of tabulated points required for each.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Allowable Values</th>
<th>Minimum Number of Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>interp</td>
<td>&quot;linear&quot; (default)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>&quot;lagrange2&quot;</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>&quot;lagrange3&quot;</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>&quot;akima&quot;</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>&quot;hermite&quot;</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>&quot;discrete&quot;</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>&quot;none&quot;</td>
<td></td>
</tr>
<tr>
<td>extrap</td>
<td>&quot;linear&quot; (default)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>&quot;lagrange2&quot;</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>&quot;lagrange3&quot;</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>&quot;akima&quot;</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>&quot;hermite&quot;</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>&quot;discrete&quot;</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>&quot;none&quot;</td>
<td></td>
</tr>
<tr>
<td>extrapHigh</td>
<td>same as above</td>
<td>same as above</td>
</tr>
<tr>
<td>extrapLow</td>
<td>same as above</td>
<td>same as above</td>
</tr>
</tbody>
</table>

If "none" or "discrete" is specified for interp or extrap, the edge value is used whenever extrapolation is necessary. Several of the other options above are illustrated in the following example.

```plaintext
Table Ratings2( real alt, int PLA ) {
    alt = 0. { 
        PLA = { 10, 30, 50 } 
        BOT = { 1756.7, 2374.6, 2800.0 } 
    }
    alt = 10000. { 
        PLA = { 10, 20, 30, 40, 50 } 
        BOT = { 1702.0, 2066.0, 2317.6, 2545.8, 2797.9 } 
    }
}
Ratings2.PLA.interp = "lagrange2";
Ratings2.PLA.extrap = "linear";
Ratings2.alt.interp = "linear";
Ratings2.alt.extrap = "none";
```

The interpolation and extrapolation settings could also have been included within the block defining the table.

The effect of these options is summarized as follows:
A second-order interpolation on \textit{BOT} will be performed at \textit{PLA} = 25 for \textit{alt} = 0 and for \textit{alt} = 10000 since \textit{PLA.interp} = "lagrange2". Then these results will be \textit{linearly} interpolated to \textit{alt} = 5000 since \textit{alt.interp} = "linear".

A linear extrapolation on \textit{BOT} will be performed at \textit{PLA} = 55 for \textit{alt} = 0 and for \textit{alt} = 10000 since \textit{PLA.extrap} = "linear". Then these results will be \textit{linearly} interpolated to \textit{ALT} = 5000 since \textit{ALT.interp} = "linear".

A second order interpolation on \textit{BOT} will be performed at \textit{PLA} = 25 and \textit{alt} = 10000. Since \textit{ALT.extrap} = "none", the edge value (i.e., the value at \textit{alt} = 10000) is used.

If \textit{PLA.interp} = "lagrange3" were specified, an error would be generated since one of the subtables using \textit{PLA} as an independent variable has less than 4 points. Likewise, if \textit{alt.interp} = "lagrange2" were specified, an error would be generated since only 2 values of \textit{alt} have been tabulated.

Setting the \textit{interp} attribute to \textit{discrete} causes the table to return the value of the dependent corresponding to the \textit{closest} independent value. Suppose the example above were modified as follows:

\begin{verbatim}
Ratings2.PLA.interp = "discrete";
\end{verbatim}

Consider for the sake of example, interpolations on the first subtable, with \textit{alt} = 0. If \textit{PLA} = 10, the table will return 1756.7. If \textit{PLA} = 11, the table will again return 1756.7, because 11 is nearest to the tabulated independent value of 10 to which the dependent value of 1756.7 corresponds. If \textit{PLA} = 21, the nearest tabulated independent value is 30, and the table will return 2374.6. If \textit{PLA} = 20, midway between the tabulated values of 10 and 30, the larger independent value is chosen, so that the table would again return 2374.6.

In certain cases it may be desirable set extrapolation methods of a table to different techniques based on the independent values being either off the high or low end of the table. NPSS provides the capability to change extrapolation methods based on this, which use the same allowed values as the general .extrap variable.

These extrapolation attributes are accessed by:

\begin{verbatim}
indep_variable_name.extrapHigh
indep_variable_name.extrapLow
\end{verbatim}

As noted earlier, setting the \textit{extrap} attribute to \textit{discrete} has the same effect as setting it to \textit{none}.

An example of using these features is as follows. Utilizing the following table for engine inlet pressure recovery (ERAM) data may only be provided for a certain range of tested airflows. However if the engine model is desired to run transiently, operations outside the available data range may occur during engine start or transient overshoots. By utilizing some features of NPSS tables, we can better handle these conditions with existing table.
Table ERAM( real WC020 ) { 
   WC2 = { 200., 300., 400., 500. }
   ERAM = { .98, .88, .86, .8 }
} 
ERAM.WC2.interp = "lagrange2";
ERAM.WC2.extrapLow = "none";
ERAM.WC2.extrapHigh = "linear";

The above settings would keep the ERAM value from going greater than 1 in the airflow region from 0 to 200, by capping it at the .98 discrete point. Since the table has some curvature in the actual data range the Lagrange method is used, while for data above the 500 point the table will linearly extrapolate, continuing the trend of the data.

2.2.7.8 Akima Method

Given a set of n data points, (xi,fi), i=0,1,2,....,n-1. Akima’s cubic polynomial on an interval (xi, xi+1) is defined as

\[ P_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \]

Where:
\[ h_t = x_{t+1} - x_t \]
\[ m_t = \frac{f_{t+1} - f_t}{h_t} \]
\[ \omega_t = f_t \]
\[ b_t = \frac{|m_{t+1} - m_t| + |m_{t-1} - m_{t-2}|}{|m_{t+1} - m_t| + |m_{t-1} - m_{t-2}|} \]
\[ c_t = \frac{3m_t - 2b_t - b_{t+1}}{h_t} \]
\[ d_t = \frac{b_t - b_{t+1} - 2m_t}{h_t^2} \]

This method uses five intervals (the desired interval plus the two preceding and the two following intervals) for interpolation. Additional logic provides for the cases in which any of the preceding and/or following intervals are not defined.

### 2.2.7.9 Hermite Method

The Hermite polynomials:
\[ y = ax^3 + bx^2 + cx + d \]
\[ y' = 3ax^2 + 2bx + c \]

With four ascending nodes \((x_1,x_2,x_3,x_4),(y_1,y_2,y_3,y_4)\)
Where input \(x\), in between \(x_2\) and \(x_3\), can satisfy a polynomial:
\[ y_2 = ax_2^3 + bx_2^2 + cx_2 + d \]
\[ y_3 = ax_3^3 + bx_3^2 + cx_3 + d \]
\[ y_2' = 3ax_2^2 + 2bx_2 + c \]
\[ y_3' = 3ax_3^2 + 2bx_3 + c \]

Where the \(y\)'s are: (Note symmetry, so when our \(x\) moves to the next segment of the Table the \(y\)' matches)
\[ y_2' = (y_3-y_1)/(x_3-x_1) \]
\[ y_3' = (y_4-y_2)/(x_4-x_2) \]

Solve for \(a,b,c,d\).
Evaluate at actual \(x\) to get \(y\)

Assumes two nodes on either side of the input \(X\).
Note, simple logic handles cases where there are not two nodes on either side.

### 2.2.7.10 Extrapolation Warnings and Errors

Each entire table has two attributes that control how extrapolations are handled. These are \texttt{printExtrap} and \texttt{extrapIsError}. Either can be set within the table definition, or otherwise outside by using:

```
These attributes are described in the following table.

**Table 10. Table Extrapolation Warning Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Allowable Values</th>
<th>Action Taken When an Extrapolation Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printExtrap</code></td>
<td>FALSE (0) (default)</td>
<td>No notification</td>
</tr>
<tr>
<td></td>
<td>TRUE (1)</td>
<td>Notification</td>
</tr>
<tr>
<td><code>extrapIsError</code></td>
<td>FALSE (default)</td>
<td>Notification = Warning</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>Notification = Error</td>
</tr>
</tbody>
</table>

No notification is provided if `printExtrap = FALSE` regardless of the value of `extrapIsError`. When `printExtrap = TRUE`, any warning and error notifications issued are *provisional*. This means that if the fault condition exists when the solver exits, the message is issued then, and the program continues running. Provisional fault conditions that occur momentarily during the solver's iterations, but clear by the time it exits generate no messages. A solver must be present for a provisional error or warning to be printed. The example that follows will provide no messages unless it is associated with a solver.

```plaintext
Ratings1( real PLA ) {
   PLA = { 10.0, 30.0, 50.0 }
   BOT = { 1756.7, 2374.6, 2800.0 }
}
Ratings1.printExtrap = TRUE;  // Overrides default value of FALSE.
Ratings1( 55. );             // PLA is extrapolated.
CASE++;
run();                      // Upon solver termination, a warning message is issued.
Ratings1.extrapIsError = TRUE;
Ratings1( 55. );
CASE++;
run();                      // Upon solver termination, an error message is issued.
Ratings1.printExtrap = FALSE;
Ratings1( 55. );
CASE++;
run();                      // Even though extrapIsError = TRUE, no message is issued.
```

### 2.3 Interactive Mode Syntax

#### 2.3.1 Introduction to Interactive Mode Syntax

The mode of operation in view in preceding sections involves storing NPSS commands in various files that are processed by NPSS without further interaction with the user. Interactive mode allows the user to enter commands to NPSS directly from the keyboard while NPSS is active. Commands exist to control the entry into interactive mode, and to transition back and forth between interactive mode and file-based mode. Interactive mode can be a powerful tool to debug or interrogate a model.

Command line options affecting interactive mode are discussed in Sections 2.1, 2.3.2, and 2.3.3.

#### 2.3.1.1 Interactive Mode Input

Interactive mode is indicated by a prompt on the user's screen of two greater-than signs, ">>", followed by a space. When NPSS is waiting for information continued from a preceding line, the prompt changes to "more?".

Interactive mode syntax and file-based syntax require the semicolon statement terminator. In interactive mode as well as file-based syntax, one line is continued to the next simply by starting a new line without the semicolon statement terminator.

```plaintext
>> real ra[] = { 0.0, 1.0, 2.0, 3.0,
```
In interactive mode, the value of any expression that is not assigned to a variable will be displayed on the terminal. For example:

```
>> ra;
    { 0, 1, 2, 3, 4, 5, 6 }
>>
```

This also allows the output of functions such as list() to be displayed easily.

```
>> list( "Subelement", 1 );
    { "CmpH.S_map", "TrbH.S_map" }
>>
```

All standard file-based NPSS commands are also valid in interactive mode. For example:

```
>> int i;
>> for ( i=0; i<ra.entries(); i++ ) {
        more?   Cout << ra[i] << ", ";
        more? };
    cout << endl;
    0, 1.5, 2, 3, 4, 5, 6,
>>
```

2.3.1.2 Interactive Mode Command Line Editing

The arrow keys and backspace keys can be used as follows:

- Up arrow ("↑") and down arrow ("↓") scroll through a saved list of previously entered commands. A displayed command is available to be executed again, or to be edited before execution.
- Left arrow ("←") and right arrow ("→") move the cursor left and right through a displayed command.
- The backspace key deletes one character at a time to the left of the cursor position.

New text can be inserted into a displayed command at the cursor location by simply typing the text. There is no overstrike mode of editing available. Once a command has been edited as desired, pressing the enter or return key executes it.

2.3.2 Entering Interactive Mode

There are three ways to enter interactive mode:

1. Use the –i command line option when starting an NPSS run to enter interactive mode after all file-based processing is complete. See Section 2.1.
2. Execute function icl.start(); from within an input file.
3. Execute a breakpoint from within an input file.

NPSS object icl("interactive command line") is automatically created for the user to control entrance into interactive mode. As noted above, icl.start(); will halt normal file processing and enter interactive mode at the place the command is encountered in an input file. The user can also define function icl.preexecute(); which will execute once at the start of each interactive mode session.

Breakpoints are discussed in the following section. It can be noted here, however, that the user can define functions icl.preBreak(); and icl.postBreak(); to execute in connection with active break points.

Function icl.preBreak(); executes whenever a breakpoint executes. Function icl.postBreak(); executes whenever processing of file-based commands is resumed using commands continue, step, stepOut, or stepOver (see Section 2.3.4). These commands cause interactive mode to be reentered after executing one or more file-based commands. Upon reentering interactive mode, icl.preBreak(); will be executed first, then icl.preexecute();.

56 2 Syntax Guide
2.3.3 Breakpoints

Breakpoints are used to enter interactive mode at desired points in a model's execution. They are like icl.start(); but provide the user more power in determining the location of the start of interactive running. Breakpoints are ignored unless at least one of the following conditions is met:

- NPSS was started with the `-debug` command line option (see Section 2.1).
- Command `debug()` has been executed in the file. Only breakpoints encountered after the `debug()` command will be processed. See Section 12.3.

2.3.3.1 The `breakPoint()` command

Breakpoints are established using the `breakPoint()` command as follows:

```
breakPoint( location, condition, action );
```

All three arguments are optional, but to enter `condition`, `location` must also be entered, and to enter `action`, both `location` and `condition` must be entered. Executing `breakPoint()` without any arguments causes an immediate break to interactive mode at that location, like `icl.start()`.

Location

The `location` argument is a string (enclosed in double quotes) consisting of a keyword and one or more location indicator separated by colons (" : "). The syntax options are as follows:

- `ENTRY: function_pathname:occurrence_number`
- `EXIT: function_pathname:occurrence_number`
- `FILE: filename, line_number:occurrence_number`
- `WATCH: parameter_name`

For keywords `ENTRY`, `EXIT`, and `FILE`, the occurrence number is optional. If present, it is a single integer number that causes a break to interactive mode to occur only on or after the specified number of times when the break would otherwise have executed.

The keywords cause the following behavior:

- `ENTRY`: causes a break just before the named function is entered. For example:
  
  ```
  breakPoint( "ENTRY:CmpH.preexecute" );
  ```

- `EXIT`: causes a break after the last statement has been executed but before exiting the function.
  
  ```
  breakPoint( "EXIT:CmpH.postexecute" );
  ```

- `FILE`: causes a break before the statement on the indicated line is executed.
  
  ```
  breakPoint( "FILE:interactive.file,2" );
  ```

- `WATCH`: causes a break whenever the value of the named parameter changes.
  
  ```
  breakPoint( "WATCH:CmpH.Fl_I.W" );
  ```

Breaks can only be made in interpreted functions. For example, all `Element` objects have an internal function named `calculate()` that is executed every time the object is executed. However, the following breakpoint:

```
breakPoint( "ENTRY:CmpH.calculate" );
```

will have no effect unless object `CmpH` was created from interpreted code.

There are five exceptions to this rule. The user can set breakpoints for the following internal solver functions:

Table 11. Internal Solver Functions for Which a Breakpoint May Be Specified

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>run()</code></td>
</tr>
<tr>
<td><code>runPreExecutionSequence()</code></td>
</tr>
<tr>
<td><code>runExecutionSequence()</code></td>
</tr>
<tr>
<td><code>runPostExecutionSequence()</code></td>
</tr>
<tr>
<td><code>generateJacobian()</code></td>
</tr>
</tbody>
</table>
For example, by entering the following breakpoint:

```
breakPoint( "EXIT:solver.generateJacobian" );
```

the user can examine each Jacobian matrix after it has been generated. More discussion on the preExecutionSequence, executionSequence, and postExecutionSequence attributes of solver objects is found in 4.6.1. General solver theory and operation are discussed in Chapter 6.

Breaks in files are only made when the file is processed by the parser. For example, suppose a model includes (using #include) file hpc.map containing subelement instantiations belonging to Compressor element CmpH. Suppose after the file has been processed, the following breakpoint is set:

```
breakPoint( "FILE:hpc.map,1" );
```

Even though the information in this file is used repeatedly in the execution of element CmpH, the file itself is not reprocessed, so the breakpoint will have no effect. The breakpoint would have to be set before the initial #include "hpc.map" statement in order to effect a break.

Furthermore, a break in a file can only be performed at lines representing the start of complete statements. For example, file hpc.map used in the example above would normally contain lengthy table definitions spanning many lines of the file. Attempting to break at a line in the middle of the table definition will not work, because the parser treats the entire table definition as a compound statement.

The file name specified with the FILE: keyword must be recognizable to NPSS. If in doubt about the name NPSS associates with a file, force a break to interactive mode somewhere in the file (for example, using icl.start()), and issue the show command (see Section 2.3.4).

Keyword watch:, which looks for changes in the value of a named parameter, should be used with care because it can greatly add to execution time. It can also be difficult to determine which object produced the change if the change occurred in compiled code (see notes under command where in Section 2.3.4). If a variable being "watched" is changed in compiled code, interactive mode will be entered sometime after the exit from that segment of code. If the variable is changed by a line of interpreted code, interactive mode will be entered after the execution of that line.

**Condition**

The condition argument is a string containing an expression that should evaluate to an integer value. The integer is treated as a Boolean value (0 = "false", nonzero = "true"). If the condition expression evaluates to "false," the breakpoint will be skipped. This argument defaults to "true". To force a value of "true," syntax such as the following can be used:

```
breakPoint( "ENTRY:solver.runExecutionSequence:5", "TRUE" );
```

The condition argument must always be enclosed in quotes because it is a string. In this case, the string evaluates to the value of the automatically created global variable TRUE (see Table 7. Automatically Created Global Variables) which holds the value 1.

The following command causes a break to interactive mode at the beginning of CmpH.postexecute() if the CmpH inlet Mach number is greater than 0.28.

```
breakPoint( "ENTRY:CmpH.postexecute", "CmpH.Fl_I.MN > 0.28" );
```

**action**

The action argument is also a string containing a command (or series of commands separated by semicolons) that is executed when the break is executed. A common usage for the action argument is to print a value or set of values to the screen. Object cout is generally used for this. Strings are always enclosed in quotes, but it is often necessary to use quotes as part of a cout command. To do this (as discussed in Section 2.2.4.6), precede any quotation marks within the action string by a backslash ("\") as in the following example:
2 Syntax Guide 59

breakPoint( "ENTRY:solver.run", "Amb.MN > 0.5",
   "cout << \"Amb.MN = \" << Amb.MN << endl; ");

This breakpoint causes an entry into interactive mode at the beginning of the solver's run() command if the ambient Mach number is greater than 0.5. When the break is executed a message is written to the screen informing the user of the magnitude of the ambient Mach number.

2.3.3.2 Breakpoint Number

Each breakpoint created in a given NPSS session is assigned a unique integer identification number, beginning with 0 and increasing sequentially. This breakpoint number is used to identify the breakpoint in the commands discussed in the following section. Once a breakpoint number has been assigned, it is never reassigned to another breakpoint, even if the original breakpoint is deleted. For example, if eight breakpoints (numbered 0-7) were created, and later all deleted, the next breakpoint created after these events would be given number 8.

2.3.3.3 Commands Used with Breakpoints

The following rules generally apply to the breakpoint_number argument in the commands discussed in this section:

- A positive integer value selects the breakpoint of that number.
- A negative value (-1) selects all breakpoints.
- Omitting the argument entirely selects the current breakpoint. When the command is entered from interactive mode, the current breakpoint is the breakpoint that initiated the current interactive session.

The following commands can be issued either from interactive mode, or as file-based commands.

listBreak ( breakpoint_number, show_condition );
Returns a string array containing information about existing breakpoints. Each entry gives a breakpoint number, its location string, its status (whether active or inactive), and optionally its condition string. Argument show_condition is a string containing an expression that should evaluate to either "true" (nonzero integer) or "false" (integer 0). It defaults to "false," and can be omitted entirely. If "true," the condition string is included in the information returned. To list all existing breakpoints, execute listBreak( -1 ). This is a convenient way to confirm the breakpoint number assigned to each breakpoint.

The NPSS language (like C and C++) does not allow the user to omit initial arguments to a function. Thus, it is not possible to list the condition string of the current breakpoint in a single step. The user should discover the breakpoint number of the current breakpoint by issuing listBreak() (with no arguments), and then issue the command again using that breakpoint number, as in listBreak( 5, "TRUE" ).

suspendBreak ( breakpoint_number );
Makes the specified breakpoint inactive. Inactive breakpoints still exist, but are ignored.

activateBreak ( breakpoint_number );
Makes the specified breakpoint active. Breakpoints are active when created, so this command is used to reactivate breakpoints that were previously made inactive by suspendBreak().

clearBreak ( breakpoint_number );
Deletes the specified breakpoint. Although the number associated with a deleted breakpoint is not reused, the breakpoint no longer exists. It is not listed by listBreak(), and cannot be reactivated by activateBreak().

2.3.4 Interactive Debugging Commands

The following commands are available only in interactive mode. They work whether interactive mode was entered through icl.start() or through a breakpoint. They are generally used when tracing the execution of a file-based model to understand unexpected behavior.
**continue**

Ends the current interactive session and resumes normal file-based execution. Interactive mode will be reentered if an active breakpoint or an `icl.start()` command is encountered. Compare with `quit` and the `step` commands below.

**load filename**

Reads and processes the contents of the specified file. This is equivalent to `#include <filename>` in file-based processing. As with `#include`, the `load` command searches for the specified file using the NPSS include path (see Sections 2.1 and 2.2.3.1).

**quit**

Ends the interactive session and terminates the program. Compare with `continue` and the `step` commands.

**show**

Displays the absolute pathname and line number of the current file. See notes under the `where` command below.

**step**

Executes the next interpreted statement and then returns to interactive mode. If the next statement results in a call to an interpreted function, either directly or indirectly, interactive mode is activated upon entry into that function. Otherwise, interactive mode resumes after the current statement is executed.

It takes two step commands to get through a statement such as:

```plaintext
int x = 11;
```

The first step is for the declaration of `x`; the second step is for the assignment of value 11 to the variable.

Repeated executions of the step command are conveniently accomplished using the up arrow key followed by the enter or return key (see Section 2.3.1.2).

Commands `step`, `stepOut`, and `stepOver` only function properly if `-debug` has been specified on the command line (see Section 2.1) or command `debug()` has been entered (see Sections 2.3.3 and 12.3).

**stepOut**

Completes execution of the current block of interactive code (unless an active breakpoint or `icl.start()` is encountered), then returns to interactive mode. The current block can be a loop (Section 2.2.5.6), a conditional (Section 2.2.5.4), or an interpreted function body (Section 2.2.6).

**stepOver**

Behaves the same as `step`, except that interactive mode resumes after the current statement is executed, even if the current statement results in a call to an interpreted function (unless an active breakpoint or `icl.start()` is encountered).

**trace [on/off]**

Command `trace on` turns trace on; `trace off` turns it off. Entering `trace` with no following keyword causes the system to report the status of trace (on or off).

When trace is on, the system writes to standard output a record of each interpreted statement and function call as it is executed. The value of each expression is shown in parentheses after the expression. Trace can be turned on from the very beginning of an NPSS session using the `-trace` command line option (see Section 2.1). Trace can be controlled in batch mode via the `traceExecution()` function.

**where**

Displays the stack trace of functions that were called. This command often does not show anything below the global call to `run()`, even when interactive mode is entered in an interpreted function such as a `preexecute()`. Function `show` above is sometimes more helpful in finding the location of a break to interactive mode. When `WATCH` is being used (see Section 2.3.3.1), interactive mode may be entered at a piece of interpreted code even when the variable change took place in compiled code. In these cases use of `where` and `show` can be misleading.
3 Acquiring Components in NPSS

Components include Elements, Subelements, and Assemblies – the primary building blocks of NPSS simulations. There are four ways to define NPSS components, and several ways to acquire them, or make them available for use in a specific simulation. These are summarized in the following table.

Table 12. Component Acquisition Methods

<table>
<thead>
<tr>
<th>Component Definition</th>
<th>Acquisition Method</th>
<th>Description</th>
<th>Further Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpreted</td>
<td>Write component definition into an NPSS input file.</td>
<td>Use the NPSS input syntax already described plus some special commands to define a component with the desired functionality.</td>
<td>Developer Guide.</td>
</tr>
<tr>
<td>interpreted</td>
<td>#include</td>
<td>Use the #include preprocessor command to draw a file containing an interpreted component definition into an NPSS input file. The location of the desired file must be part of the include path specified to NPSS.</td>
<td>Sections 2.2.3.1 and 2.1</td>
</tr>
<tr>
<td>interpreted</td>
<td>Interpreted Component Load on Demand (ICLOD)</td>
<td>NPSS can automatically load interpreted component definitions when they are needed.</td>
<td>Section 3.1 below</td>
</tr>
<tr>
<td>Dynamically Loadable Module (DLM)</td>
<td>command line</td>
<td>Use the –l command line option to specify a specially compiled piece of code called a DLM to be loaded with NPSS.</td>
<td>Section 2.1</td>
</tr>
<tr>
<td>DLM</td>
<td>loadDLM()</td>
<td>Use function loadDLM() either interactively or in an NPSS input file to load a DLM.</td>
<td>Section 12.2.4</td>
</tr>
<tr>
<td>DLM</td>
<td>DLM Component Load on Demand (DCLOD)</td>
<td>NPSS can automatically load component definitions existing as DLMs when they are needed.</td>
<td>Section 3.3 below</td>
</tr>
<tr>
<td>internal</td>
<td>Compile into the NPSS executable.</td>
<td>C++ code defining components can be written and compiled into NPSS.</td>
<td>Developer Guide.</td>
</tr>
<tr>
<td>external</td>
<td>External Component Load on Demand (ECLOD)</td>
<td>Specially wrapped executables representing components can be accessed by NPSS using a CORBA-compliant object request broker.</td>
<td>Section Error! Reference source not found. below.</td>
</tr>
</tbody>
</table>

Creation Method Facilities (CMFs) are NPSS modules that search for and acquire necessary components at run time. The available CMFs, and the order in which they are accessed, is given by function listLoadedCMFs(). NPSS currently provides three CMFs, called in the following order.

1. DLM Component Load on Demand (DCLOD)
2. Interpreted Component Load on Demand (ICLOD)
3. External Component Load on Demand (ECLOD)

ECLOD is available only when a CORBA-aware NPSS executable is called with the -corba command line option. ICLOD and DCLOD are always available unless disabled by command line options.
### 3.1 Interpreted Component Load on Demand (ICLOD)

The Interpreted Component Load on Demand (ICLOD) CMF assists in finding and loading interpreted components. ICLOD works on the principle that the file `component_name.int` contains exactly one interpreted component named `component_name`. For example, `Duct.int` is an ASCII file containing NPSS input syntax defining an object type named Duct. It uses this one-to-one mapping of an interpreted component name to a file name to look for and load a component.

When ICLOD is asked to create a component, for example Duct, it searches a given path for a file named Duct.int. If the ICLOD_PATH environment variable is set, ICLOD searches the directories specified by the contents of that variable, in the order they are specified. If the file is not found in ICLOD_PATH, ICLOD then searches the NPSS include path. This is the combination of the environment variable NPSS_PATH and any paths specified using the -I command line option (see Section 2.1). For informational purposes, ICLOD shows the search path it is using in variable ICLODPATHS (see Table 7). ICLODPATHS may be modified by the user by simply setting ICLODPATHS equal to a string array of desired paths, as shown in this example.

```
ICLODPATHS = {"/NPSS/dev/CustomPath", "/NPSS/dev/CustomPath2"};
```

Setting ICLODPATHS in this manner will discard any previous paths that were being held in ICLODPATHS. So if a user wants to append paths to the existing ICLODPATHS, the user may wish to incorporate a temporary string array to hold the original values, as shown here:

```c
string tempPaths [] = ICLODPATHS;  //store ICLODPATHS in tempPaths
string newPaths[]={"/new/path/1", "/new/path/2")
tempPaths += newPaths;  //combine the old and new paths
ICLODPATHS = tempPaths;  //net result is appending new paths
```

As currently implemented, ICLOD is not active until it is used in some way. Thus, when NPSS is first started, variable ICLODPATHS does not exist, and a command like `cout << ICLODPATHS` will fail. Calling any function that touches ICLOD, such as `listLoadedCMFs()` or `listKnownTypes()` will activate it.

If ICLOD finds a file of the name it is seeking, it then parses that file. If the file contains inappropriate code, such as a `quit()`, this will be executed with all its consequences. ICLOD tries to inform the user when it encounters a parsing error while reading the file, or when it successfully parses the file but does not find the appropriate component definition.

### 3.2 Overview of Dynamically Loadable Modules (DLMs)

Dynamically Loadable Modules (DLMs) allow NPSS to use compiled components not statically compiled into the code. Thus, any number of components can be compiled into an entity similar to a shared library to be dynamically loaded into NPSS at run time if and when needed. DLMs can provide a wide range of functionality. “Components,” then, can be taken in a very general sense: Elements, Subelements, Assemblies, thermodynamics packages, Ports, DataViewers, functions, CMFs, customer decks, compiled models, etc. It is even possible to have CORBA support be a DLM. Most users, however, will use DLMs as means to make Elements, Subelements, and Assemblies available to NPSS in a form that will execute roughly twice as fast as their interpreted code counterparts.

A DLM component is created in the same way a regular compiled NPSS component is created, only using a special `make` procedure. See the Developer Guide for details.

A DLM is strongly tied to a particular NPSS executable (more specifically to a particular version of the NPSS Executive library). If a header file in the Executive library is changed, DLMs must be recompiled to work correctly with NPSS executables built on the new library. When loading a DLM, NPSS compares the DLM version ID to its own version ID. If they differ, the load will fail and an error message will be generated.

As summarized in Table 12 above, a DLM can be loaded into NPSS:
1. By passing \texttt{-l pathToDLMModule} as an argument on the NPSS command line. For example:
   \texttt{npss.exe -l ./MyComponents.scl other_arguments_to_executable}
2. By calling function \texttt{loadDLM(pathToDLMModule)} either interactively or from an input file. For example:
   \texttt{loadDLM("./MyComponents.scl");}
3. By using DCLOD (discussed below).

In addition, one can call the Executive library's \texttt{DM::loadDLM(const NCPString & pathToDLM)} function from C++ code. For example,

```c++
#include <DataManagerInterface.H>
...
void someFunct(some args) {
    DM::loadDLM("./MyComponents.scl");
}
```

This option will probably not be called from a component but will most often be called behind the scenes by, for example, a site-specific Creation Method Facility.

### 3.3 DLM Component Load on Demand (DCLOD)

The \texttt{DLM Component Load on Demand (DCLOD)} CMF assists in loading DLMs. DCLOD works on the principle that the file named \texttt{component_name.scl} contains exactly one DLM component named \texttt{component_name}. For example, \texttt{Duct.scl} is a compiled DLM file defining an object type named \texttt{Duct}. It uses this one-to-one mapping of a DLM component name to a file name to look for and load a component.

When DCLOD is asked to create a component, for example \texttt{Duct}, it searches a given \texttt{path} for a file named \texttt{Duct.scl}. If the \texttt{DCLOD_PATH} environment variable is set, DCLOD searches the directories specified by the contents of that variable, in the order they are specified. If the file is not found, DCLOD then searches the NPSS include path. This is the combination of the environment variable \texttt{NPSS_PATH} and any paths specified using the \texttt{-I} command line option (see Section 2.1). For informational purposes, DCLOD shows the search path it is using in variable \texttt{DCLODPATHS}. This is similar to the INCLUDEPATHS variable that lists the include paths NPSS is using. Currently \texttt{DCLODPATHS} is read only, meaning that the paths it searches cannot be modified at run time.

As currently implemented, DCLOD is not active until it is used in some way. Thus, when NPSS is first started, variable \texttt{DCLODPATHS} does not exist, and a command like \texttt{cout << DCLODPATHS} will fail. Calling any function that touches DCLOD, such as \texttt{listLoadedCMFs()} or \texttt{listKnownTypes()} will activate it.

If DCLOD finds a file of the name it is seeking, it then loads that file. If the file does not contain the component sought for, DCLOD throws an exception indicating that it cannot create the requested component. \textit{There is no guarantee that the DLM does not contain inappropriate code.}

### 3.4 How NPSS Uses CMFs

As mentioned earlier, NPSS can recognize several CMFs and uses them in a specified order. Currently NPSS has three standard CMFs accessed in the following order:

1. DLM Component Load on Demand (DCLOD)
2. Interpreted Component Load on Demand (ICLOD)
3. External Component Load on Demand (ECLOD) – only if the \texttt{-corba} command line option is used

When NPSS is asked to create a component it cannot find, it will normally ask DCLOD to find the component as a DLM. This strategy seeks to minimize execution time since DLM components execute roughly twice as fast as comparable interpreted components. As noted above, if DCLOD finds a file named as though it contains the desired component, but the file does not contain the component, an error results. If, however, DCLOD cannot find a file of the appropriate name (\texttt{component_name.scl}), it simply returns control to NPSS, which calls the next available CMF, namely ICLOD. Similarly, ICLOD will either find the desired component, generate an error due to incorrect
content in a file named `component_name.int`, or fail to find the file. If NPSS exhausts all its CMFs without finding the component required, it generates an error and stops execution.

As summarized in Section 2.1, the user can modify this behavior with command-line options.

- `-iclodfirst` Causes ICLOD to be used before DCLOD.
- `-nodclod` Disables use of DCLOD.
- `-noiclod` Disables use of ICLOD.

If a component is not found using DCLOD and ICLOD, and the –corba command line option was used, NPSS seeks the component using ECLOD.

### 3.5 Component Creation Query Function `findResource()`

Function `findResource()` can be used to indicate how NPSS can create a component. The function is called with the component name as its argument: `findResource(component_name)`. Note that the component in question is a type (such as `Compressor`), not a specific instance of a type (such as `CmpH`). See Section 12.2.4 for details. Also discussed in Section 12.2.4 is function `loadResource()` that can be used to manually load components.

### 3.6 Metadata

Metadata is used in NPSS to describe classes that have not yet been loaded into NPSS. Without metadata, there is no way to determine the base type of a class that has not been loaded, making it impossible to categorize unloaded components by type, e.g., as Elements and Subelements. Metadata also provide dependency information allowing NPSS to load any necessary DLMs before a given DLM is loaded. These dependencies can arise when one DLM component inherits from another, or when a given DLM component contains references to some other component type. The NPSS converter will automatically generate a metadata file during conversion, placing the file in the directory specified by the `–mo` command line argument.

If you are adding a new component or thermopackage to NPSS and are not using the converter to create a DLM out of it, you will have to manually create your own metadata file. Manually created metadata files should be placed in the `$NPSS_TOP/MetaData` directory or in some other directory in the NPSS include path. All metadata files must have an extension of `.met`. The metadata file syntax is described below.

```plaintext
[ClassName]
metavar1 = val1
metavar2 = val2
...
metavarn = valn
```

The `ClassName` specifier is optional. If it is not specified, the name of the metadata file, without the `.met` extension, will be assumed to be the class name. The right-hand-side of each assignment may be an integer, real, string literal, or array. Arrays are currently not supported. Metadata files for thermopackages contain two `ClassName` specifiers. One specifies a FlowStation and the other a FuelStation. This is necessary because a thermopackage DLM contains two classes, unlike the typical component DLM or `.int` file, which contains only a single class.
4 Common Tasks

Chapter 2 covered the basic building blocks of the NPSS input language. Chapter 3 covered how to gain access to more complex pre-written NPSS objects. This chapter covers how to use these items to create complete NPSS models. The first several sections discuss the basics of key NPSS objects: thermodynamics packages, elements, subelements, assemblies, and how to connect them together. The specification of bleeds, solver setup, and output formats are also discussed. After covering the basics of running a model, a number of special topics are treated, such as how to modify an existing model, expand the capability of existing objects, generate initial guesses, handle error messages, and match test data.

4.1 Thermodynamics Packages

Calculations involving fluid properties require a thermodynamics package that models the fluid. NPSS offers the user the ability to choose the thermodynamics package to be used. Table 13 below lists the packages distributed with a standard NPSS release.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEA</td>
<td>Implementation of the NASA chemical equilibrium code.</td>
</tr>
<tr>
<td>Janaf</td>
<td>Implementation of the National Institute of Standards and Technology gas properties prepared by Honeywell.</td>
</tr>
<tr>
<td>GasTbl</td>
<td>Package developed by Pratt &amp; Whitney based on Therm, but adding humidity calculations and some chemical equilibrium capabilities.</td>
</tr>
<tr>
<td>AllFuel</td>
<td>Package developed by General Electric that contains gas properties and fuel properties.</td>
</tr>
<tr>
<td>FPT</td>
<td>Package used to define NPSS tables and/or functions that describe the thermodynamic properties of the fluid</td>
</tr>
</tbody>
</table>

A thermodynamics package must be specified before any elements or subelements are instantiated. Its specification, therefore, is generally among the first actions accomplished in an NPSS model. A thermodynamics package is specified with command `setThermoPackage()`. For GasTbl, Janaf, allFuel, and FPT this command takes one argument defining the package. For example,

```plaintext
setThermoPackage( "GasTbl" );
setThermoPackage( "Janaf" );
setThermoPackage( "allFuel" );
setThermoPackage( "FPT" );
```

The `setThermoPackage` command can be executed any number of times to allow the use of more than one thermodynamics package in a single model. Care must be taken, however, since the enthalpy and entropy bases for packages often differ. The copyFlow and add functions throw a warning if the thermodynamic package is of a different type. The model developer will need to compensate for this by emulating this functionality within the component in question. See the NPSS Developer’s Guide for details on creating components. See the NPSS Thermodynamic Property Package User Guide for details on using the individual thermodynamics packages.

To run CEA, the user must have CEA "thermo.inp" and "trans.inp" files (if transport properties are to be calculated) in the NPSS include path. These files can be created by running CEA off-line with a library input file.

For CEA, the `setThermoPackage` command takes several arguments defining the thermodynamic package, the name of the "thermo.inp" file, the name of the "trans.inp" file and also the names of the possible reactants. An example is shown below:

```plaintext
setThermoPackage( "thermo.inp", "trans.inp", "reactant1", "reactant2", "reactant3" );
```

To run CEA, the user must have CEA "thermo.inp" and "trans.inp" files (if transport properties are to be calculated) in the NPSS include path. These files can be created by running CEA off-line with a library input file.

For CEA, the `setThermoPackage` command takes several arguments defining the thermodynamic package, the name of the "thermo.inp" file, the name of the "trans.inp" file and also the names of the possible reactants. An example is shown below:
  "", "" );

When setting the thermodynamic package to CEA, the third argument of the setThermoPackage() function does not need a value and can be input as blank string. This will cause the default "trans.inp" file to be loaded. The fourth argument identifies which CEA reactant represents air. The fifth argument identifies which CEA reactant represents water. The additional names represent the other possible reactants. They do not need values and can be input as blank strings. All these reactants must be defined in the "thermo.inp" and "trans.inp" files.

The FPT package allows the user/developer to create tables and/or functions to describe the thermodynamic properties of a fluid. These tables and/or functions are defined in NPSS interactive format. The tables and/or functions are placed in a file with the name "fluidName.fpt." These files can be converted to C++.

To use the fluid, set the composition (flowstationName.setComp("fluidName");) then use the flowstation as you would any other. A list of the functions available for FPT flowstations can be found in the Reference Sheets document.

For expediency, and to support new proprietary thermodynamics packages, the setThermoPackage() command will accept an arbitrary number of string arguments in addition to the mandatory first argument defining the package. These additional arguments, if any, will be stored in the FlowStation class' thermoOptions attribute as an array of strings. CEA does not support this additional argument capability. The following example demonstrates a setThermoPackage() command with eleven custom arguments. For example:

    setThermoPackage("GasTbl", "parm1", "parm2", "parm3", "parm4", "parm5", 
    "parm6", "parm7", "parm8", "parm9", "parm10", "parm11");

The stored arguments can be extracted using the FlowStation accessors getThermoOptList() and getThermoOption(). For example:

    FlowStation myFlowStation;

Constructs a GasTblFlowStation based on the above setThermoPackage command, and:

    myFlowStation.getThermoOptList(); // Returns "parm1" through "parm11".
    myFlowStation.getThermoOption(5); // Returns "parm5".

Returns the requested thermoOptions. Note that the getThermoOption() indexing starts at one rather than zero.

### 4.2 Elements

**Elements** are an important class of NPSS objects. As summarized earlier in Section 1.5.1, elements are used to model the basic components of an engine cycle, such as compressors, turbines, burners, ducts, and nozzles.

Element objects are instantiated using the following items in order:

1. Keyword Element (optional)
2. The specific element type
3. The object name
4. An optional block of statements (enclosed in curly braces) executed when the object is instantiated. If present, this block is called the **instantiation block**. This block of statements can instantiate other objects that become children of the element object.

For example:
In this case, variable eRamBase is a variable created by the system in all objects of type Inlet (the variables created by the system for each element and subelement type can be found in the NPSS Reference Sheets).

The instantiation block is optional. Many users prefer to include an empty set of curly braces if no instantiation block statements are required. It is also permissible, however, to terminate the statement with a semicolon after the object name.

As noted in Section 2.2.2.4, a block of statements can be associated with an object after its instantiation by giving the object name immediately before the block:

```plaintext
Inl {  
  a_eRamAud = 0.0;  
  s_eRamAud = 1.0;  
}
```

where a_eRamAud and s_eRamAud are predefined variables contained in objects of type Inlet. Everything that can be done in an object's instantiation block can also be done in a block such as that above associated with an object after its instantiation.

If users wish, when they instantiate an element they have the option of giving it a version designation, or number, using the string variable "version" as follows:

```plaintext
Element TESTELT elt {  
  version="Test version 1";  
  ...
}
```

Note also that the attributes description and iDescription (as discussed in section 2.2.4.1) exist for complex objects such as elements (and subelements). For such objects, description is intended to explain the general purpose of the object type, and is usually given a predefined default value by the element writer. Attribute iDescription is intended to explain the purpose of a specific instance of a type. It can be supplied by the user to help document a model.

The order in which the NPSS solver executes elements is controlled by string array variable executionSequence. Each time an Element object is instantiated, NPSS appends its name to executionSequence. Thus, by default, NPSS will execute its elements in the order they are instantiated. By editing array executionSequence, the user can easily change the execution sequence, so in principle the user is free to instantiate elements in any order desired. The only restrictions are that any object must be instantiated before its internal attributes are assigned values, and before it is referenced by another object. It is good practice, however, to instantiate elements in a logical order, generally following the gas flow from inlet to exit.

In addition to using pre-written standard elements, NPSS allows the user to create entirely new element types which can then be instantiated and used in a model. Details on how to do this are found in the Developer Guide.
4.3 Ports

As introduced in Section 4, NPSS elements communicate with each other via *ports*. Five kinds of ports exist in NPSS:

- fluid
- fuel
- shaft
- data
- file

The presence of a port signals a collection point for properties to be shared with another element: fluid properties, fuel properties, and shaft properties. In the case of a data port, a single specified data value is shared, and for a file port, one or more files. Each port on an element is usually connected to a port on another element so that the two elements share the properties of the port as a mutual boundary condition. For example, if the core flow of a gas turbine engine passes from the high pressure compressor diffuser (represented, say, by a Duct element named Diffuser) to the burner (represented by a Burner element named Brn), then the fluid output port on element Diffuser would be connected to the fluid input port on element Brn. The fuel would be introduced to the burner by instantiating a FuelStart element whose fuel output port would be connected to the fuel input port on element Brn.

Most elements create the ports they need for the user. These ports are automatically named. The naming convention for fluid, fuel, and shaft ports is given in Table 14 below.

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FluidInputPort</td>
<td>Fl_I</td>
</tr>
<tr>
<td>FluidOutputPort</td>
<td>Fl_O</td>
</tr>
<tr>
<td>FuelInputPort</td>
<td>Fu_I</td>
</tr>
<tr>
<td>FuelOutputPort</td>
<td>Fu_O</td>
</tr>
<tr>
<td>ShaftInputPort</td>
<td>Sh_I</td>
</tr>
<tr>
<td>ShaftOutputPort</td>
<td>Sh_O</td>
</tr>
</tbody>
</table>

Elements with multiple input and output ports are generally named Fl_I1, Fl_I2, Fl_O1, Fl_O2, etc., but consult the NPSS Reference Sheets to be sure. (The Splitter, for example, has primary and secondary fluid outlet ports named Fl_O1 and Fl_O2 respectively.) Shaft input ports are generally created and named by the user (see Section 4.3.1).

Ports are NPSS objects and contain a number of variables. The variables contained by fluid and fuel ports depends somewhat on the thermodynamics package being used, which is specified by the `setThermoPackage()` command (see Section 4.1 and the NPSS Reference Sheets). Port objects are children of the element on which they are created, and their variables are children of the port. Accessing the fuel/air ratio (variable name `FAR`) on the fluid output port of a Burner object named `Brn`, for example, would be done as follows:

```
Brn.Fl_O.FAR
```

4.3.1 Creating Ports

Some elements either allow or require the user to create ports. Ports are created in a similar way as elements are (see Section 4.2) by giving a keyword, a port name, and an optional instantiation block. The keywords used to create ports are summarized in the following table.
Table 15. Port Creation Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Port Type</th>
<th>Typical Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>FluidInputPort</td>
<td>fluid</td>
<td></td>
</tr>
<tr>
<td>FluidOutputPort</td>
<td>fluid</td>
<td></td>
</tr>
<tr>
<td>BleedInPort</td>
<td>fluid</td>
<td>Bleed</td>
</tr>
<tr>
<td>BleedOutlet</td>
<td>fluid</td>
<td>Bleed</td>
</tr>
<tr>
<td>InterStageBleedInPort</td>
<td>fluid</td>
<td>Turbine</td>
</tr>
<tr>
<td>InterStageBleedOutlet</td>
<td>fluid</td>
<td>Compressor</td>
</tr>
<tr>
<td>FuelInputPort</td>
<td>fuel</td>
<td></td>
</tr>
<tr>
<td>FuelOutputPort</td>
<td>fuel</td>
<td></td>
</tr>
<tr>
<td>ShaftInputPort</td>
<td>shaft</td>
<td>Shaft</td>
</tr>
<tr>
<td>ShaftOutletPort</td>
<td>shaft</td>
<td></td>
</tr>
<tr>
<td>DataInputPort</td>
<td>data</td>
<td>any</td>
</tr>
<tr>
<td>DataOutputPort</td>
<td>data</td>
<td>any</td>
</tr>
<tr>
<td>FileInPort</td>
<td>files</td>
<td>indirectly wrapped elements</td>
</tr>
<tr>
<td>FileOutletPort</td>
<td>files</td>
<td>indirectly wrapped elements</td>
</tr>
</tbody>
</table>

When using standard NPSS elements, the user does not normally need to create ports using FluidInputPort, FluidOutputPort, FuelInputPort, FuelOutputPort, or ShaftOutletPort. Ports that would be created with these commands are created automatically by the system. The commands are included in Table 15 for completeness. These commands are often needed, however, when the user creates new element types (see the Developer Guide).

A typical example of creating ports with some of these commands is given below.

```plaintext
CmpH {
    InterStageBleedOutlet Bld1out {
        fracBldW = 0.05;
    }
    InterStageBleedOutlet Bld2out, Bld3out;
    Bld2out {
        fracBldW = 0.002;
    }
    Bld3out.fracBldW = 0.003;
}
TrbH {
    InterStageBleedInPort Bld1in, Bld2in, Bld3in;
    Bld3in.diaPump = 1.2;
}
```

In the example, CmpH is assumed to be an existing Element object of type Compressor, and TrbH is an existing Element object of type Turbine. Variable fracBldW is an attribute of all InterStageBleedOutlet objects, and sets the bleed flow as a fraction of the parent object's inlet flow (bleeds are discussed in detail in Section 4.7). Note the following points:

- Objects instantiated within a block associated with another object become children of that object (Section 2.2.2.4). Ports Bld1out, Bld2out, and Bld3out are children of element CmpH. Ports Bld1in, Bld2in, and Bld3in are children of element TrbH.
- When no instantiation block is included, several ports can be created after the keyword by separating their names with commas and ending the statement with a semicolon.
- Attributes of existing ports can be set by associating a block with the port, or by direct assignment using dot notation.
- The user is free to name the port objects as desired, subject to the general naming rules given in Section 2.2.4.2.

DataPorts are discussed further in Section 4.3.3.
For information on which elements require or permit user-created ports, see the Element Reference Sheets in the NPSS Reference Sheets document. [Need a section that lists ports created for the user.] Some user-created ports can have an associated block of statements in which variables affecting the behavior of the port can be set. Once ports are created on an element, they must generally be linked together. This is the topic of the next section.

### 4.3.2 Linking Ports

Ports are connected together using the `linkPorts()` command. It has the following form:

```plaintext
linkPorts ( "port_name", "port_name", "station_name" );
```

Each link statement connects one output port to one input port, and it is customary to name the output port as the first argument and the input port as the second. The `linkPorts()` command causes a *station* to be created and assigned the given station name (more on this below). The same statement is used to link all the different types of ports. Port type and direction are checked to confirm that both ports are of the same type and output is connected to input. If an attempt is made to link two output ports, or a fluid to a mechanical port, an error message will be issued.

Some examples of linking ports follow:

```plaintext
linkPorts( "Diffuser.Fl_O", "Brn.Fl_I", "FL36" );  // fluid ports
linkPorts( "Fuel.Fu_O", "Brn.Fu_I", "FU36" );  // fuel ports
linkPorts( "CmpH.Bld1out", "TrbH.Bld1in", "Bld1" );  // fluid ports
```

The first example above links a pre-defined fluid output port on element *Diffuser* to a pre-defined fluid input port on element *Brn*, naming the resulting station "FL36". The second links a pre-defined fuel output port on element *Fuel* to a pre-defined fuel input port on element *Brn*, naming the resulting station "FU36". The third links a user-defined fluid output port (a bleed port) to a user-defined fluid input port (another bleed port), naming the resulting station "Bld1". See the example in Section 4.3.1 for the commands used to create the ports used in this third example. Shaft and data ports are linked in an identical fashion as the examples above.

Element objects representing rotating components, such as elements of type *Compressor* and *Turbine*, are automatically instantiated with shaft *output* ports (one per object). The user manually creates the required number of shaft *input* ports on an element object of type *Shaft*, and links the output ports of the components on the same shaft to the input ports of a single *Shaft* object. For example, to connect a *Compressor* object named *CmpH* to the same shaft as a *Turbine* object named *TrbH*, the following code would be required:

```plaintext
Element Shaft ShH {
    ShaftInputPort CmpH_SH, TrbH_SH;
}

linkPorts( "CmpH.Sh_O", "ShH.CmpH_SH", "CmpH_SH" );  // shaft ports
linkPorts( "TrbH.Sh_O", "ShH.TrbH_SH", "TrbH_SH" );  // shaft ports
```

To model two compressors and a turbine on a single shaft, three shaft input ports would need to be created on the Shaft element, and a third `linkPorts()` command given.

When two ports are connected, the set of fluid, fuel, shaft, or data properties at that location is called a *station*. It too is given a name – in the `linkPorts` command as indicated above. The fluid, fuel, shaft, or data properties at a location can generally be accessed using the station name or either of the two port names. For example, using the first example given above, *Diffuser.Fl_O.Tt*, *Brn.Fl_I.Tt*, and *FL36.Tt* all refer to the total temperature between element *Diffuser* and element *Brn*.

For fluid, fuel, and shaft ports, the following two rules apply:

- Every port must be linked to another port. Ports that have been created but not linked are detected when one attempts to run the model, and an error is generated unless the attributes `checkLink` or `checkAllLinks` are changed from their default values – see below.
- Each port must be linked to one and only one other port. For example, a fluid output port cannot be linked to more than one fluid input port. Likewise, a fluid input port cannot be linked to more than one fluid output port. Special elements exist for splitting and combining flows (see the NPSS Reference Sheets).
An alternative form of the linkPorts() command is linkPortsDesc(). It has the following form:

```c
linkPortsDesc( "port_name", "port_name", "station_name", "iDescription" );
```

The linkPortsDesc() function operates in a similar manner to linkPorts() with the following two major differences:

1) Will generate a default ‘station_name’ if the 3rd function argument is an empty string ("").
2) Has a 4th string argument to fill in the iDescription attribute for the first three arguments.

Example:

```c
linkPortsDesc("Amb.Fl_O", "D010.Fl_I","F0", "Free Steam Flow Entrance");
```

Links fluid port Amb.Fl_O to D010.Fl_I and creates the station name F0, sets Amb.Fl_O.iDescription, D010.Fl_I.iDescription and F0.iDescription equal to "Free Steam Flow Entrance".

Example:

```c
linkPortsDesc("Cmp025.Fl_O", "Hs030.Fl_I", ",", "Intermediate station");
```

Links fluid port Cmp025.Fl_O to Hs030.Fl_I, creates the default station name PORTDUM10001, sets Cmp025.Fl_O.iDescription, Hs030.Fl_I.iDescription and PORTDUM10001.iDescription equal to "Intermediate station".

The default station name is incremented as more linkPortsDesc() calls are made with an empty 3rd argument, i.e. PORTDUM10002, PORTDUM10003, etc.

The attribute checkAllLinks (default = 1 – check) exists at the global level. By setting this attribute to FALSE no port link checks will be made.

The attribute checkLink (default = 0 – do not check) exists at the port level. By setting this attribute to TRUE a check will be made for the individual port specified. This local attribute overrides the global attribute checkAllLinks for the local port only.

Truth Table for checkLink and checkAllLinks:

<table>
<thead>
<tr>
<th>checkLink</th>
<th>checkAllLinks</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (default)</td>
<td>0</td>
<td>No links checked</td>
</tr>
<tr>
<td>0 (default)</td>
<td>1 (default)</td>
<td>All links checked</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Selected local links checked</td>
</tr>
<tr>
<td>1</td>
<td>1 (default)</td>
<td>All links checked</td>
</tr>
</tbody>
</table>

An example follows:

```c
setThermoPackage("GasTbl");
FlowStart Fs; // create a FlowStart element
FlowEnd Fe; // create a FlowEnd element

// Do NOT link the FlowStart and FlowEnd links
//linkPorts ("Fs.Fl_O", "Fe.Fl_I", "Fs_out"); // linkPorts is commented out
checkAllLinks = FALSE; // turn off the check for all links
run(); // run the model, there should NOT be an error generated
```

With the linkPorts() command commented out in the example above, an error would normally be generated. However, by setting checkAllLinks equal to FALSE, no link checking is performed.
Another example:

```java
setThermoPackage("GasTbl");
FlowStart Fs; // create a FlowStart element
FlowEnd Fe; // create a FlowEnd element

// Do NOT link the FlowStart and FlowEnd links
//linkPorts ("Fs.Fl_O", "Fe.Fl_I", "Fs_out"); // linkPorts is commented out

checkAllLinks = FALSE; // turn off the check for all links
Fs.Fl_O.checkLink = TRUE; // local flag is true for the Fs.Fl_O link
run(); // run the model, there SHOULD be an error generated
```

With the `linkPorts()` command commented out in the example above, an error WILL be generated because the local attribute `Fs.Fl_O.checkLink` is set equal to `TRUE`, overriding the global attribute `checkAllLinks`, which was set to `FALSE`.

DataPorts follow somewhat different rules than fluid ports, as discussed in the following section.

### 4.3.3 DataPorts

DataPorts contain a single attribute named `val` that holds the value transferred by the port. The element on which the output port is created must assign a value to this attribute. Once the output port is linked to an input port (see Section 4.3.2), the attribute can be retrieved by the element on which the input port was created. An example follows:

```java
CmpH {
    DataOutputPort CDP;
    DataOutputPort CDT;
    void postexecute() {
        CDP.val = Fl_O.Pt;
        CDT.val = Fl_O.Tt;
    }
}

PERF {
    DataInputPort CDP;
    DataInputPort CDT;
}

TrbH {
    DataInputPort CDP;
}
```

```java
linkPorts { "CmpH.CDP", "PERF.CDP", "CDP1" };
linkPorts { "CmpH.CDP", "TrbH.CDP", "CDP2" };
```

In the example, two `DataOutputPorts` are created on existing object `CmpH` (an element of type `Compressor`). Two `DataInputPorts` are created on existing object `PERF` (a special element that collects overall cycle information). A `DataInputPort` is also created on object `TrbH` (an element of type `Turbine`). One of the `CmpH DataOutputPorts (CDP)` is linked to two `DataInputPorts`: one on `PERF`, the other on `TrbH`. Element `PERF` can access the compressor discharge pressure as `CDP.val` (the complete path would be `PERF.CDP.val`), and element `TrbH` can access the value as `CDP.val` (the complete path would be `TrbH.CDP.val`). The remaining `DataOutputPort` on `CmpH`, and the remaining `DataInputPort` on `PERF` are not linked.

The example illustrates the following special rules for DataPorts:

- No error is generated when DataPorts are not linked. Values are transmitted from one DataPort to another, however, only if they are linked. The value in unlinked `DataInputPort` `PERF.CDT.val` is zero.
DataOutputPorts may be linked to more than one DataInputPort. DataInputPorts, however, can be linked to only one DataOutputPort.

DataPorts are useful for writing general elements that use a piece of information that can bear different names in different models. For example, element PERF might display the overall pressure ratio of a cycle. The name and location of the last compressor's discharge varies from model to model. If DataInputPort PERF.CDP is correctly linked to a DataOutputPort on the last compressor, however, it can always access the highest pressure in the cycle using the name CDP.val.

DataOutputPorts support a user-defined preexecute() function, so that the user may collect the data just before it is to be transferred from current model contents if necessary. Similarly, DataInputPorts support a user-defined postexecute() function which may be used to distribute data contents into the model if needed.

4.3.4 FilePorts

FilePorts are used to transfer files from one component to another. They are typically used with components that wrap external codes indirectly, i.e., codes that interface with each other only through file input and output.

FilePorts contain the following attributes:

- filenames[]: A list of names of files to be transferred
- directory: The parent directory containing the files in filenames. Defaults to the current working directory of the process.
- deleteFilesOnExit: If TRUE, the FilePort will delete the files in filenames when the port itself is deleted. Default value is FALSE.
- transferType: The style of transfer, either PUSH or PULL. PUSH is the default. Typically this is only changed to handle transfers through firewalls.
- conditionalRecv: (FileInputPorts) If TRUE, only invoke postexecute() if one or more files have changed, as indicated by their modification timestamp. If FALSE, postexecute() is always invoked. Default TRUE.
- newFileFlags[]: (FileInputPorts) Integer flags indicating which files in filenames have changed according to their modification timestamp.
- autoExecute: (FileOutputPorts) If TRUE, always send files whenever execute() is invoked. If FALSE, only send files when explicitly requested via the execute() method (i.e., not via the solverSequence).
- conditionalSend: (FileOutputPorts) If TRUE, only send those files which have changed, as indicated by their modification timestamp. If FALSE, always send all files. Default TRUE.

In most port types, data is pulled from the output port to the input port, but in FilePorts, files may be pushed from the FileOutPort to the FileInPort or pulled. Pushing means that the actual transfer of files occurs when the component containing the FileOutPort is executed, immediately after the component's postexecute function, if it exists. Pulling means that the actual transfer of files occurs when the component containing the FileInPort is executed, immediately after the component's preexecute function, if it exists. FileOutPorts keep track of the modification times for their files so that they won't transfer them if they haven't changed since the last time they were transferred. Like DataOutputPorts, FileOutPorts may be connected to multiple FileInPorts, but each FileInPort may be connected to only one FileOutPort. It is not an error if FilePorts are not connected. If filenames in the FileInPort is not empty and the number of entries in filenames in the FileOutPort does not match the number of entries in filenames in the FileInPort, an error will be reported. The number of entries must match because the entries in filenames in the FileInPort are used as destination file names for the files being transferred from the FileOutPort. If filenames is empty in the FileInPort, the file names from the FileOutPort are used for the destination files. Also, if the filenames are the same, then directory must be different, which means that it must either be set explicitly to be different, or the two parent components that contain the FilePorts must be running in different processes with different working directories. The following is an example of FilePort usage:
// a local component
MyComponent {
    FileOutPort FOP {
        filenames = { "a.out", "b.out", "c.out" };  
        directory = "mydir";
    }
}

// another local component
MyOtherComponent {
    FileInPort FIP {
        filenames = { "1.in", "2.in", "3.in" }; 
        directory = "somedir"; // directory could be the same since 
        // filenames are different, but we'll make it 
        // different anyway
        deleteFilesOnExit = TRUE; // clean up files when we exit
    }
}

// this is an external component that is running in another process
// (with a different working directory) or on another machine, so 
// we'll assume here that our working directory is different than that of 
// MyComponent.FOP. We also don't set the value of filenames, so our 
// filenames will be the same as MyComponent.FOP.
MyExternalComponent {
    FileInPort FIP {
        deleteFilesOnExit = TRUE;
    }
}

linkPorts ( "MyComponent.FOP", "MyOtherComponent.FIP", "files1" );
linkPorts ( "MyComponent.FOP", "MyExternalComponent.FIP", "files2" );

Now, when MyComponent is executed, MyComponent.FOP will transfer its files to MyOtherComponent.FIP and 
MyExternalComponent.FIP, provided that the files have changed since the last time they were sent. 

FilePorts also have a user-accessible execute() function, so that the user can explicitly trigger the file transfer 
outside of a normal component execution.

FileOutputPorts support a user-defined preexecute() function, so that the user may create the file just before it is 
to be transferred from current model contents if necessary. Similarly, FileInputPorts support a user-defined 
postexecute() function which may be used to read file contents into the model if needed.

4.4 Subelements
As mentioned in Sections 1.5.1 and 4.2, elements are the primary building blocks used to model components such as 
compressors, turbines, burners, etc. Elements can reside in the top-level assembly (global scope), not being the 
children of any other object. Subelements are always active as the children of other elements or subelements. They 
assist their parent object by performing certain major calculations on behalf of the parent. For example, compressor 
and turbine maps are implemented in NPSS using subelements.

Subelement objects are instantiated much like element objects are, using the following items in order:
1. Keyword Subelement (optional)
2. The specific subelement type
3. The object name
4. An optional block of statements (enclosed in curly braces) executed when the object is instantiated. This block 
of statements can instantiate other objects that become children of the subelement object.
Since active subelements are always the child of another element or subelement, they are usually instantiated that way. Recall (Section 2.2.2.4) that an object instantiated in a block associated with another object becomes the child of that object. For example:

```plaintext
Element Turbine TrbH {
    Subelement TurbinePRmap S_map {
        PRmapDes = 4.975;
        NpMapDes = 100.0;
    }
    Subelement TurbineReynoldsEffects S_Re {
        ...
    } // end S_Re
    } // end S_map
} //END TrbH
```

Notice that a subelement object can be the child of another subelement object (the TurbineReynoldsEffects subelement object named S_Re is the child of the TurbinePRmap subelement object named S_map, which itself is the child of the Turbine element named TrbH. Note that NPSS does not come with a standard TurbineReynoldsEffects subelement, but if the user happened to create such a subelement it could be plugged into the S_Re socket as shown above.). The PRmapDes and NpMapDes variables are attributes of all TurbinePRmap subelements, and values may be assigned in the block of statements accompanying the instantiation of the subelement. These variable values could have been defined outside the block, using their path names (e.g., `TrbH.S_map.PRmapDes = 4.975;`). Another way to set such attributes after an element or subelement object has been instantiated is to create another block of lines associated with the object, as in the following example:

```plaintext
TrbH.S_map {
    PRmapDes = 4.975;
    NpMapDes = 100.0;
}
```

If users wish, when they instantiate a subelement they may designate a version number using the string variable "version" (as discussed in Section 4.2). Note also that the attributes `description` and `iDescription` exist for complex objects such as subelements. For such objects, `description` is intended to explain the general purpose of the object `type`, and is usually given a predefined default value by the subelement creator. Attribute `iDescription` is intended to explain the purpose of a specific `instance` of a type. It can be supplied by the user to help document a model.

In addition to using pre-written standard subelements, NPSS allows the user to create entirely new subelement types which can then be instantiated and used in a model. Details on how to do this are found in the Developer’s Guide.

Subelements can be associated only with certain parent objects, and must be given a predetermined name in order to work properly with their parent. This is because subelements are designed to fit into sockets, discussed in the next section.

### 4.5 Sockets

Whenever an element or subelement can accept a subelement as a child object, a socket exists to plug in that subelement. In the example in the preceding section, element `TrbH` could accept subelement `S_map` because all Turbine elements contain a socket that can accept a subelement of type `TurbinePRmap`. It is most common to insert a subelement into a socket (as in the example above); but sockets can be designed to accept functions or tables as well. Sockets are created and named automatically when their parent object (an element or subelement) is instantiated.

The following points are important to keep in mind regarding sockets:

- Not all elements and subelements have sockets.
- Some elements and subelements have more than one socket, each having a unique name.
- Most sockets do not have to be filled. Sockets can be modified, however, such that they must be filled (see attribute `required` in Table 16 below).
• Each socket can accept only one object, and that object must be of a specific type, which is determined by the socket (see attribute `socketType` in Table 16 below).
• For sockets that accept subelements, sometimes two or more standard NPSS subelements are available for use in that socket. This affords the user a choice of more than one method to calculate the information the socket is designed to provide to its parent object.

4.5.1 Socket Attributes

Sockets are themselves objects residing within their parent element or subelement. As such, all sockets have certain attributes that can be accessed by the user. The following table summarizes these attributes.

<table>
<thead>
<tr>
<th>Socket Attribute</th>
<th>Type</th>
<th>User Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>string</td>
<td>set/get</td>
<td>Describes the socket.</td>
</tr>
<tr>
<td>baseType</td>
<td>string</td>
<td>get only</td>
<td>Immediate base type. For sockets this is VariableContainer.</td>
</tr>
<tr>
<td>required</td>
<td>Boolean</td>
<td>set/get</td>
<td>If set to TRUE, an error message is produced if the socket is empty when the model is run. Default value is FALSE.</td>
</tr>
<tr>
<td>allowedValues</td>
<td>string array</td>
<td>set/get</td>
<td>Contains the names of the variables that an object in the socket is permitted to set.</td>
</tr>
<tr>
<td>argTypes</td>
<td>string array</td>
<td>set/get</td>
<td>Defines the argument types required for a function or table inserted in the socket.</td>
</tr>
<tr>
<td>returnType</td>
<td>string</td>
<td>set/get</td>
<td>Defines the return type of a function inserted in the socket.</td>
</tr>
<tr>
<td>socketType</td>
<td>string</td>
<td>set/get</td>
<td>Defines what type of object can be inserted into the socket.</td>
</tr>
</tbody>
</table>

Attributes `argTypes` and `returnType` are discussed further in Section 4.5.4.

As will be discussed in Section 4.5.3, objects inserted into a socket are always treated as having the same name as the socket. Attributes in both the socket and the object that fills it are then accessed as `parent_name.socket_name.attribute_name`. If the object and the socket have identically named objects, there is potential for ambiguity. For attributes, this normally occurs only for the description and baseType attributes. NPSS handles these variables as follows:

• When a socket is empty, `parent_name.socket_name.description` and `parent_name.socket_name.baseType` access attributes of the socket.
• When a socket is filled, `parent_name.socket_name.description` and `parent_name.socket_name.baseType` access attributes of the object filling the socket (i.e. the corresponding attributes of the socket are hidden).
• If a socket is filled with an object whose `description` attribute is empty, the socket's description is copied to the object's.

Notice that sockets do not have an iDescription attribute, so it always pertains to the inserted object.

The user should not create variables named `required`, `allowedValues`, `argTypes`, `returnType`, or `socketType` in an object filling a socket. The naming conflicts arising between the socket and the object filling it can produce unexpected results.

A number of important member functions (those described in Section 12.1.1 for example) exist in both Socket objects and in the objects that fill sockets. Unlike attributes description and baseType, the socket member functions are not hidden once the socket is filled. Notation using ".child" is provided to distinguish the socket from the object filling it:
• \textit{parent\_name.\ socket\_name.\ function\_name} accesses a member function of the \textit{socket}.
• \textit{parent\_name.\ socket\_name.chld.function\_name} accesses a member function of the \textit{object filling the socket}.

In general, then, the following rules may be applied:

• When accessing attributes, the only attributes of importance after a socket has been filled are those belonging to the object filling the socket. Access them using the simpler syntax \textit{parent\_name.\ socket\_name.attribute\_name}.
• When accessing the member functions of an object inserted into a socket, use \textit{parent\_name.\ socket\_name.chld.function\_name}.

The \textit{.chld} syntax is also important when using the \texttt{move()} command to insert an object into a socket (see Section 4.11).

Although only knowledgeable users should do so, the values of the socket attributes listed in Table 16 can be changed by the user before an object is inserted into a socket (except for \texttt{baseType}, which the user can never change). These attributes are primarily used at the time a user attempts to insert an object into a socket to test the validity of the object. After an object is successfully inserted into a socket, however, the socket attributes are no longer used. The system currently allows the user to change socket attributes after the socket is filled, but these changes do not affect the interaction between the socket and the object filling it. It is recommended that the user refrain from changing socket attributes once the socket has been filled, and exercise caution in changing them before the socket is filled.

### 4.5.2 Determining Allowable Socket/Subelement Combinations

As noted above, each socket accepts only certain objects that match its \texttt{socketType} attribute. Every NPSS object has a member function named \texttt{hasInterface()} that accepts a single string argument. If the argument matches an interface the object supports, \texttt{hasInterface()} returns integer 1 (for Boolean "true"). If the argument does not match, \texttt{hasInterface()} returns integer 0 (for Boolean "false"). When a user tries to insert an object in a socket, the socket calls that object's \texttt{hasInterface()} function with the socket's \texttt{socketType} attribute as the argument. For example, each \texttt{Turbine} element has a socket named \texttt{S\_map}. If \texttt{TrbH} is a \texttt{Turbine} element, then \texttt{TrbH.S\_map.socketType} equals "TURBINE\_MAP". Suppose subelement \texttt{TrbH\_map} is a subelement object of type \texttt{TurbinePRmap}. If the user tries to insert this object into socket \texttt{TrbH.S\_map}, the socket will call

\begin{verbatim}
    TrbH_map.hasInterface( "TURBINE\_MAP" )
\end{verbatim}

Only if the function returns 1, or "True," the insertion will be allowed.

The interfaces an object supports can be found by calling the object's \texttt{listInterfaces()} member function. For example:

\begin{verbatim}
    TrbH_map.listInterfaces()
\end{verbatim}

would return the following string array: \{"TurbinePRmap","TURBINE\_MAP","Subelement","ElementBase", "VariableContainer","VariableOnlyContainer","VCInterface"\}. Several of these items reflect the way subelements inherit properties from other NPSS objects. The significant point at hand, however, is that the string "TURBINE\_MAP" is among the interfaces supported by objects of type \texttt{TurbinePRmap}. They can therefore be inserted into \texttt{Turbine} element \texttt{S\_map} sockets, which require type "TURBINE\_MAP".

A desired interface can be forced on an object by calling its \texttt{addInterface()} function, supplying the desired interface as a single string argument. This feature should be used with caution by experienced users only. It is often necessary when writing new subelement types as discussed in the \textit{Developer Guide}.

Tables showing NPSS Element and Subelement objects, the sockets they contain, the \texttt{socketType} required by each socket, and the Subelement objects supporting each socket type are found in the \textit{NPSS Reference Sheets}.

### 4.5.3 Inserting Subelements into Sockets

The simplest way to insert a \texttt{Subelement} object into a socket is to:
1. Instantiate the Subelement object within a block associated with its intended parent.
2. Give the Subelement object the same name as its intended socket.

For example:

```plaintext
Element Turbine TrbH {
    Subelement TurbinePRmap S_map {
        Subelement TurbineReynoldsEffects S_Re {
            ...
        } // end S_Re
    } // end S_map
} //end TrbH
```

Turbine elements contain a socket named S_map. Therefore, to insert a subelement of type TurbinePRmap into this socket, the TurbinePRmap object is instantiated within a block associated with the Turbine object (in this case the instantiation block), and is named S_map. Likewise, TurbinePRmap objects contain a socket named S_Re. To insert a subelement of type TurbineReynoldsEffects into this socket, the TurbineReynoldsEffects object is instantiated within a block associated with the TurbinePRmap object, and is named S_Re. Note that NPSS does not come with a standard TurbineReynoldsEffects subelement, but if the user happened to create such a subelement, it could be plugged into the S_Re socket as shown above. As noted in Section 2.2.2.4, a block can be associated with an object after its instantiation.

There is another, more complicated, way to insert subelements into sockets using the move() function. This is discussed in Section 4.11.

### 4.5.4 Inserting Functions and Tables into Sockets

Functions support the following interfaces (as determined by the listInterfaces() command, Section 4.5.2):

- UserFunction
- Function
- VCInterface

Tables support the following interfaces:

- Table
- Function
- VCInterface

Notice that a table supports the "Function" interface. As noted in Section 2.2.7, tables are really special functions that always return a real value.

A socket with socketType = "Function" can accept either a function or a table. A socket with socketType = "Table" can only accept a table. As noted in Table 16, sockets have two attributes that specify information about functions and tables that can be inserted into the socket. These are argTypes and returnType. Attribute argTypes is a string array containing the types ("real", "int", "string", etc.) of the arguments, in order, supplied to the function or table. Attribute returnType is a string variable that specifies the type returned by the function or table. For ordinary tables, this must be "real". For stacked tables, it must be "real[]". These attributes are used to test prospective functions or tables being inserted into the socket. They are ignored for sockets designed to accept subelements.

Functions and tables are inserted into sockets designed to accept them in the same way subelements are. The simplest way is to define the function or table within a block associated with its intended parent object, and give it the same name as its intended socket (see Section 4.5.3). The function or table can also be created separately from the parent object, and moved to fill the socket using the move() command (see Section 4.11).

There are no standard NPSS elements or subelements that contain sockets designed to accept functions or tables. Notice that whereas the types of arguments supplied to a function or table filling a socket are specified, nowhere can the user specify the specific variable names supplied as arguments. For this reason, the user cannot take a standard NPSS element or subelement with a socket designed to accept a subelement, modify the socket attributes, and fill the socket with a function or table instead. Rather, the user must write a custom element or subelement with sockets designed to accept functions or tables.
that accept functions or tables, and logic to handle the specific functions or tables envisioned. The writing of custom
elements and subelements is discussed in the *Developers Guide*.

### 4.5.5 list() Function and Sockets

Function list() along with other pre-programmed functions available to the user, is discussed in Chapter 12.
Briefly, list() is a member function of all NPSS objects (including the top-level assembly) that returns a string
array of all objects contained within the object on which list() was called that match certain criteria. This section
shows how to use list() to gain information about an object's sockets.

Calling list() on an element or subelement as follows generates a list of all the sockets in the object, and any
subelements of the object.

```cpp
CmpH.list( "Socket", TRUE )
```

In the example, CmpH is an element object of type *Compressor*. Typical output from this command would be a
string array like the following:

```cpp
  "CmpH.S_map.TB_Wc", "CmpH.S_Qhx" }
```

This indicates that elements of type *Compressor* have two sockets named S_map and S_Qhx. In this particular case,
socket CmpH.S_map must already be filled with a subelement, because four additional sockets were found in that the
S_map subelement: S_Re, TB_eff, TB_PR, and TB_Wc. The list does not indicate whether those sockets are filled.
Nor does it indicate whether CmpH.S_Qhx is filled.

Sockets have a member function (see Section 12.1.18) named isEmpty() that returns integer 1 for "true" if the
socket is empty and 0 for "false" if it is filled. The following function, then, returns a string array listing the empty
sockets on and below an object.

```cpp
string[] findEmptySockets( string Obj ) {
  string tempList[], returnList[];
  tempList = Obj->list("Socket", TRUE);
  int i;
  returnList = { };  
  for ( i=0; i<tempList.entries(); i++ ) {
    if ( tempList[i]->isEmpty() ) {
      returnList.append(tempList[i]);
    }
  }
  return returnList;
}
```

In the example above, `findEmptySockets( "CmpH" )` might return `{ "CmpH.S_map.S_Re",
"CmpH.S_Qhx" }`.

Function list() could also be used in a function to find all sockets expecting a certain type of object:

```cpp
string[] findSocketsOfType( string Obj, string type ) {
  string tempList[], returnlist[];
  tempList = list("Socket", TRUE);
  int i;
  returnList = { };  
  for ( i=0; i<tempList.entries(); i++ ) {
    if ( tempList[i]->socketType == type ) {
      returnList.append(tempList[i]);
    }
  }
  return returnList;
}
```

In the example, `findSocketsOfType( "CmpH", "COMPRESSOR_MAP" )` would return `{ "CmpH.S_map" }.`
4.6 Assemblies

As introduced in Section 1.5.3, an assembly is a group of elements (together with any subelements or other objects that are children of those elements) that is treated by the model as though it were a single element.

Assemblies are instantiated much as ordinary elements are (see Section 4.2). In fact they are instantiated as an element of type Assembly. The following items are provided in order:

1. Keyword Element (optional)
2. The specific element type: Assembly
3. The assembly name
4. An optional block of statements (enclosed in curly braces) executed when the assembly is instantiated. This block of statements is usually used to instantiate the elements contained within the assembly.

The following example instantiates an assembly named Core:

```plaintext
Element Assembly Core {
  Element Compressor CmpH {
    #include "hpc.map"
  }
  Element FuelStart Fuel;
  Element Burner Brn;
  Element Turbine TrbH {
    #include "hpt.map"
  }
  Element Shaft ShH {
    ShaftInputPort CmpH_SH, TrbH_SH;
  }
  linkPorts( "CmpH.Fl_O", "Brn.Fl_I", "FL3" );
  linkPorts( "Fuel.Fu_O", "Brn.Fu_I", "FU3" );
  linkPorts( "Brn.Fl_O", "TrbH.Fl_I", "FL4" );
  linkPorts( "CmpH.Sh_O", "ShH.CmpH_SH", "CmpH_SH" );
  linkPorts( "TrbH.Sh_O", "ShH.TrbH_SH", "TrbH_SH" );
} // end Core
```

This groups together all the components and links associated with the core of a fan engine. When incorporated in the overall model, the assembly will be treated as though it were a single element.

As noted in Section 1.5.3, assemblies can contain other assemblies. In fact, "global scope" in NPSS is actually an assembly created for the user, known as the top-level assembly. The user seldom needs to be aware of this since usually no path is necessary to access objects belonging to the top-level assembly (although one can be supplied – see Sections 2.2.2.1 and 2.2.4.3). The only special property of the top-level assembly is that it must (and always does) have a solver. The top-level assembly's solver is automatically instantiated for the user, but the user must set it up properly (see Section 4.8). Assemblies created inside the top-level assembly may or may not have solvers, at the user's discretion.

Like ordinary elements, assemblies need ports in order to communicate with other elements or assemblies. Normally, these ports already exist as part of elements within the assembly. To tell NPSS that an element's port is to be considered a port on the entire assembly, that port is promoted. The promotePort() command has the following form:

```plaintext
promotePort { "element_port_name", "assembly_port_name" };
```
Continuing the example above:

```csharp
Core {
    promotePort("CmpH.Fl_I", "Fl_I");
    promotePort("TrbH.Fl_O", "Fl_O");
}
```

Once a port has been promoted, it can be referred to as a member object of the assembly, and connected with another element’s or assembly’s port using the linkPorts() command. The following example links the primary fluid output port of a Splitter element named Spl to the fluid input port of the Core assembly, and the fluid output port of the assembly to the fluid input port of a Turbine element named TrbL.

```csharp
linkPorts( "Spl.Fl_O1", "Core.Fl_I", "FL21" );
linkPorts( "Core.Fl_O", "TrbL.Fl_I", "FL5" );
```

Note that since the promotePort() commands in the previous example above were given inside a block associated with assembly Core, the names given to the ports (Fl_I and Fl_O) had no path given. Whenever these ports are referenced outside the scope of the assembly, however, their path (Core.Fl_I and Core.Fl_O) must be given, as in the linkPorts() examples immediately above. The same is true for all objects inside the assembly. Within the assembly, compressor CmpH can be referenced as simply CmpH. Outside the assembly, it must be referenced as Core.CmpH.

The Assembly class has a counter-function to promotePort called unpromotePort(portName), which unlinks all of the connections to the promoted port and un-resisters the port from the Assembly to which it had previously been promoted. To unpromote the ports that had been promoted in the above example, the code would look like this:

```csharp
Core{
    unpromotePort("FL_I");
    unpromotePort("Fl_O");
}
```

### 4.6.1 Order of Execution in Assemblies

Control over order of execution was mentioned in Section 4.2. Every assembly, including the top-level assembly, has an Executive which contains three string array attributes that control the order of execution of objects within the assembly. These are summarized in the table that follows.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>presolverSequence</td>
<td>An alias for solver.preExecutionSequence, which is a list of objects to be executed, in order, before the solver executes. Defaults to empty.</td>
</tr>
<tr>
<td>solverSequence</td>
<td>An alias for solver.executionSequence, which is a list of objects in the order they are to be executed by the solver. If there is no solver, the objects in solverSequence are executed once, in the order given. Defaults to all the elements instantiated, in the order in which they are instantiated.</td>
</tr>
<tr>
<td>postsolverSequence</td>
<td>An alias for solver.postExecutionSequence, which is a list of objects to be executed, in order, after the solver executes. Defaults to empty.</td>
</tr>
</tbody>
</table>

Normally these lists contain the names of elements or assemblies. The execution of an element automatically executes all its subelements, so subelements should not be added to the lists. It is permissible, however, to add other objects such as functions or DataViewers to the lists. The execution of an assembly automatically executes the objects in its Executive’s execution sequences lists, even if it does not have a solver.
The model containing the Core assembly given as an example in Section 4.6 might have the following objects in attribute executionSequence of its top-level assembly’s Executive.

1. "Amb" (Ambient element establishing ambient conditions)
2. "InletStart" (Connects Ambient element and initiates the flow stream)
3. "In1" (Inlet element establishing conditions at the fan inlet)
4. "CmpFSec" (Compressor element modeling the fan)
5. "Spl" (Splitter element dividing flow between core and bypass)
6. "Core" (Assembly containing the core components)
7. "TrbL" (Turbine element modeling the low-pressure turbine)
8. "NozPri" (Nozzle element modeling the primary exhaust nozzle)
9. "FePri" (FlowEnd element terminating the flow from the primary nozzle)
10. "Bypass" (Duct element modeling the bypass duct)
11. "SecNoz" (Nozzle element modeling the secondary nozzle)
12. "FeSec" (FlowEnd element terminating the flow from the secondary nozzle)
13. "ShL" (Shaft element modeling the low-pressure shaft)
14. "PERF" (User-written element calculating overall performance measures)

Notice that the Core assembly is listed as a single element. When assembly Core is executed, even though it has no separate solver, the objects in its executionSequence attribute will be executed, which would be as follows (see the example code in Section 4.6):

1. "CmpH" (Compressor element modeling the high-pressure compressor)
2. "Fuel" (FuelStart element establishing fuel properties)
3. "Brn" (Burner element modeling the primary burner)
4. "TrbH" (Turbine element modeling the high-pressure turbine)
5. "ShH" (Shaft element modeling the high-pressure shaft)

The solver for the top-level assembly executes assembly Core once every pass. Thus all the objects within Core are executed once per pass, and the entire model can converge correctly.

In the example above, if element PERF calculated values to be used for final output only, it could be removed from executionSequence and placed in postExecutionSequence instead, thus saving the time required for its execution every pass. This is often done with DataView objects (see Section 4.9.2). The presence of PERF in executionSequence, however, allows its values to be used as one of the criteria for solver convergence. For example, the model can be run to match a desired total net thrust, calculated by PERF by summing the thrusts associated with the primary and secondary nozzles.

As element and assembly objects are instantiated, their names are added to the executionSequence array of the Executive in the assembly that contains them. The default execution sequence of an assembly, therefore, is the order in which its elements were instantiated. With some care, the user can avoid the need to manually edit the executionSequence array. Success is generally had by insuring that output ports are executed before the input ports to which they are linked. Thus:

- As much as possible, create elements in normal flow order, following each stream from its start to its end.
- Create elements that are the source of bleeds before the elements that receive those bleeds.
• Create a **FuelStart** element before each burner since the **FuelStart** element's fuel output port must be linked to the burner's fuel input port.

• Create **Shaft** elements *after* the rotating components to be connected by the shaft. This is because the rotating components have shaft output ports that must be linked to shaft input ports on the **Shaft** element.

• Create elements with data output ports before the elements with corresponding data input ports.

• Elements without ports, such as **PERF** in the example above, and objects such as DataViewers, should normally be executed last, after all cycle information is available.

If the user finds it necessary to alter the contents of **preExecutionSequence**, **executionSequence**, or **postExecutionSequence**, the array manipulation commands mentioned in Sections 2.2.4.6 and 12.1.2 are often helpful. An example is given in Section 4.12.

The parameter **autoAddToSolvSeq** (defaults to TRUE) can also be set by the user. The autosolver checks this attribute to see what elements should be included in the solver sequence.

The following example illustrates the use of **autoAddToSolvSeq**. In element **Comp**, **autoAddToSolvSeq** has been set to FALSE. This excludes the element from the solver sequence.

```cpp
setThermoPackage( "GasTbl" );

Element FlowStart S1{
    W = 10;
    Pt = 14;
    Tt = 500;
}

Element Compressor Comp{
    PR = 10;
    autoAddToSolvSeq = FALSE;
}

Element FlowEnd E1{
}

Element Shaft Shaft{
    ShaftInputPort Sh_I;
    Nmech = 10000;
}

linkPorts( "S1.Fl_O", "Comp.Fl_I", "F1" );
linkPorts( "Comp.Fl_O", "E1.Fl_I", "F2" );
linkPorts( "Comp.Sh_O", "Shaft.Sh_I", "link" );

Comp.Sh_0.trq = 200;
Comp.Sh_0.inertia = 100;

cout << "Execution Sequence Comp Removed: " << solver.executionSequence << endl;

This will produce the following output:

Execution Sequence Comp Removed: {"S1", "E1", "Shaft"}
```

Issues pertaining to solvers and assemblies are discussed in Section 4.8.5.
4.7 Bleed Usage

Certain NPSS elements are designed to allow bleed flows between them. Compressor and Bleed elements can be bleed sources; Turbine and Bleed elements can be bleed sinks. Bleed flows are handled the same way as primary flows: by creating bleed ports and linking them together. Any number of bleed ports can be created on an element designed to accept them. NPSS provides some built-in functions to simplify the creation of bleed flows (see Section 4.7.5).

The following sections will illustrate and describe use of some of the variables associated with bleed ports. The user should consult the NPSS Reference Sheets, however, for complete details.

4.7.1 Compressor Element Bleed Extraction

Compressor elements are designed to be bleed sources, and consequently permit creation of fluid output ports for bleed flows. As illustrated previously in Section 4.3.1, bleed ports on a Compressor element are created using the InterStageBleedOutPort command:

```
CmpH {
    InterStageBleedOutPort Bld1out {
        fracBldW = 0.05;
    }
    InterStageBleedOutPort Bld2out, Bld3out;
    Bld2out {
        fracBldW = 0.002;
    }
    Bld3out.fracBldW = 0.003;
}
```

A compressor InterStageBleedOutPort has three input variables:

- **fracBldW** Fraction of the compressor inlet flow that is extracted as bleed.
  
  \[ \text{fracBldW} = \frac{\text{bleed flow}}{\text{compressor inlet flow}} \]

- **fracBldP** Fraction of the total pressure rise in the bleed flow. If total pressure is \( P \), then:
  
  \[ \text{fracBldP} = \frac{(\text{bleed } P - \text{compressor inlet } P)}{(\text{compressor exit } P - \text{compressor inlet } P)} \]
  
  - 0 for bleed extracted from the compressor inlet
  
  - 1 for bleed extracted from the compressor exit

- **fracBldWork** Fraction of the total enthalpy rise in the bleed flow. Analagous to fracBldP.

4.7.2 Turbine Element Bleed Entry

Turbine elements are designed to be bleed sinks, and consequently permit creation of fluid input ports for bleed flows. As illustrated previously in Section 4.3.1, bleed ports on a Turbine element are created using the InterstageBleedInPort command:

```
TrbH {
    InterStageBleedInPort Bld1in, Bld2in, Bld3in;
    Bld3in.diaPump = 1.2;
}
```

As documented in the NPSS Reference Sheets, a turbine InterstageBleedInPort has two input variables:
Fraction of the total pressure "rise" associated with the turbine that acts as the sink pressure for the bleed.

\[ \text{Pfrac} = \frac{\text{bleed P} - \text{turbine exit P}}{\text{turbine inlet P} - \text{turbine exit P}} \]

\[ \text{Pfrac} = 1 \text{ for bleed introduced at the turbine inlet} \]

\[ \text{Pfrac} = 0 \text{ for bleed introduced at the turbine exit} \]

Effective pumping diameter used to calculate the power used to accelerate the bleed flow to the rotational speed of the turbine blades.

### 4.7.3 Bleed Element Extraction and Entry

The **Bleed** element serves as a utility element that allows the user to create bleed input and output ports. Like most NPSS elements, the **Bleed** element has one predefined fluid inlet port (named **F1_I**) and one predefined fluid outlet port (named **F1_O**), both for the primary flow. Bleed inlet and outlet ports can be created by the user using the **BleedInPort** and **BleedOutPort** commands illustrated below. Both kinds of ports can be created on a single **Bleed** object, and multiple ports of either kind can be created on a single **Bleed** object.

```
Element Bleed DiffBld {
  BleedOutPort TrbHV1out {
    fracW  = 0.08;
    Pscale = 0.95;
    hscale = 1.00;
  }
}
Element Bleed TrbHV1 {
  BleedInPort TrbHV1in;
}
```

The bleed inflow port has two input variables that are not commonly used. They are discussed in the **Turbine** element reference sheet.

The bleed outflow port has three input variables:

- **fracW**  
  Ratio of bleed flow to Bleed element inlet flow.

- **Hscale**  
  Ratio of bleed enthalpy to primary flow enthalpy.

- **Pscale**  
  Ratio of bleed pressure to primary flow pressure.

In the example the bleed outflow pressure is 5% lower than the primary flow, and the bleed outflow enthalpy is equal to that of the primary flow. The pressure scaling has no effect on the primary flow. Enthalpy scaling, however, is reflected in the primary flow. Scaling the bleed enthalpy up will lower the outgoing primary stream enthalpy to satisfy conservation of energy.

### 4.7.4 Linking Bleed Flows

Once bleed ports have been created, they are linked just as other fluid ports are. For example:

```
linkPorts( "DiffBld.TrbHV1out", "TrbHV1.TrbHV1in", "TrbHV1bld" );
```

Only fluid output ports can be linked to fluid input ports.

### 4.7.5 Bleed User Functions

The previous sections illustrate how to create bleeds using a three-step process:

1. Create the bleed source port.
2. Create the bleed sink port.
3. Link the two together using the linkPorts() function.

NPSS provides some functions (called macros, which are nothing more than predefined global functions the user can employ to simplify this process by creating the source and sink ports, and linking them together, all in one step. Only two element types can be the source of bleed outflows: Compressor and Bleed. Only two element types can receive bleed inflows: Turbine and Bleed. There are thus four possible bleed combinations, and four macros to handle them:

- linkBleedCT() for bleeds from a Compressor to a Turbine
- linkBleedCB() for bleeds from a Compressor to a Bleed element
- linkBleedBT() for bleeds from a Bleed element to a Turbine
- linkBleedBB() for bleeds from one Bleed element to another

In order to use these functions, the file bleed_macros.fnc must be included in one of the NPSS input files before any of the functions is called.

```c
#include <bleed_macros.fnc>
```

To create a bleed between a compressor and a turbine, use linkBleedCT() as follows:

```c
linkBleedCT("compr_name", "turb_name", fracBldW, fracBldP, fracBldWork, Pfract, diaPump, "port_name");
```

The five arguments before "port_name" are the bleed port variables discussed in sections 4.7.1 and 4.7.2. For example:

```c
linkBleedCT("CmpH", "TrbH", 0.02, 0.85, 0.80, 0.20, 1.2, "TrbHlpBld");
```

To create a bleed between a compressor and a Bleed element, use linkBleedCB() as follows:

```c
linkBleedCB("compr_name", "bld_elem_name", fracBldW, fracBldP, fracBldWork, "port_name");
```

The three arguments before "port_name" are discussed in Section 4.7.1. For example:

```c
linkBleedCB("CmpH", "TrbHV1", 0.01, 1.00, 1.00, "TrbHhpBld");
```

To create a bleed between a Bleed element and a turbine, use linkBleedBT() as follows:

```c
linkBleedBT("bld_elem_name", "turb_name", fracW, hscale, Pscale, Pfract, diaPump, "port_name");
```

The five arguments before "port_name" are discussed in sections 4.7.3 and 4.7.2. For example:

```c
linkBleedBT("DiffBld", "TrbH", 0.01, 1.00, 0.95, 0.70, 1.05, "TrbHV2bld");
```

To create a bleed between two Bleed elements, use linkBleedBB() as follows:

```c
linkBleedBB("bld_out_name", "bld_in_name", fracW, hscale, Pscale, "port_name");
```

The three arguments before "port_name" are discussed in Section 4.7.3. For example, the bleed established between Bleed elements DiffBld and TrbHV1 in the examples above could be established in the following single step (after the Bleed elements themselves had been instantiated):

```c
linkBleedBB("DiffBld", "TrbHV1", 0.08, 1.00, 0.95, "V1bld");
```

Each of these functions makes triple use of the name provided for "port_name":

- The bleed output port is given this name.
- The bleed input port is given this name.
• The station that results from linking these ports is given this name.

There is no collision between the names because of the NPSS hierarchy of objects. The station exists in global scope, and the ports exist as children of their respective parent elements.

The station name can be used to access variables in either fluid port. Therefore changes in bleed properties after the links have been established can be done by either using the bleed port name or the station name. In the following example, refer to the `linkBleedCT()` example above. The following statements are equivalent:

```plaintext
CmpH.TrbHlpBld.fracBldW = 0.;
TrbHlpBld.fracBldW       = 0.;
TrbH.TrbHlpBld.Pfrac    = 0.;
TrbHlpBld.Pfrac         = 0.;
```

For more information on the functions in `bleed_macros.fnc` see Section 12.4.1.

Another available User Function is the `preBleed()` function. The source terms for Bleed output ports (prior to being overwritten by linking) are available via the user hook "preBleed" which exists for all interStageBleeds. An example of a preBleed function definition for compressor “Comp” is shown below. This example reads the initial values of Pt and Tt.

```plaintext
Comp {
  PRdes = 2.507;
  effDes = 0.8850;
  NcDes  = 10000.;
  Sh_{O}.Nmech = NcDes*sqrtTheta; // initial pass not at 0 speed.
  bleed1 {
    Wbld   = 0.1;
    switchFlow = "ABSOLUTE";
    fracBldP = 0.5;
    fracBldWork = 0.5;
    int initialPass = 0;
    void preBleed() {
      if(!initialPass) {
        initialPass = 1;
        cout << "Comp.bleed1.Pt = " << Comp.bleed1.Pt << endl;
        cout << "Comp.bleed1.Tt = " << Comp.bleed1.Tt << endl;
      }
    }
  }
}
```

### 4.8 Solver Setup

As mentioned in Section 4.6, the top-level assembly of any NPSS model automatically contains a solver created by the system. This solver is named simply `solver`. NPSS solver objects contain many attributes that the user can set to affect the solver's behavior. These are discussed in detail in Chapters 6 and 16. Basic solver setup, however, is quite simple.

#### 4.8.1 Basics

The first step in solver setup is ensuring that all the required objects in the model will be executed in an acceptable order. Each assembly, including the top-level assembly, has a string array variable named `executionSequence` that holds the names of objects to be executed on each solver pass, in the order in which they are to be executed. The user can also place object names in arrays `preExecutionSequence` and `postExecutionSequence`, which by default are empty. It is common to place `DataViewers` in `postExecutionSequence` to store or display the results of each converged solution (see Section 4.9.2).

Next the operating mode of the model is set as determined by Option variable `switchDes`.

• `switchDes = "DESIGN"` causes geometry and scale factors to be calculated to achieve specified performance.
Common Tasks

• \texttt{switchDes} = "OFFDESIGN" uses specified geometry and scale factors to calculate resulting performance.

Variable \texttt{switchDes} is an attribute of many NPSS elements and subelements, allowing the user to specify some elements/subelements to run in design mode while others in the same model are in offdesign mode. For most purposes, however, it is sufficient to set all objects throughout the model in one mode or the other. Global function \texttt{setOption()} is available to accomplish this in a single statement (see Section 2.2.4.7). Most models are initially run in design mode, so the following command would be issued from global scope:

\begin{verbatim}
setOption( "switchDes", "DESIGN" );
\end{verbatim}

Some elements have additional option variables that control the specific inputs they require in either design or offdesign mode. As the user instantiates elements, he or she should consult the \textit{NPSS Reference Sheets} for those elements to determine the available options and necessary inputs. For example each \texttt{Burner} object has an option variable named \texttt{switchBurn} that determines if the object uses fuel/air ratio, fuel flow, or outlet temperature as its input value. This variable, and any others like it that may pertain to other objects in the model, should be set before the solver is set up since their values may affect which quantities are regarded as independent variables by the solver. Suppose, then, a cycle has a single \texttt{Burner} object named \texttt{Brn}. Its mode is set as follows:

\begin{verbatim}
Brn.switchBurn = "TEMPERATURE";
\end{verbatim}

Finally, a powerful global function named \texttt{autoSolverSetup()} is available that normally produces a complete and correct solver setup, even when multiple solvers are involved. It is called from global scope:

\begin{verbatim}
autoSolverSetup();
\end{verbatim}

For many situations, all that remains after this step is to define the design point condition and run the model (see Section 4.10).

Once a design point has been run, the resulting geometry and scale factors can be fixed by switching to offdesign mode.

\begin{verbatim}
setOption( "switchDes", "OFFDESIGN" );
\end{verbatim}

It may be desired to change the operating mode of certain components, and start with appropriate initial conditions from the design point.

\begin{verbatim}
Brn.switchBurn = "FAR";
Brn.FAR = Brn.Fl_O.FAR;
\end{verbatim}

Switching to offdesign mode necessitates a different solver setup. This is automatically handled by \texttt{autoSolverSetup()}, which should be called after the call to \texttt{setOption()} illustrated above. Function \texttt{autoSolverSetup()} clears the former solver setup before establishing the new one, so after it has been issued the model is again ready to run. More information on this function is given in Section 6.7.

4.8.2 Critiquing Solver Setup

The solver varies a number of \textit{independent variables} to satisfy an equal number of \textit{dependent conditions}. Each independent variable, and certain information on how it is handled, is specified by an object of type \texttt{Independent}. Each dependent condition is similarly specified by an object of type \texttt{Dependent}. Information on creating \texttt{Independent} and \texttt{Dependent} objects, and adding them to a solver, is given in Section 4.8.3 below. This section notes some ways to identify the independent variables and dependent conditions active in a solver, such as those added automatically by \texttt{autoSolverSetup()}.

The names of the independent objects active in a solver can be obtained by calling the \texttt{list()} member function of the solver:

\begin{verbatim}
solver.list( "Independent", FALSE );
\end{verbatim}
(See Section 12.1.1 for more information on the \texttt{list()} function.) The second argument to the function controls whether recursion through child objects is performed; it can always be \texttt{FALSE} for solver objects. Note also that a function named \texttt{list()} exists in the top-level assembly. The command \texttt{list("Independent", TRUE)}, called from global scope, will list all \texttt{Independent} objects that exist throughout the model. As will be discussed in Section 4.8.3, however, an \texttt{Independent} object must both exist and be \texttt{added} to a solver in order for the solver to use it. The command given above, where the solver's \texttt{list()} function is called rather than the global \texttt{list()} function, is the correct way to determine the independent variables controlled by the solver.

The names returned by \texttt{solver.list()} are the names of objects of type \texttt{Independent}. These are \texttt{not} the names of the variables controlled as independents by the solver. The name of the variable actually controlled is stored in attribute \texttt{varName} of the \texttt{Independent} object (more on this below).

Similarly, the names of the dependent objects active in a solver can be obtained as follows:

\begin{verbatim}
   solver.list( "Dependent", FALSE );
\end{verbatim}

Note, as with finding active \texttt{Independent} objects, that the solver's \texttt{list()} function must be called, not the global \texttt{list()} function. Each \texttt{Dependent} object has two attributes, \texttt{eq_lhs} and \texttt{eq_rhs}, that hold strings indicating quantities that are to be equal when the solver converges. They may be variable names, constants, or expressions.

\texttt{Independent} and \texttt{Dependent} objects can reside within elements and subelements. The variables referenced in \texttt{varName}, \texttt{eq_lhs}, and \texttt{eq_rhs} are found by beginning in the scope of the \texttt{Independent} or \texttt{Dependent} object containing them. For example, suppose \texttt{CmpH.S_map.dep_errWc} is a \texttt{Dependent} object. \texttt{CmpH.S_map.dep_errWc.eq_lhs} equals "Wc". NPSS begins looking for a variable named \texttt{Wc} in \texttt{CmpH.S_map}. In this case, it does not find it there, so in accordance with the scoping rules for variables (see Section 2.2.4.3) it looks next in the parent object. In this case it finds it there. The variable on the "left-hand side" of the dependent condition equation, therefore, is \texttt{CmpH.Wc}. This situation makes it somewhat more difficult than might be expected for the user to identify the exact variables active in a solver.

The following function illustrates how to critique a solver setup. Function \texttt{printSolverSetup()} uses \texttt{list()} to find all the active \texttt{Independent} and \texttt{Dependent} objects in a specified solver. The user must know the names of these objects to alter their attributes for special purposes (see Sections 6.5 and 6.6). The path name of each object also indicates the scope of the variable names contained in attributes \texttt{varName}, \texttt{eq_lhs}, and \texttt{eq_rhs}. The function prints these attributes in a convenient format. Independent variables and dependent conditions are numbered, \texttt{not} because the two are paired, but to allow the user to readily grasp the size of the system being solved, and confirm that there are an equal number of independents and dependents.

\begin{verbatim}
void printSolverSetup ( string Sname ) {
   string indeps[] = Sname->list("Independent", FALSE );
   string deps[] = Sname->list("Dependent", FALSE );
   cout << Sname << " Independent Variables:" << endl;
   int i;
   string var;
   for (i=0; i<indeps.entries(); i++) {
      var = indeps[i]->varName;
      cout << "  " << i+1 << " " << indeps[i] << ": 
" << var << endl;
   }
   cout << Sname << " Dependent Conditions:" << endl;
   string lhs, rhs;
   for (i=0; i<deps.entries(); i++) {
      lhs = deps[i]->eq_lhs;
      rhs = deps[i]->eq_rhs;
      cout << "  " << i+1 << " " << deps[i] << ": " << rhs << endl;
   }
} // end printSolverSetup()
\end{verbatim}
For a single solver in the top-level assembly, `printSolverSetup( "solver" )` will print the solver's active independent variables and dependent conditions to the screen for user inspection. The same function can be used to critique the solvers in sub-assemblies by giving the name of the solver (e.g., `printSolverSetup("Core.solver")` – see Section 4.8.5). By doing this, or manually executing the same basic steps, the user can confirm that a model's solvers are converging all the important conditions of the model.

### 4.8.3 Adding to the Automatic Solver Setup

The user will not always want to specify burner fuel flow, fuel/air ratio, or exit temperature. In many cases he or she will want the solver to vary one of these quantities until some other condition (such as a desired net thrust) is achieved. To do this, additional `Independent` and `Dependent` objects must be instantiated and added to a solver.

To give the solver control over a burner fuel flow, for example, an `Independent` object must be created. This is done by using the following items in order:

1. **Keyword** `Independent`
2. **The object name**
3. An optional block of statements (enclosed in curly braces) executed when the object is instantiated. Normally, at a minimum, attribute `varName` will be set here. The user **must** assign a value to `varName` before the object can be used by a solver.

As an example, assume a cycle with a single `Burner` object named `Brn`.

```plaintext
Brn.switchBurn = "FAR";
Independent fuelAir {
    varName = "Brn.FAR";
    indepRef = "0.02";
}
```

Attribute `indepRef` a string variable containing a numerical value, variable name (such as a design parameter), or expression that yields a single real number, provides a *reference value* indicating the approximate value (order of magnitude) of the independent variable. It is used by the solver when calculating partial derivatives and when damping iterations (see sections 6.2 and 6.5). Defining `indepRef` is not required. If `indepRef` is not specified, the reference value becomes the value of the variable specified by `varName`. This often proves satisfactory. More details on this and other `Independent` object attributes are given in Section 6.5.

The desired operating condition must be specified in a `Dependent` object, which is created similarly:

```plaintext
Dependent RunCondition {
    eq_lhs = "CmpH.S_map.NcMap";
    eq_rhs = "1.0";
}
```

The user **must** assign values to `eq_lhs` and `eq_rhs` before the object can be used by a solver. In this example, the model is run to 100% map speed on the high-pressure compressor. More details regarding `Dependent` objects are given in Section 6.6.

The two objects just created do not become an active part of a solver until they are *added* to a solver. This is done with two solver member functions: one for independents and one for dependents. In the following example, the objects are added to the top-level assembly solver named `solver`:

```plaintext
solver.addIndependent("fuelAir");
solver.addDependent("RunCondition");
```

Adding an `Independent` or `Dependent` object to the solver does not move the object in the NPSS hierarchy. In the examples above, objects `fuelAir` and `RunCondition` were created in the top-level assembly. After being added to `solver`, they still reside in the top-level assembly. They **cannot** be addressed as `solver.fuelAir` or `solver.RunCondition`. 
Normally the steps above would be taken after `autoSolverSetup()` had been run. However, both Independent and Dependent objects have an attribute named `autoSetup` that, when set to TRUE (or integer 1), instructs `autoSolverSetup()` to add the objects when it executes. A similar task as that above could be accomplished with the following statements:

```plaintext
Independent fuelAir {
    varName   = "Core.Brn.FAR";
    indepRef  = "0.02";
    autoSetup = TRUE;
}
Dependent RunCondition {
    eq_lhs    = "CmpFSec.S_map.NcMap";
    eq_rhs    = "1.0";
    autoSetup = TRUE;
}
autoSolverSetup();
```

In this example a specified fan map speed is the desired operating condition. Also, the burner object is in an assembly named `Core`. Specific issues involving solvers and assemblies are discussed in Section 4.8.5.

Once a general Dependent object such as `RunCondition` above has been created and added to a solver, the operating condition of the model can easily be changed. For example, the following lines set the model up to converge to a net thrust (calculated in an element named `PERF`). The desired thrust value is stored by the user in a global variable named `thrust`.

```plaintext
RunCondition.eq_lhs = "PERF.Fn";
RunCondition.eq_rhs = "thrust";
solver.forceNewJacobian = TRUE;
```

As discussed in Section 6.2, the NPSS solver constructs a matrix called a Jacobian matrix that records the influence of each independent variable on each dependent condition. By default, the solver uses the Jacobian matrix from the previous point to begin iterations on the current point. When Independent and Dependent objects are added or removed from the solver setup, the solver sets its `forceNewJacobian` attribute to TRUE so that a new Jacobian is formed with the new independents and dependents before beginning iterations. However, if the user simply changes the identity of an Independent or Dependent object already added to the solver as in the example above, the solver does not know that its current Jacobian is no longer valid. It is important to force the generation of a new Jacobian by manually setting the `forceNewJacobian` attribute to TRUE as in the example. This is strictly necessary only when the variable being controlled by an independent (specified by `varName`) or the variable being matched by a dependent (normally specified by `eq_lhs`) is changed. If only the value of the right-hand-side of the dependent condition is changed, it is not normally necessary to force formation of a new Jacobian. Sometimes, however, forcing the initial formation of a new Jacobian can help converge a difficult point.

It is possible to remove objects from a solver using commands `removeIndependent()` and `removeDependent()`.

```plaintext
solver.removeIndependent("fuelAir");
solver.removeDependent("RunCondition");
```

These commands do not destroy the Independent and Dependent objects; they only make them "inactive" by removing them from the solver setup. They can be added to a solver again at any time.

It is also possible to remove all objects from a solver with a single function: the `clear()` member function of the solver object. For example:

```plaintext
solver.clear();
```

This removes all Independent and Dependent objects from the Solver object named `solver`. It does not reset any of the Solver object's attributes (see Section 6.7). It is usually unnecessary to call `clear()` explicitly because `autoSolverSetup()` calls it automatically before constructing a new solver setup.
4.8.4 Saving and Restoring A Solver Setup

The solver contains the parameters independentNames, dependentNames and DSVNames that will return or accept an array of Independent, Dependent and DSV object path names. If these parameters appear on the right hand side of an assignment statement or in output stream, the solver returns a string array containing the full path name of each of the corresponding terms. For example, if the solver contained the Independents CmpH.S_map.ind_RlineMap and fuelAir, then the following text

    string savedIndepNames[] = solver.independentNames;

would create an 1D string array savedIndepNames whose contents are ("CmpH.S_map.ind_RlineMap", "fuelAir").

When these parameters appear on the left side of an assignment statement, the solver first removes all items of the given type already present in the solver and then adds each of the named objects to the solver setup. The following restores the Independents in the solver to the list saved above. Any Independents that were present in the solver prior to this are removed.

    solver.independentNames = savedIndepNames;

The same applies to DSVNames. The dependentNames array has one difference. The Dependent names array returned by the solver contains only unconstrained Dependents and target Dependents. It does not contain any constraint Dependents. However, if any of the Dependents that are listed have constraint Dependents associated with them, those constraints are added to the solver along with their associated target Dependent when the dependentNames array is assigned to the Solver. An additional attribute, constraintNames, is available in the solver that will return a string array containing the path names of all the constraint Dependents present in the solver. This attribute is intended for output use only. Assigning an array of constraint Dependent names to constraintNames will not affect the constraints present in the solver. The only effect will be to check whether the number of names in the assigned array is equal to the number of constraints already present in the solver. An error message is issued if they are not.

The solver also contains the attributes independentValues, DSVvalues, and jacobian. The first two return and accept 1-D arrays of real numbers containing the values of the Indendents and DSVs. The jacobian attribute returns and accepts a 2-D real array containing the contents of the full rectangular Jacobian matrix (containing rows for Dependents and Constraints).

The three value arrays along with the three name arrays represents the full solver setup and state. When the names and value arrays have been set in the solver and any user input in the rest of the model has been restored to the same values as when the solver values were saved, the model is completely restored to the same converged state it was when the values were saved. This represents a very efficient way to restore a model to a previously converged point. If all inputs are reset to the same values, then a point run immediately after restoring the solver state should converge on the first pass regardless of the operating conditions prior to restoring the solver.

4.8.5 Solvers and Assemblies

By default, no solver is instantiated for assemblies created under the top-level assembly. When assemblies with no dedicated solvers are contained within the top-level assembly, autoSolverSetup() will find the necessary independent variables and dependent conditions within the sub-assemblies, and add them correctly to the top-level assembly’s solver. It is often unnecessary, then, to instantiate a solver object dedicated to a sub-assembly. When assemblies with solvers are contained within the top-level assembly, autoSolverSetup() adds the necessary independent variables and dependent conditions to each solver as appropriate. Function autoSolverSetup(), called once from global scope, normally produces complete and correct solver setups even when multiple solvers are involved.

When an assembly's solver executes a sub-assembly, the sub-assembly's solver (if present) will attempt to iterate to convergence before returning control to the higher level assembly's solver. Although useful for some problems, this situation can result in less stable models that take longer to converge. In general, the user should not instantiate solvers in sub-assemblies unless there is good reason to do so.
To instantiate a solver within an assembly, use the following items in order:

1. **Keyword** `Solver`
2. The solver name
3. An optional block of statements (enclosed in curly braces) executed when the solver is instantiated. See Section 6.7 for information on `Solver` object attributes.

Suppose a model contains an assembly named `Core` (see Section 4.6). A solver for this assembly would be instantiated as follows:

```c++
Core {
    Solver solver;
}
```

There is no name collision between this solver and the solver by the same name in the top-level assembly because of the hierarchy of objects within NPSS. From global scope, the top-level assembly's solver is referenced by simply `solver`; the `Core` assembly's solver is referenced by `Core.solver`. From within the `Core` assembly's scope, the `Core` assembly's solver is referenced by simply `solver`.

The model with two solvers, the top-level solver and the solver in `Core`, can be set up and run as follows:

```c++
autoSolverSetup();
CASE++;
runch;`

**4.9 Input and Output**

### 4.9.1 File Streams

Information flows into and out of NPSS in what are called *streams*. Specifying an *input stream* means specifying the *source* of information NPSS will process, such as the keyboard or a file on disk. Specifying an *output stream* means specifying the *destination* of information NPSS will produce, such as the monitor, a file on disk, or a printer. A typical NPSS model might have several streams for such things as detailed cycle sheets, data for plotting, and error or warning messages.

#### 4.9.1.1 Input Streams

Input streams are normally used to read information that is not in NPSS input syntax. To process an NPSS input file, simply use the preprocessor command `#include` or command `parseFile()` (see Section 2.2.3.1).

By default, input streams are opened in ‘text’ mode. The `binary` attribute may be set TRUE to open the file in binary mode, where end-of-line translations are disabled and data values are read as binary entities rather than text. The discussion below refers to the default mode (`binary == FALSE`).

A specific input stream is established by instantiating an object of type `InFileStream`. This is done in a manner similar to the instantiation of other NPSS objects. `InFileStream` objects contain an attribute named `filename` that is normally set in the instantiation block. The general syntax is:

```c++
InFileStream stream_name {
    filename = "source";
}
```

*Source* can either be a file name or the special source named `cin` that accepts information from standard input (by default, the keyboard). For example:

```c++
InFileStream test_dat a {
    filename = "engine001.dat";
}
```

```c++
InFileStream keyboard {
```
filename = "cin";
}

An alternate syntax for declaring an input stream uses the InFileStream object's open() member function:

```cpp
InFileStream stream_name;
stream_name.open("source");
```

For example:

```cpp
InFileStream test_data2 {
    test_data2.open("engine002.dat");
}
```

For the following examples, assume file engine001.dat, connected to input file stream test_data1 in the first example above, has the following content:

```
PBARO     1001  14.652
TCELL1    2001  75.332
TCELL2    2002  77.021
PT1       1021  15.077
```

Once an input stream has been defined, there are multiple ways to obtain information through it. The first is used to read information directly into NPSS variables of the appropriate type. This is done using the "get from" operator ">>". Information can be put into any of the basic NPSS variable types (see Section 2.2.4). For example, to read a string, an integer, and a real number from the same line in file engine001.dat:

```cpp
string instName;
int instNum;
real meas;
test_data1 >> instName >> instNum >> meas;
```

In the input source, the string, integer, and real value must be separated by one or more spaces, and the string cannot contain spaces. Any spaces before the first character of the string will be ignored. In the example, variable instName will equal "PBAR0", instNum will equal 1001, and meas will equal 14.652.

If an end-of-line is encountered before all the variables have been read, NPSS will attempt to read from the next line. If more information remains on a line, the next use of >> with that stream will resume reading from the middle of the line. For example, consider the following input statements:

```cpp
test_data1 >> instName;
test_data1 >> instNum >> meas;
```

Variable instName will equal "TCELL1", instNum will equal 2001 and, meas, read from the following line, will equal 75.332. The line beginning with "TCELL2" will not be read.

The second way to obtain information from an input stream is to use function getline(), which is a member function of InFileStream objects. Function getline() reads until an end-of-line is encountered. For example:

```cpp
string line;
line = test_data1.getline();
```

With the sample content of file engine001.dat given above, this will give variable line the content "TCELL2 2002  77.021". Any leading spaces on the line are ignored. If a previous input statement left information remaining on a line, function getline() will pick up where the previous statement left off. For example:

```cpp
test_data1 >> instName;
line = test_data1.getline();
```

Variable instName will equal "PT1", and line will equal "1021  15.077".
The third way to obtain information from an input stream is to use member function `getc()`, which reads a single character.

```cpp
string char;
char = test_data1.getc();
```

Function `getc()` reads spaces and end-of-line characters just like non-blank characters. Like `getline()`, `getc()` reads its next character exactly where any previous input statement left off, even if this is in the middle of a line. The following example reads the remainder of a line into a string variable, one character at a time, stopping when an end-of-line character is detected.

```cpp
Line = "";
while ( ( char = test_data1.getc() ) != endl ) {
    line += char;
}
```

Finally, input streams include the functions `getInt()`, `getReal()`, and `getString()`. These are analogous to the “get from” operator, but may be invoked directly, or more typically, via the indirect reference operator.

If a read has been successful, integer variable `good`, an attribute of `InFileStream` objects, will be set to 1 (or `TRUE`). If an end-of-file has been encountered, integer attribute `eof` will be set to 1 (or `TRUE`). These variables allow the user to test for input problems, or for the end of an input file.

If input information is to be read from standard input, it is not necessary to create an input stream with `filename = cin` (as in the second example of this section). Rather, `cin` can be used as a predefined input stream connected to standard input:

```cpp
cin >> line;
line = cin.getline();
char = cin.getc();
```

To close an input stream, member function `close()` may be used:

```cpp
test_data1.close();
```

All streams are closed automatically when NPSS terminates if they have not been closed manually as above.

Input streams have additional attributes that control aspects of their behavior. See the Stream Reference Sheet for full details (Chapter 15).

### 4.9.1.2 Output Streams

Output streams allow data to be sent to a monitor, a file, or a printer.

By default, output streams are opened in "text" mode. The `binary` attribute may be set `TRUE` to open the file in binary mode, where end-of-line translations are disabled and data values are written as binary entities rather than text. The discussion below refers to the default mode (`binary == FALSE`).

A specific output stream is established by instantiating an object of type `OutFileStream`. This is done in a manner similar to the instantiation of other NPSS objects. Like `InFileStream` objects, `OutFileStream` objects contain an attribute named `filename` that is normally set in the instantiation block. The general syntax is:

```cpp
OutFileStream stream_name {
    filename = "destination";
}
```

`Destination` can be a file name, a printer port, or one of the special destinations `cout` or `cerr`. `Destination ``cout` is connected to standard output; `cerr` is connected to standard error. By default, both are displayed on the monitor screen. Some examples follow:

```cpp
OutFileStream diag {
    filename = "model.diagnostics";
}
```
An alternative syntax for declaring an output stream uses the stream's open member function:

```cpp
OutFileStream screen {
    filename = "cout";
}

OutFileStream errors {
    filename = "cerr";
}
```

For example:

```cpp
OutFileStream plotfile;
plotfile.open("model.plt");
```

Once an output stream has been defined, there are three ways to send information to it. The first is to use the "put to" operator `<<`. The value of any NPSS variable type can be "put to" an output stream this way, as well as quoted text supplied by the user. A line return is written using the predefined string `endl` (see Table 7), or the quoted characters "\n". The examples below use `OutFileStream diag` defined in the first example above, and are written to a file named "model.diagnostics".

```cpp
string str1 = "The variables are: ";
int dec = 10;
real a = 3.1415;
diag << "Variables output:\n";
diag << str1 << "dec = " << dec << "", a = " " << a << endl;
diag << str1 << "dec = " << dec;
diag << ", a = " << a << "\n";
```

When information to an output stream is not terminated by an end-of-line character, subsequent "puts" to the stream will begin in the middle of the line where the previous statement left off. Thus, the last two lines in the example above produce a single line of output identical to that produced by the line before them, namely:

```
The variables are:  dec = 10, a = 3.1415
```

`OutFileStream` objects use the "put to" operator to produce special formatting for entire arrays. Arrays of numerical values are written across a line, enclosed in curly braces. For example,

```cpp
real realA[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
diag << "realA = " << realA << endl;
```

produces

```
realA = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 }
```

String arrays are written with one entry per line, with a final end-of-line character automatically appended. For example:

```cpp
string strA[] = { "A", "B" };
diag << "strA = " << strA;
```

produces:

```
strA = {"A",
"B" }
```

The second way to produce output using an output stream is to use the stream object's `print()` member function. It is used in much the same way the "put to" operator is.

```cpp
Diag.print("Variables output:\n");
```
diag.print(str1); diag.print("dec = "); diag.print(dec);
diag.print("", a = ""); diag.print(a);
diag.print(endl);

Member function \texttt{print()} is useful when combined with the indirect reference operator (see Section 2.2.6.5) for writing general functions that can produce output to any stream.

```c
void varOut( string o ) {
    o->print(str1); o->print("dec = "); o->print(dec);
    o->print("", a = ""); o->print(a);
    o->print(endl);
}
varOut( "diag" );
varOut( "screen" );
```

The third way to produce output using an output stream is to use a \texttt{DataViewer} object. This is the topic of the next section (4.9.2).

If an output operation has been successful, integer variable \texttt{good}, an attribute of \texttt{OutFileStream} objects, will be set to 1 (or \texttt{TRUE}).

If output information is to be written to standard output or standard error, it is not necessary to create an output stream with \texttt{filename = cout} or \texttt{filename = cerr} (as in the second and third examples of this section). Rather, one can use \texttt{cout} or \texttt{cerr} as predefined output streams connected to their default devices.

```c
cout.print("Variables output:\n");
cout << str1 << "dec = " << dec << ", a = " << a << endl;
cerr << str1 << "dec = " << dec << ", a = " << a << endl;
```

Take special note of the middle line in the example above. Using \texttt{cout} in this way is a very common way to write information about a model to the screen.

To redirect the output from one of the special output objects, assign a file (or printer port) name to its \texttt{filename} attribute just as with user-created \texttt{OutFileStream} objects. For example:

```c
cerr.filename = "model.errors";
cerr << "This is written to the file named above." << endl;
```

To close an output stream, member function \texttt{close()} may be used:

```c
diag.close();
```

All steams are closed automatically when NPSS terminates if they have not been closed manually as above.

Output streams have additional attributes that control aspects of their behavior. See the Stream Reference Sheet for full details (Chapter 15).

The \texttt{cerr} and \texttt{cout} streams may also be individually closed by setting the output stream attribute ‘\texttt{isActive}’ equal to ‘0’. For example:

```c
cout.isActive = 0;
cerr.isActive = 0;
```

The \texttt{cerr} and \texttt{cout} streams may be individually opened again by setting the output stream attribute ‘\texttt{isActive}’ equal to ‘1’. For example:

```c
cout.isActive = 1;
cerr.isActive = 1;
```

As a convenience, the \texttt{reopen()} function is available for both input and output streams. This function remembers the file name associated with an open stream, closes the stream, then re-opens the stream using the memorized file name. It essentially renews an input file, and clears/resets an output file. For example:

```c
OutFileStream myOutputStream {
```
filename = "test.out";
}
myOutputStream << "test.out is alive and well.\n"; // Initial file content.
myOutputStream.reopen(); // No need to close the file and open it again.
myOutputStream << "test.out has been closed and reopened!\n"; // New content.

After the above use of the reopen() function, the file test.out will contain:

```
> test.out has been closed and reopened!
```

Output streams have additional attributes that control aspects of their behavior. See the Stream Reference Sheet for full details (Chapter 15).

### 4.9.1.3 LocalOutFileStream

LocalOutFileStream is a special form of OutFileStream that keeps its output local to the process that created it. In a model without remote servers, LocalOutFileStream will behave identically to OutFileStream. If a LocalOutFileStream is declared in a remote server, then any output written to it will remain local to the remote server. If a regular OutFileStream is declared in a remote server, its output will be shipped back to the main calling program. The general syntax for declaring a LocalOutFileStream is:

```cpp
LocalOutFileStream stream_name {
    filename = "destination"
}
```

### 4.9.1.4 Binary and Unformatted Streams

NPSS supports reading and writing binary and Fortran unformatted stream data via the binary and unformatted stream attributes, as well as special handling within the various stream functions. A binary stream is one in which the data is not stored in a human readable format. It has the advantage of being smaller than a formatted stream when handling full precision real data (8 bytes for the binary stream vs. over 17 for the formatted stream). Binary streams also are generally quicker to process, since the formatting operations can be skipped. The drawback to binary streams is that they are "endian" specific, meaning that some machines will store the binary data in an order that doesn’t match with some other machines, rendering the file unreadable. In general, AMD/Intel-based machines are termed “little-endian” and other machines are “big-endian.” The byteSwap attribute can be used to allow reading/writing files in the format of another machine.

An unformatted stream is intended for use with a Fortran program. Fortran unformatted files are binary files with integer record-length indicators added before and after the data. NPSS directly supports reading and writing simple forms of unformatted records (i.e., scalars and individual arrays). More complicated record types can be handled by treating the stream as binary and dealing with the record length indicators in your program.

### 4.9.2 DataViewers

NPSS provides DataViewer objects to simplify designing and producing output. These objects retrieve variable values from the model and display those values in some formatted way. There are three types of DataViewers: `VarDumpViewer`, `CaseViewer`, and `PageViewer`. Two functions, `display()` and `update()`, control the way in which a DataViewer retrieves and displays information; these functions perform different tasks with different viewers.

- **A VarDumpViewer** will simply print out an alphabetical list of variables and their values. This is done whenever the viewer's `display()` function is called.
- **A CaseViewer** stores values and displays them all at once in either columns (`CaseColumnViewer`) or rows (`CaseRowViewer`). In `CaseColumnViewer` output, each column represents a single case and each row corresponds to a particular variable. A `CaseRowViewer` prints out case results so that each row represents a single case and each column corresponds to a particular variable. The `update()` function instructs a
CaseViewer to collect its data from the model, and store it internally. The `display()` function instructs it to output its internally stored data. Its internal storage is then cleared by default, but this behavior can be changed by setting attribute `doReset` to `FALSE` (see Section 14.2).

- A PageViewer, like a VarDumpViewer, displays all the variables requested at once, showing the state of the model at that instant. A single "page," as formatted by the user, is displayed whenever the viewer's `display()` function is called.

The output stream to which a DataViewer writes its data is specified by setting its `outStreamHandle` attribute equal to the stream name. For example:

```cpp
OutFileStream dump1 {
    filename = "demo.dump1";
}
VarDumpViewer Dump1 {
    outStreamHandle = "dump1";
}
```

The `outStreamHandle` variable for each DataViewer defaults to `cout`.

A DataViewer can be updated and displayed manually by inserting calls to the viewer's `update()` and/or `display()` functions where appropriate. For example,

```cpp
Dump1.display();
```

It can also be updated or displayed automatically by adding it to the `postExecutionSequence` variable of an assembly’s Executive (such as the top-level assembly’s Executive). For example:

```cpp
solver.postExecutionSequence.append( "Dump1" );
```

Each time the solver terminates, the variables specified to be printed by VarDumpViewer `Dump1` will be sent to its output stream, just as if `Dump1.display()` were called after each point. Adding a PageViewer to `postExecutionSequence` produces similar behavior. If a CaseViewer is added to `postExecutionSequence`, it is automatically updated after every point (each point's data is stored), but no data is displayed. The user must manually call the viewer's `display()` function to produce output.

A DataViewer can be made inactive by setting its `isActive` attribute as follows:

```cpp
Dump1.isActive = FALSE;
```

When a DataViewer is created, this attribute defaults to `TRUE`. Setting it to `FALSE` as above causes the DataViewer to ignore calls to its `update()` and `display()` functions. If a DataViewer has been added to an Executive's `postExecutionSequence`, using its `isActive` attribute is a convenient way to prevent output for an intermediate series of points, then resume output when it is desired.

All DataViewer objects have other common attributes that are discussed in Section 14.1. These allow the specification of headers and footers for each page, the length and width of pages, and other items. The remainder of Chapter 14 gives complete details on each type of DataViewer. The sections that follow give an introduction to each type sufficient for the user to begin using them to produce model output.

**Note:** When the AutoDoc is run to generate html files, it uses an AutoDocViewer (see Section 14.4.7). For more information on AutoDoc, see Section 13.

### 4.9.2.1 VarDumpViewers

By design, VarDumpViewers are by far the simplest DataViewers provided by NPSS (see Section 14.4.6 for full details on VarDumpViewer objects). They are convenient for quickly providing information about a model. Apart from the attributes common to all DataViewers (see sections 4.9.2 and 14.1), VarDumpViewers have only a single attribute for the user to set: `variableList`. This is a string array containing the names of variables to be displayed. These variable names can make use of two wildcard characters: `*` and `?`. When NPSS encounters a `*` in a VarDumpViewer `variableList` entry, it attempts to replace the character with zero or more other characters to match an existing NPSS variable. When NPSS encounters a `?` in a VarDumpViewer `variableList` entry, it
attempts to replace it with a single character to match an existing NPSS variable. Any number of these wildcard characters can be used in a variable name. The variables printed by the DataViewer correspond to all possible matches. For example,

```
Dump1 {
    variableList = { "CASE",
                    "*.alt",
                    "*.dTs",
                    "Amb.MN",
                    "Amb.?amb",
                    "HP?.Fl_I.?t",
                    "Brn.FAR",
                    "Brn.TtCombOut",
                    "PERF.Fn" }
}
```

This specifies the following variables to be displayed: CASE (see Table 7), all variables named alt and dTs belonging to some other object (such as a Ambient object named Amb), Amb.MN, Amb.Ps and Amb.Ts, the inlet total quantities (such as variables Pt and Tt) of objects such as CmpH and TrbH, and variables Brn.FAR, Brn.TtCombOut, and PERF.Fn. The output might appear as follows:

```
Amb.alt = 15000
Amb.dTs = 0
Amb.MN = 0.5
Amb.Ps = 8.29357
Amb.Ts = 465.178
Brn.FAR = 0.0194843
Brn.TtCombOut = 2402.94
CASE = 41
CmpH.Fl_I.ht = -11.5567
CmpH.Fl_I.kt = 81
CmpH.Fl_I.Pt = 0.0685604
CmpH.Fl_I.Rt = 488.478
CmpH.Fl_I.ut = -45.047
TrbH.Fl_I.ht = 473.566
TrbH.Fl_I.kt = 1641
TrbH.Fl_I.Pt = 161.932
TrbH.Fl_I.Rt = 0.0685497
TrbH.Fl_I.Tt = 2303.83
TrbH.Fl_I.ut = 315.639
PERF.alt = 15000
PERF.dTs = 0
PERF.Fn = 4531.06
```

A simple list of variable names and values is produced. Notice that the list is in alphabetical order, not the order given in variableList. For more control over output format, the user must use one of the other DataViewer types, which use format indicators as discussed below.

### 4.9.2.2 Format Indicators

The information output by CaseViewers and PageViewers is specified by two items:

- Lists of variable names whose values are to be output
- Corresponding format indicators

The number of variable names must always equal the number of format indicators provided for them.

The format indicators, in general, can consist of fixed text enclosed in quotes, and place holders for the values of the specified variables. Question marks ("?") are used to indicate place holders. The question marks are replaced by actual data values when the DataViewer produces its output. The number of question marks indicates the size of the
field available for the data value. Decimal points and exponent indicators can also be included to control the formatting of real values.

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Sample Format Indicator</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>??????????</td>
<td>Text is right-justified in the field indicated by the place holder.</td>
</tr>
<tr>
<td>int</td>
<td>?????</td>
<td>Values are right-justified in the field indicated by the place holder.</td>
</tr>
</tbody>
</table>
| real, decimal               | ???.????                | • The place holders right of the decimal point are always filled. For example, 10.24 output with the example format indicator would display as 10.2400.  
• Negative values require enough question marks left of the decimal point to allow for the negative sign.  
• If a real value is output using a format indicator that has no decimal point, its fractional part is dropped and it is printed as though it were an integer.  |
| real, exponential           | ???.????E??? (UNIX)     | • Either “E” or “e” can be used to indicate presence of an exponent.  
• The number of question marks right of the “E” must always equal three (one for a sign, and two for the exponent).  
• Negative values require two question marks left of the decimal point to allow for the negative sign.  |
| real, significant digit     | ?????S????             | • Values are right-justified in the field indicated by the place holder.  
• A blank space is always included in the left-most column.  
• The number of ‘?’ to the left of the ‘S’ (or ‘s’) indicate the number of significant digits that will be displayed.  
• A decimal number will be used if it fits within the total width.  
• If a decimal number does not fit, then an exponential number will be used.  
• If the exponential number does not fit then the number of significant digits will be reduced as necessary to make it fit.  
• If the total width of the field is insufficient to display the value then the format will be ignored and a default system exponential format will be used.  |

If a value will not fit in the field specified by the format indicator, a string of dashes (“-”) will be output. The only exception to this is the significant digit format. It will increase the width as needed to force the value to be displayed.

CaseViewers and objects used with PageViewers have attributes named defIntFormat, defRealFormat, defSNFormat, and defStrFormat that set default format indicators for each variable type. See sections 14.2 and 14.4.2 for details.

### 4.9.2.3 CaseViewers

After VarDumpViewers, the next DataViewers in terms of complexity are the CaseViewers of which there are two types: the **CaseColumnViewer** and the **CaseRowViewer**. As mentioned in Section 4.9.2, in CaseColumnViewer output, each column represents a single case and each row corresponds to a particular variable. A CaseRowViewer prints out case results so that each row represents a single case and each column corresponds to a particular variable. The attributes available for each are identical.

There are three basic parts to a CaseViewer's output: the **title**, the **headers**, and the **variable values**. The title is printed once each time the viewer’s `display()` function is executed. Attribute `titleBody` is a scalar string variable in which the user specifies the fixed title text and format indicators for variable values included in the title. Attribute `titleVars` is a string array in which the user specifies the variables whose values are to appear in the title. For example:

```c
OutFileStream case1 {
```
filename = "demo.case1";
}
CaseRowViewer Case1 {
    outStreamHandle = "case1";
    titleBody = "???????????????? Run on ???????? at ?????????";
    titleVars = { "VERSION", "date", "timeOfDay" }
}

Attribute titleBody can contain newline characters ("\n") to create a multi-line title. The example above will produce a title like the following:

NPSS_1.3.0 – Rev:N run on 08/17/01 at 14:03:53

A default title definition is created when a CaseViewer object is instantiated, so the user is not required to define one.

The headers identify each case run. A CaseColumnViewer prints them at the top of each column, while a CaseRowViewer prints them at the start of each row. Attributes caseHeaderBody and caseHeaderVars are used to in an analogous fashion as titleBody and titleVars. A difference is that only single-line headers should be produced (do not use "\n" in caseHeaderBody). Headers should normally be short. For example:

Case1 {
    caseHeaderBody = "Case ???";
    caseHeaderVars = { "CASE" }
}

Each header will have an appearance like "Case  42". As with titles, the user can accept a default header definition rather than defining one as described above.

The variables to be displayed and their format indicators are specified in a single string array named variableList. More than just the format can be specified. A label or alias for each variable can be specified as well as the output units. Each entry of variableList should give the variable name first. If a format indicator is provided, it should be preceded by a colon (":"). If an alias is provided, it should be preceded by an equals ("="). If a units string is provided, it should be preceded by an "at" symbol ("@"). For example:

Case1.variableList = { "Amb.alt", "Amb.MN:?.????", "Amb.dTs:???.??=dTs", "Brn.Wfuel@lmb/hr", "Brn.TtCombOut:????.??=BOT@F", "PERF.Fn:?????.?=Fn" };

Units strings must be among the valid strings recognized (see Section 11.2), and must be compatible with the units of the variable. Units conversion is performed for output only; the value and units stored in the variable remain the same.

The example above, used in a CaseRowViewer, will produce output like the following:

<table>
<thead>
<tr>
<th>Amb.alt</th>
<th>Amb.MN</th>
<th>dTs</th>
<th>Brn.Wfuel</th>
<th>BOT</th>
<th>Fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 42</td>
<td>15000.00</td>
<td>0.5000</td>
<td>0.00</td>
<td>4500.629</td>
<td>1872.64</td>
</tr>
</tbody>
</table>

Unless given an alias, the variable labels are the full variable names as specified in variableList. If no format indicator is given, the default is used. Units conversions are performed as specified, but no units strings are displayed.

Unlike titles and headers, no default is set up for variableList. The user must define this variable at a minimum for a CaseViewer to work properly.

Suppose the CaseRowViewer built by the examples above is used as shown below. The example assumes that an Independent object has been added to the solver to vary a fuel flow parameter (see Section 4.8.3), that
Dependent object RunCondition has been added to the solver (see Section 4.8.3), and that variable thrust exists in the top-level assembly. Object PERF calculates the net thrust of the engine and stores the result in PERF.Fn.

```java
    solver.postExecutionSequence.append( "Case1" );
    RunCondition.eq_lhs = "PERF.Fn";
    RunCondition.eq_rhs = "thrust";
    solver.forceNewJacobian = TRUE;

    for ( thrust=5000.; thrust<=6001.; thrust+=500. ) {
        CASE++;
        run();
    }
    Case1.display();

    for ( thrust=6500.; thrust<=7501.; thrust+=500. ) {
        CASE++;
        run();
    }
    Case1.display();
```

Output like the following will be produced:

```
NPSS_1.3.0 – Rev:N  run on 08/17/01 at 14:03:53
Amb.alt  Amb.MN   dTs  Brn.Wfuel      BOT       Fn  
Case  42  15000.000  0.5000    0.00   4500.629  1872.64   5000.0
Case  43  15000.000  0.5000    0.00   4995.383  1947.77   5499.9
Case  44  15000.000  0.5000    0.00   5509.832  2024.37   5999.8

NPSS_1.3.0 – Rev:N  run on 08/17/01 at 14:03:53
Amb.alt  Amb.MN   dTs  Brn.Wfuel      BOT       Fn  
Case  45  15000.000  0.5000    0.00   6069.513  2105.42   6500.0
Case  46  15000.000  0.5000    0.00   6645.594  2191.29   7000.1
Case  47  15000.000  0.5000    0.00   7262.976  2285.75   7500.1
```

A CaseColumnViewer with identical attribute settings would appear similarly, with columns and rows transposed.

Full details on CaseColumnViewer and CaseRowViewer objects is found in Section 14.2.

### 4.9.2.4 PageViewers

The most advanced formatting features are available with PageViewer objects. These are intended to provide extensive detail about a model’s results on a single "page" (which could occupy more than one sheets of paper when printed) per case. PageViewers are constructed with objects called text blocks (not to be confused with a collection of statements enclosed within curly braces which is called simply a "block"). The user instantiates text block objects within a PageViewer object, and positions them in relationship to one another on the page. Several kinds of text block objects are available for use with PageViewers. These are detailed in Chapter 14, but introduced here in the sections that follow.

NPSS is distributed with a general PageViewer that is adequate for many purposes. Its use is illustrated in Chapter 5. See especially sections 5.1.2, 5.3.1, and 5.4.11.

Apart from the attributes common to all DataViewers (see sections 4.9.2 and 14.1), the only attribute belonging directly to a PageViewer object is anchor. This is a string variable containing the name of the text block that is to appear in the upper left corner of the page. It defaults to the first text block object instantiated within the PageViewer object. All other text blocks are positioned beginning with the anchor text block.

All text block objects have two attributes: right and bottom. Each is a string variable containing the name of another text block object. The object named in attribute right will be placed to the right of the text block. The object named in attribute bottom will be placed below the text block. One or both of these strings can be left empty.
in a given text block, but all text blocks to be printed must be named in another text block's right or bottom attribute (except for the first text block created which is printed by default as the anchor). When objects are specified in both right and bottom attributes of a text block, the object to the right is placed first, then the object below. A given text block can only be printed once during the execution of a PageViewer.

**SimpleBlock**

This text block allows the printing of variables whose names and paths are fixed in the model. Attribute `body` is a scalar string variable in which the user specifies fixed text, and format indicators for variable values included in the body. Attribute `vars` is a string array in which the user specifies the variables whose values are to appear in the body. For example:

```plaintext
SimpleBlock Title {
    body = "???????????????????? Run on ???????? at ????????";
    vars = { "VERSION", "date", "timeOfDay" }
}
```

This text block will produce output resembling the following:

NPSS_1.3.0 – Rev:N  run on 08/17/01 at 17:52:14

SimpleBlock objects are much like the title (titleBody and titleVars) feature of CaseViewers (see Section 4.9.2.3). They can be used for purposes other than a title banner, however. Although a given SimpleBlock object can be displayed only once per page by a PageViewer, a PageViewer can contain and display several different SimpleBlock objects. Full details on SimpleBlock objects is found in Section 14.4.1.

**Dynamic Text Blocks**

NPSS offers three dynamic text blocks for use in PageViewer objects. They are dynamic because they query the model for all objects of a specified type and print specified attributes of all objects found. This allows the writing of general PageViewers, applicable to many models. The use of dynamic text blocks (which hereafter will be referred to simply as "text blocks") resembles using miniature CaseViewers (Section 4.9.2.3) within a PageViewer. Instead of the rows or columns being determined by the separate cases run, they are determined by the separate objects in the model matching a specified type.

Text blocks of type `DRowTBlock` list attributes in rows with a column for each object to which they belong. Text blocks of type `DColTBlock` list attributes in columns with a row for each object to which they belong. Text blocks of type `LinkColTBlock` are like objects of type `DColTBlock`, but can be used to print the attributes of ports. Port attributes cannot be written using `DColTBlock` or `DRowTBlock` objects.

Like CaseViewers, the output of text blocks consists of three parts: title, headers, and variable values. The title is printed once at the top of the text block each time it is executed. Attribute `titleBody` is a scalar string variable in which the user specifies the fixed title text and format indicators for variable values included in the title. Attribute `titleVars` is a string array in which the user specifies the variables whose values are to appear in the title.

The `headers` identify the variables being printed. They appear at the left of each row in `DRowTBlock` objects, and at the top of each column in `DColTBlock` and `LinkColTBlock` objects. They are automatically generated from the information provided about the variables to be printed.

The `variables` to be printed are specified using two text block attributes: `compType` and `compAttr`. Attribute `compType` is a scalar string variable containing the name of a component type. Generally this is the name of an element type (such as "Compressor", "Turbine", "Duct", "Burner", "Nozzle", etc.) or, for `LinkColTBlock` objects, the name of a port type (such as "FluidInputPort", "FluidOutputPort", or "FuelStation"). The complete list of object types that can be specified is the same as that for the `list()` command (see Section 12.1.1). Attribute `compAttr` is a string array in which the user specifies the attributes (or variables) belonging to objects of the type specified in `compType` that are to be printed. For example, if `compType = "Compressor"`, the user might wish to print `Compressor` object attributes "Nc" (corrected speed), "Wc" (corrected inlet flow), "PR" (pressure ratio),
and "eff" (adiabatic efficiency). The complete list of attributes in each component type is given in the NPSS Reference Sheets.

Like CaseViewer attribute variableList, text block attribute compAttr can be used to specify not only the names of component attributes to be printed, but also their formats, the header text associated with each, and the units of each value to be printed. Each entry of compAttr should give the variable name first. If a format indicator (see Section 4.9.2.2) is provided, it should be preceded by a colon ("="). If no format indicator is given, a default is used (see Section 14.4.2). If header text is provided, it should be preceded by an equals ("="). If no header text is provided, the attribute name is used as the header. If a units string is provided, it should be preceded by an "at" symbol ("@"). Units strings must be among the valid strings recognized (see Section 11.2), and must be compatible with the units of the attribute to be printed. Units conversion is performed for output only; the value and units stored in the attribute itself are unchanged. Full details on DRowTBlock, DColTBlock, and LinkColTBlock objects is found in Section 14.4.2.

The following examples present DRowTBlock, DColTBlock, and LinkColTBlock objects using the features discussed above.

```plaintext
DRowTBlock Ambient {
    titleBody = "Ambient --------";
    compType  = "Ambient";
    compAttr  = { "alt:?????.?",
                  "dTs",
                  "MN:?????.??",
                  "F1.O.Vflow=KTAS@knot",
                  "WAR:????E???",
                  "Ps",
                  "Ts",
                  "Fl_O.Pt=Pt1",
                  "Fl_O.Tt=Tt1"
                 };
    right      = "Compressors";
}

DColTBlock Compressors {
    titleBody = "Compressors -----------------";
    compType  = "Compressor";
    compAttr  = { "Nc:?????.?",
                  "Wc:????.??=Wc",
                  "PR:????R",
                  "eff:?.???" }
    bottom    = "Turbines";
}

DColTBlock Turbines {
    titleBody = "Turbines -----------------------";
    compType  = "Turbine";
    compAttr  = { "Np:?????.?",
                  "Wp:????.??",
                  "PR:????R",
                  "eff:?.???" }
    bottom    = "Stations";
}

LinkColTBlock Stations {
    titleBody = "Inlet Stations -----------------";
    compType  = "FluidInputPort";
    compAttr  = { "Tt",
                  "Pt",
                  "W",
                  "FAR" }
}
```
The examples above do not print any variable values in their titles. Consequently, no format indicators (Section 4.9.2.2) appear in the titleBody strings, and the titleVars array is left undefined. Notice in the first example (DRowTBlock Ambient) that DRowTBlock and DColTBlock objects can print variables in specifically named ports (e.g. Fl_0.Pt and Fl_0.Tt belonging to elements of type Ambient). To search for all ports of a certain type and print selected attributes from them, however, requires a LinkColTBlock as in the last example above.

Suppose that DRowTBlock Ambient above is the anchor text block in a PageViewer object. It will produce a block of output that is relatively few columns wide, but several lines high since each of the FlightConditons1 object attributes specified will be printed on a separate line (or row). Ambient.right = "Compressors", so text block Compressors will be printed to the right of text block Ambient. Compressors is a DColTBlock object, and will tend to produce output that is only a few lines high but many columns wide since each Compressor object attribute specified is printed left to right in a separate column. Compressors.bottom = "Turbines", so text block Turbines will be printed below text block Compressors. Turbines.bottom = "Stations", so text block Stations will be printed below text block Turbines. The output will resemble the example below.

```
<table>
<thead>
<tr>
<th>Ambient --------</th>
<th>Compressors -----------------------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt 35000.0</td>
<td>CmpFSec 3500.0</td>
</tr>
<tr>
<td>dTs 0.000</td>
<td>Core.CmpH 9125.1</td>
</tr>
<tr>
<td>MN 0.800</td>
<td>88.31 25.0014</td>
</tr>
<tr>
<td>KTAS 461.295</td>
<td>0.8250</td>
</tr>
<tr>
<td>WAR 0.0000E+00</td>
<td>MN 0.80</td>
</tr>
<tr>
<td>Ps 3.458</td>
<td>KTAS 461.295</td>
</tr>
<tr>
<td>Ts 393.854</td>
<td>WAR 0.0000E+00</td>
</tr>
<tr>
<td>Pt1 5.273</td>
<td>PS 3.458</td>
</tr>
<tr>
<td>Tt1 444.410</td>
<td>Ts 393.854</td>
</tr>
</tbody>
</table>
```

By default, the order in which objects are printed by dynamic text blocks is the order given in the executionSequence attribute of the Executive that exists in each assembly. It can be observed from the output above that an assembly below the top-level assembly exists named Core that contains objects CmpH, Brn, and TrbH (see Section 4.6).

The user might have envisioned text block Turbines appearing immediately below text block Compressors, occupying room remaining to the right of text block Ambient. Once one object has been placed to the right of another, however, objects placed below either of them will be placed below the pair considered as a group. This can result in wasted space, as in the example above. GroupBlocks, discussed in the next section, are the solution to this problem.

**GroupBlock**

A GroupBlock object is like a miniature PageViewer within the overall PageViewer object. GroupBlocks contain other text blocks, and can contain other GroupBlocks. Like PageViewer objects, GroupBlock objects have an attribute named anchor that specifies the name of the text block to be located in the upper left corner of the space
allocated to the block. By default, the anchor is the first text block instantiated within the GroupBlock object. Full details on GroupBlock objects can be found in Section 14.4.3.

An example of a GroupBlock object incorporating the Compressors and Turbines dynamic text blocks is shown below.

```
GroupBlock RotatingComponents {
    DColTBlock Compressors {
        titleBody = "Compressors ------------------";
        compType = "Compressor";
        compAttr = { "Nc:?????.?", "Wc:????.??=Wc", "PR:??.????=R", "eff:?.???" }
        bottom = "Turbines";
    }

    DColTBlock Turbines {
        titleBody = "Turbines ------------------";
        compType = "Turbine";
        compAttr = { "Np:?????.?", "Wp:????.??", "PR:??.????=R", "eff:?.???" }
    }
}
```

The incorporation of this GroupBlock in a complete PageViewer is shown in the PageViewer example below. It resolves the problem of wasted space.

**EmptyTextBlock**

An EmptyTextBlock object is used purely to help position other text blocks. It contains only white space. EmptyTextBlock objects have two integer attributes named height and width (see Section 14.4.4 for full details). Attribute height gives the number of lines or rows in the block, width gives the number of columns. For example:

```
EmptyTextBlock EndofCase {
    height = 2;
    width = 0;
}
```

**UserTextBlock**

A UserTextBlock object contains a single string array attribute named filledBlock. In it, the user specifies a list of variables whose values are set by a user-defined update() function. These become available for other text blocks in the PageViewer to print. Details are given in Section 14.4.5.

**PageViewer Example**

The following is a complete PageViewer object incorporating objects used as examples in the preceding sections.

```
OutFileStream page2 {
    filename = "demo2.page2";
}

PageViewer Page2 {
    outStreamHandle = "page2";
    anchor = "Title";
}
```
SimpleBlock Title {
    body = "???????????????????? Run on ??????? at ???????";
    vars = { "VERSION", "date", "timeOfDay" }
    bottom = "Ambient";
}

DRowTBlock Ambient {
    titleBody = "Ambient --------";
    compType = "Ambient";
    compAttr = { "alt:?????.?", "dT", "MN:?.????", "Fl_O.Vflow=RTAS@knot", "WAR:?.????E???", "Ps", "Ts", "Fl_O.Pt=Pt1", "Fl_O.Tt=Tt1" }
    right = "RotatingComponents";
    bottom = "Stations";
}

GroupBlock RotatingComponents {

    DColTBlock Compressors {
        titleBody = "Compressors -------------------";
        compType = "Compressor";
        compAttr = { "Nc:?????.?", "Wc:????.??=Wc", "PR:??.????=R", "eff:?.????" }
        bottom = "Turbines";
    }

    DColTBlock Turbines {
        titleBody = "Turbines -------------------";
        compType = "Turbine";
        compAttr = { "Np:?????.?", "Wp:????.??", "PR:?.????=R", "eff:?.????" }
    }

} // end RotatingComponents

LinkColTBlock Stations {
    titleBody = "Inlet Stations -------------------";
    compType = "FluidInputPort";
    compAttr = { "Tt", "Pr", "W", "FAR" }
    bottom = "EndofCase";
}

EmptyTextBlock EndofCase {
    height = 2;
    width = 0;
}

} // end Page2
A SimpleBlock object defining a title is the anchor. Below it appears a DRowTBlock object named Ambient that gives ambient, flight, and engine inlet conditions. To its right is a GroupBlock object named RotatingComponents, which itself contains two DColTBlock objects displaying information about all the Compressor and Turbine objects in the model. Below this entire grouping (as specified by Ambient.bottom) is a LinkColTBlock object giving information at the inlet of all components in the model. Finally, an EmptyTextBlock object produces extra space between the output for each case.

Suppose PageViewer object Page2 was executed in a model as follows:

```java
RunCondition.eq_rhs = "1.0";
CASE++; run(); Page2.display();
```

```java
RunCondition.eq_rhs = "0.9";
CASE++; run(); Page2.display();
```

In this example, it is assumed that the solver contains an Independent object that causes a burner input parameter (such as fuel flow) to be controlled by the solver, and that RunCondition is an active solver Dependent object defining some desired operating condition. (See Section 4.8 for information on solver setup.) The output produced by these lines will resemble the following.

```plaintext
NPSS_1.3.0 – Rev:N  run on 08/18/01 at 11:12:26
Ambient --------   Compressors ---------------------
    alt       35000.0  CmpFSec        3500.0   743.95   1.5000  0.8700
dTs       0.000  Core.CmpH   9125.1    88.31  25.0014  0.8250
MN         0.8000
KTAS      461.295
W eff
    Ps        3.458  Core.TrbH    181.9     8.33   5.8373  0.8950
Ts      393.854  TrbL          78.5    40.43   2.5884  0.9000
Pt1         5.273
Tt1       444.410

Inlet Stations -------------------
    FL0  Inl.Fl_I           444.410      5.273    286.919      0.000
    FL1  CmpFSec.Fl_I        444.410      5.246    286.919      0.000
    FL2  Splt.Fl_I           507.230      7.870    286.919      0.000
    FL21 Core.Fl_I           507.230      7.870    47.821      0.000
    FL3  Core.Brn.Fl_I       1375.156    12.371     48.638   1.71E-02
    FL4  Core.TrbH.Fl_I     1701.440    32.020     48.638   1.71E-02
    FL5  TrbL.Fl_I          1751.156    32.020     48.638   1.71E-02
    FL7  NozPri.Fl_I       1375.156    12.371     48.638   1.71E-02
    FL9  FePri.Fl_I       1375.156    12.371     48.638   1.71E-02
    FL12 Bypass.Fl_I        507.230      7.870    239.097      0.000
    FL17 SecNoz.Fl_I        507.230      7.870    239.097      0.000
    FL19 FeSec.Fl_I        507.230      7.870    239.097      0.000
```

```plaintext
NPSS_1.3.0 – Rev:N  run on 08/18/01 at 11:12:26
Ambient --------   Compressors ---------------------
    alt       35000.0  CmpFSec        3500.0   743.95   1.5000  0.8700
dTs       0.000  Core.CmpH   9125.1    88.31  25.0014  0.8250
MN         0.8000
KTAS      461.295
W eff
    Ps        3.458  Core.TrbH    181.9     8.33   5.8373  0.8950
Ts      393.854  TrbL          78.5    40.43   2.5884  0.9000
Pt1         5.273
Tt1       444.410

Inlet Stations -------------------
    FL0  Inl.Fl_I           444.410      5.273    286.919      0.000
    FL1  CmpFSec.Fl_I        444.410      5.246    286.919      0.000
    FL2  Splt.Fl_I           507.230      7.870    286.919      0.000
    FL21 Core.Fl_I           507.230      7.870    47.821      0.000
    FL3  Core.Brn.Fl_I       1375.156    12.371     48.638   1.71E-02
    FL4  Core.TrbH.Fl_I     1701.440    32.020     48.638   1.71E-02
    FL5  TrbL.Fl_I          1751.156    32.020     48.638   1.71E-02
    FL7  NozPri.Fl_I       1375.156    12.371     48.638   1.71E-02
    FL9  FePri.Fl_I       1375.156    12.371     48.638   1.71E-02
    FL12 Bypass.Fl_I        507.230      7.870    239.097      0.000
    FL17 SecNoz.Fl_I        507.230      7.870    239.097      0.000
    FL19 FeSec.Fl_I        507.230      7.870    239.097      0.000
```
4.10 Basic Model Running

As discussed in the preceding sections, the basic steps in constructing and running a model are:

1. Specify a thermodynamics package (see Section 4.1).
2. Instantiate the necessary Elements, Subelements, Ports, and Assemblies (see sections 0 through 4.7). Also create any necessary functions and tables (see sections 2.2.6 and 2.2.7). This step may involve using preprocessor commands (Section 2.2.3) to include pre-written components distributed with NPSS, or user written components. It may also involve use of Creation Method Facilities (Chapter 3).
3. Link the model's ports (see Section 4.3.2).
4. Insure that the model's execution sequence is satisfactory, and add any necessary items to the preExecutionSequence or postExecutionSequence attributes of each assembly's Executive.
5. Define input and output as required (see Section 4.9).
6. Set up the model's solver or solvers (see Section 4.8).
7. Define the desired cases, and run the model, modifying the solver setup as required (see Section 4.8.3).

These steps can be accomplished in different files that are brought together by the preprocessor to make a complete model and execution definition. Chapter 5 suggests an orderly way to do this, and presents a complete example.

As mentioned above in step 2, the user will often want to access pre-written components and global functions called macros that are distributed with NPSS. One may also wish to access personal or site-specific libraries of components, functions, and tables. The include path is important to instruct NPSS to look in the right places for such items. As discussed in chapter 3, the include path may be used to search for DLM and Interpreted components also. Unless the user launches NPSS with a special run script, it may be necessary to add the standard locations of DLM and Interpreted components to the include path using the –I command line option (see Section 2.1). In a standard NPSS installation, an environment variable named $NPSS_TOP will be defined that identifies the directory under which all the NPSS program files are stored. Another environment variable, $NPSS_CONFIG, will identify the kind of system being run (such as "sun", "hp", or "sgi"). The user can insure that the paths to standard NPSS components are part of the include path by starting NPSS with the following –I command line options.

npss –I $NPSS_TOP/InterpIncludes  -I $NPSS_TOP/AirBreathing/DLMComponents/$NPSS_CONFIG

A standard NPSS installation will include a script named runnpss that the user can customize to define environment variables like $NPSS_PATH, $DCLOD_PATH, and $ICLOD_PATH to include necessary directories. Starting npss using a properly configured runnpss script can save much typing over starting it with a command like that given above.

Once the first five steps above are completed, the first case run is usually a design point. This involves using command setOption() to set attribute switchDes equal to "DESIGN" throughout the model. The NPSS Reference Sheets indicate which object attributes are inputs when the object's switchDes attribute is "DESIGN". The user must set appropriate values for all these attributes. The model is run by issuing command run() from global scope. For example:

```c++
// Design operating conditions
Amb.alt = 35000.;
```
Depending on what has been added to the top-level assembly Executive’s `postExecutionSequence` attribute, the user may wish to execute specific calls the `display()` function of DataViewer objects after the `run()` command.

As noted in Table 7, global variable `CASE` is an automatically created integer variable initialized to zero. It is a convenient way to identify various cases run. It is not automatically indexed, however, and therefore must be given a desired value before each `run()` command in order to be useful. Most of the examples in this Guide index `CASE` by one before calling `run()`, as in the example above.

Normally, offdesign points are run next. The model’s design point performance can be repeated in offdesign mode with commands such as the following:

```plaintext
Brn.switchBurn = "FAR";
Brn.FAR = Brn.Fl.O.FAR;
```

```plaintext
setOption( "switchDes", "OFFDESIGN" );
autoSolverSetup();
```

```plaintext
CASE++;
run();
```

Finally, `Independent` and `Dependent` objects can be added to the solver setup to permit running the model to any desired condition. For example:

```plaintext
Brn.switchBurn = "FAR";
```
Independent fuelAir {
    varName = "Brn.FAR";
    indepRef = "0.02";
}

Dependent RunCondition {
    eq_lhs = "CmpH.S_map.NcMap";
    eq_rhs = "1.0";
}
solver.addIndependent("fuelAir");
solver.addDependent("RunCondition");
run();

The discussion above pertains to steady-state running. Transient running is discussed more fully in Chapter 6.

4.11 Moving and Deleting Objects

Users can move objects from one point to another within the model by using the move() command. In general, the syntax of the move() command is as follows:

move( "current_location", "new_location" );

Quotation marks around object names are required. For example:

move("a.b.c", "x.y.z");

In the above example, object "c", which resides in "a.b", is moved to reside in "x.y", and is renamed "z". Object "a.b.c" can be called the source object, and "z" can be called the destination object, with "x.y" being the parent of the destination object.

As mentioned in sections 4.5.3 and 4.5.4, the move() function can be used to insert an object into a socket. Care must be taken, however, since sockets and the objects inserted in them share the same name. For example:

move("sub_name", "socket_name");  // Don't do this!

The command above will replace the socket with the source object rather than inserting it in the socket. To insert an object into a socket, specify the destination object as socket_name.child.

move("sub_name", "socket_name.child");

For example, objects of element type Duct have a socket named S_dP that can accept subelement objects of type dPqP. Suppose Duct object Diffuser exists. The following code creates and inserts subelement DiffuserLoss into object Diffuser's S_dP socket.

Subelement dPqP DiffuserLoss {
    Table TB_Closs ( real MN, real RNI ) {
        RNI = 1.00 {
            MN = { 0.0, 0.5, 1.0 }
            C = { 0.05, 0.07, 0.10 }
        }
        RNI = 0.01 {
            MN = { 0.0, 0.5, 1.0 }
            C = { 0.08, 0.10, 0.13 }
        }
        RNI.interp = "linear";
        RNI.extrap = "none";
        MN.interp = "lagrange2";
        MN.extrap = "none";
    }
    switchDes = "OFFDESIGN";
}
move( "DiffuserLoss", "Diffuser.S_dP.child" );
Even though the subelement in the example above was instantiated with a name (DiffuserLoss) other than that of the socket into which it was inserted (S_dP), once it is inserted in the socket its attributes are accessed by the socket name (e.g. Diffuser.S_dP.Closs). The same rules for accessing socket object attributes and functions apply as were discussed in Section 4.5.1.

If another object occupied the Diffuser.S_dP socket before the move, it was destroyed by the move. To preserve an object inserted in a socket before replacing it with another, first move it to another location.

```
move( "Diffuser.S_dP.child", "OldLoss" );
```

After the move in the example above, socket Diffuser.S_dP is empty.

The following restrictions apply to using the move() function:

1. Both the source and destination arguments to the move() function must reside within the object from which move() is called. For example, if object "a" has a function that calls the move() function, both arguments to move() must be descendants of "a". It is recommended, therefore, that move() always be called from the top-level assembly (global scope), with complete path names given for the arguments. This practice should minimize confusion.

2. The source object (1st argument) and the parent of the destination object must both exist before move() is called.

Deleting objects anywhere in the model may be accomplished by using the delete() command. The syntax is as follows:

```
delete("a.b.c");
```

In this example, object "c", which currently resides in "a.b", will be deleted. To delete an object inserted in a socket, use the .child syntax:

```
delete( "Diffuser.S_dP.child" );
```

The following restrictions apply:

1. The object being deleted must reside within the object from which delete() is called. For example, if object "a" has a function that calls the delete() function, the argument to delete() must be a descendant of "a". It is safe to call delete() from global scope as long as the object's complete path name is specified.

2. The object being deleted cannot be an external component or contained in an external component (see Section Error! Reference source not found.).

3. The object being deleted must exist.

### 4.12 Adding or Removing Elements from a Model

Sometimes it is desirable to leave a basic model definition intact, but add or remove components for the sake of special analyses. It is easy to instantiate new components, and the delete() command discussed in Section 4.11 can be used to remove existing components. To add or remove Element objects one must alter the linking of ports. This section discusses the steps involved.

Suppose a simple model consists of the following elements, executed in the following order as indicated by the top-level assembly's Executive's execution attribute:

1. Ambient Amb
2. InletStart InletStart
3. Inlet Inl
4. Compressor CmpH
5. Duct Diffuser
6. FuelStart Fuel
7. Burner Brn
8. Turbine TrbH
9. Nozzle NozPri
10. FlowEnd FePri
11. Shaft ShH

The port links for these elements is as follows:

```cpp
    linkPorts( "InletStart.Fl_O", "Inl.Fl_I", "FL0" ); // fluid ports
    linkPorts( "Inl.Fl_O", "CmpH.Fl_I", "FL1A" ); // fluid ports
    linkPorts( "CmpH.Fl_O", "Diffuser.Fl_I", "FL2" ); // fluid ports
    linkPorts( "Diffuser.Fl_O", "Brn.Fl_I", "FL36" ); // fluid ports
    linkPorts( "Fuel.Fu_O", "Brn.Fu_I", "FU36" ); // fuel ports
    linkPorts( "Brn.Fl_O", "TrbH.Fl_I", "FL41" ); // fluid ports
    linkPorts( "TrbH.Fl_O", "NozPri.Fl_I", "FL7" ); // fluid ports
    linkPorts( "NozPri.Fl_O", "FePri.Fl_I", "FL9" ); // fluid ports

    linkPorts( "CmpH.Sh_O", "ShH.CmpH_SH", "CmpH_SH" ); // shaft ports
    linkPorts( "TrbH.Sh_O", "ShH.TrbH_SH", "TrbH_SH" ); // shaft ports
```

It is desired to insert two Element objects of type Bleed, one after the Duct object named Diffuser (to extract compressor discharge air), and one before the Turbine object named TrbH (to simulate nonchargeable bleed introduced in the turbine nozzle before the first rotor). First, the user should instantiate these objects.

```cpp
    Element Bleed DiffBld {
        BleedOutPort TrbHV1out {
            fracW = 0.08;
            Pscale = 0.95;
            hscale = 1.00;
        }
    }

    Element Bleed TrbHV1 {
        BleedInPort TrbHV1in;
    }
```

The existing station named FL36 connects element Diffuser to element Brn. This link must be broken in order to insert element DiffBld between Diffuser and Brn. Also, the existing station named FL41 connects element Brn to element TrbH. This link must also be broken in order to insert element TrbHV1 between Brn and TrbH. The user accomplishes this using the unlink() command, which takes a single string argument corresponding to a station name.

```cpp
    unlink( "FL36" );
    unlink( "FL41" );
```

Next, the new links should be established:

```cpp
    linkPorts( "Diffuser.Fl_O", "DiffBld.Fl_I", "FL35" );
    linkPorts( "DiffBld.Fl_O", "Brn.Fl_I", "FL36" );
    linkPorts( "Brn.Fl_O", "TrbHV1.Fl_I", "FL40" );
    linkPorts( "TrbHV1.Fl_O", "TrbH.Fl_I", "FL41" );
    linkPorts( "DiffBld.TrbHV1out", "TrbHV1.TrbHV1in", "TrbHV1bld" );
```

Notice that once a link has been broken, its name can be reused to establish a new link.

Although the desired elements now exist, and their ports are correctly linked, the model will not run correctly because the execution sequence is wrong. When new elements are instantiated, they are automatically added to the end of the execution sequence (i.e., appended to array solver.executionSequence). The new bleed elements, then, will be executed after all the other elements in the model. To correct this, the element names should first be removed from array solver.executionSequence (see sections 2.2.4.6 and 12.1.2 for information on function remove()).

```cpp
    solver.executionSequence.remove("DiffBld");
```
The new elements must now be inserted in the appropriate places. Element DiffBld should be executed immediately after element Diffuser. Element Diffuser is the fourth entry in array solver.executionSequence, and therefore has index 3 (array index numbers begin with 0). Element DiffBld should therefore be inserted as index 4 (see sections 2.2.4.6 and 12.1.2 for information on function insertAt()).

\[ \text{solver.executionSequence.insertAt(} 4, \text{ "DiffBld" }); \]

Element TrbHV1 should be inserted immediately before element TrbH, which has now been pushed to be the eighth element of array solver.executionSequence (index 7). Element TrbHV1 is inserted at index location 7, pushing TrbH one position further down.

\[ \text{solver.executionSequence.insertAt(} 7, \text{ "TrbHV1" }); \]

When objects are instantiated, the default value of their switchDes attribute (if present) is usually "DESIGN". If the user desires to run in offdesign mode, this attribute should be reset in the new objects.

In the example above, the solver setup does not need to be altered since Bleed objects do not add any Independent or Dependent objects to the solver (see the NPSS Reference Sheets for Bleed elements). In general, however, it is a good idea to reestablish a complete solver setup after model changes of this magnitude have been undertaken.

\[ \text{autoSolverSetup();} \]
\[ \text{CASE++;} \]
\[ \text{run();} \]

To remove objects from a model, a similar procedure is used. Errors may result from destroying objects while their links are still present, so the order of operations should be as follows:

1. Use command unlink() to break the links that will no longer be present in the final model.
2. Move or delete the desired elements (see Section 4.11).
3. Establish the necessary links for the new model configuration.
4. Correct the execution sequence (in the case of removing elements, this will usually involve simply removing the element names the current Executive’s executionSequence).
5. Correct the solver setup if necessary.

### 4.13 Prepass, Preexecute and Postexecute Functions

NPSS executes an element or subelement by calling the following three member functions in the following order:

1. preexecute() (if it exists)
2. calculate()
3. postexecute() (if it exists)

In addition, at the top of the solver’s executionSequence, a prePass function can be called.

4. prePass() (if it exists)

The calculate() function contains the preprogrammed engineering calculations of the element or subelement, designed to simulate the physics of a particular component. The prePass, preexecute(), and postexecute() functions are available for the user to define. They perform additional calculations at the top of the solver’s executionSequence loop and either immediately before or after the normal calculations.

The prePass, preexecute, and postexecute functions are generally used for three primary purposes:

- to set parameters not otherwise calculated (such as a special performance measure),
- to modify a preprogrammed calculation (such as implementing a different definition of surge margin), and
- to set parameters used in the cycle but not calculated elsewhere (such as a customer’s power extraction from a shaft, or the scalar on a compressor’s efficiency due to clearance effects).
In the following example, assume object \texttt{Brn} has already been instantiated as type \texttt{Burner}, and assume functions \texttt{loadParam()} and \texttt{NOx()} have also been defined in the top-level assembly.

```c
Brn {
    switchBurn = "FAR";
    real loadP;
    real NOx;

    void preexecute() {
        Table Beff( real Load ) {
            Load = { 0.0, 0.2, 0.4, 0.6, 0.8, 1.0 }
            eff  = { 1.0000, 1.0000, 0.9998, 0.9994, 0.9986, 0.9970 }
        }
        loadP = loadParam( Fl_I.W, Fl_I.Tt, FAR );
        effBase = Beff(loadP);
    }

    void postexecute() {
        NOx = NOxCal( FAR, Fl_O.Tt );
    }
}
```

In this example, \texttt{Burner} element \texttt{Brn} is modified to contain two new variables: \texttt{loadP} representing a loading parameter, and \texttt{NOx} representing a measure of Nitrous Oxide emissions. The burner is also placed in a mode in which fuel/air ratio (FAR) is an input value. The burner efficiency depends on the loading parameter, so a \texttt{preexecute()} function is written to first calculate the loading parameter, then use it to calculate the burner efficiency. The loading parameter is calculated with function \texttt{loadParam()} which uses variables that are defined before the burner is executed (inlet air flow, inlet air total temperature, and fuel/air ratio). The loading parameter is then used as the independent variable in a table lookup, the result of which is assigned to variable \texttt{effBase}. Variable \texttt{effBase} exists in all objects of type \texttt{Burner}, and determines the burner efficiency. Since these calculations are performed by the \texttt{preexecute()} function, the burner efficiency just set is available when the burner's \texttt{calculate()} function is called. After the burner's \texttt{calculate()} function has executed, the burner's exit temperature is known (\texttt{Brn.Fl_O.Tt}). Therefore, a \texttt{postexecute()} function is written to use this temperature and the fuel/air ratio to calculate an emissions parameter found by function \texttt{NOxCal()}. After the model in which these objects reside has run, the user can access variables \texttt{Brn.loadP} and \texttt{Brn.NOx} for output or further processing.

The \texttt{prePass}, \texttt{preexecute} and \texttt{postexecute} functions are defined just like any other function subject to the following rules:

- They must be defined within a block belonging to the object to which they pertain.
- They must not return any value (return type \texttt{void}) and must accept no arguments.
- They must be named "\texttt{prePass}" to be performed at the top of the \texttt{executionSequence}, "\texttt{preexecute}" to be performed before the normal calculations, or "\texttt{postexecute}" to be performed after.

It is important to note that simple variable assignments in a block belonging to an element or subelement are performed \textit{only once}, when the lines are first processed by NPSS. Consider the following code for example:

```c
Brn {
    Table Beff( real Load ) {
        Load = { 0.0, 0.2, 0.4, 0.6, 0.8, 1.0 }
        eff  = { 1.0000, 1.0000, 0.9998, 0.9994, 0.9986, 0.9970 }
    }
    loadP = loadParam( Fl_I.W, Fl_I.Tt, FAR );
    effBase = Beff(loadP);
}
```

This would set variable \texttt{effBase} equal to some value depending on the values \texttt{Fl_I.W}, \texttt{Fl_I.Tt}, and \texttt{FAR} happen to have \textit{before the solver begins to execute} the elements of this model. It will make this assignment only once; it will not be updated each time element \texttt{Brn} is executed. The only way to insure that \texttt{loadP} and \texttt{effBase} equal correct
values in the converged solution is to make the calculations part of a `preexecute()` function as in the first example above.

### 4.14 Error Handling and Messages

#### 4.14.1 ESO (Engineering Status Objects)

An ESO (Engineering Status Object) is an NPSS model object class that holds information about error/warning conditions. An ESO contains many attributes that were generally not available in previous modeling systems. These include a general description, an instance description and context information (calling object, file name, line number, etc.).

Historically engine performance simulations have utilized a four-digit NSI (Numerical Status Indicator) to convey the severity, category and purpose of an error/warning condition. The purpose of an NSI is to concisely display and store, in a printout or in a database, error/warning information about each point run. This is especially important when hundreds or thousands of cases are run.

One of the ESO attributes is an eight-digit ESI (Engineering Status Indicator). The ESI is just an extension of the NSI; it utilizes two digits (instead of one) to indicate category, three digits (instead of two) to indicate purpose, and contains two additional digits to indicate ownership. This was done to help avoid collisions in larger simulations. The first digit for both an NSI and ESI indicates severity. The name was changed to ESI to avoid confusion with NSI even though the purpose of both variables is equivalent.

#### 4.14.2 ESOREgistry

An ESO must be registered in NPSS before it can be utilized. At instantiation an NPSS object should register its potential ESOs. Only one registry exists. The registration system was created for the following reasons:

- To provide the user with a complete list of potential ESOs.
- To provide a mechanism for the developer to identify and avoid conflicts.
- To provide a mechanism for the developer to override the behavior associated with a potential error condition. For instance, the ESOREgistry may be used to promote/demote the severity of an input syntax error. The ESOREgistry holds information about a potential error/warning condition. A description of each ESOREgistry attribute follows.

#### 4.14.3 ESOREgistry: Identification Attributes (Unchangeable)

The following attributes are used to uniquely identify a registered ESO entry; hence by definition these attributes cannot be changed once they have been specified.

- **owner** – A two-digit number used to scope an ESO. Using this attribute will reduce the possibility of conflicts for large models with multiple contributors. The NPSS team must agree to the owner list if collisions are to be avoided. A starter list follows.

<table>
<thead>
<tr>
<th>Entity/Company</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>00</td>
</tr>
<tr>
<td>NASA</td>
<td>01</td>
</tr>
<tr>
<td>GE</td>
<td>02</td>
</tr>
<tr>
<td>PW</td>
<td>03</td>
</tr>
<tr>
<td>NPSS system</td>
<td>04</td>
</tr>
<tr>
<td>NPSS system</td>
<td>99</td>
</tr>
</tbody>
</table>

- **category** – A two-digit number that categorizes an error/warning condition. The reserved ESO categories are shown in Table 20. The NPSS team must agree to additions/changes to the list. The names given are predefined NPSS variables holding the values shown. These variables are intended for use in functions that allow the user to generate error and warning messages.
### Table 20. ESO Categories

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDEFINED</td>
<td>00</td>
</tr>
<tr>
<td>SYSTEM_OTHER</td>
<td>01</td>
</tr>
<tr>
<td>INPUT_FILE_ACCESS</td>
<td>02</td>
</tr>
<tr>
<td>OUTPUT_FILE_ACCESS</td>
<td>03</td>
</tr>
<tr>
<td>READ</td>
<td>04</td>
</tr>
<tr>
<td>WRITE</td>
<td>05</td>
</tr>
<tr>
<td>COMPUTING_OTHER</td>
<td>06</td>
</tr>
<tr>
<td>ARITHMETIC_EXCEPTION</td>
<td>10</td>
</tr>
<tr>
<td>ENGINEERING_OTHER</td>
<td>11</td>
</tr>
<tr>
<td>INPUT_INVALID_BUT_RESET</td>
<td>20</td>
</tr>
<tr>
<td>INPUT_INVALID</td>
<td>21</td>
</tr>
<tr>
<td>CALC_VALUE_INVALID</td>
<td>22</td>
</tr>
<tr>
<td>CONSTRAINT_VIOLATED</td>
<td>23</td>
</tr>
<tr>
<td>CONSTRAINT_USED</td>
<td>24</td>
</tr>
<tr>
<td>SOLVER_OTHER</td>
<td>25</td>
</tr>
<tr>
<td>SOLVER_ZERO_DET</td>
<td>30</td>
</tr>
<tr>
<td>SOLVER_CONV_FAILURE</td>
<td>31</td>
</tr>
<tr>
<td>SINGLE_PT_SOLVER_FAILURE</td>
<td>32</td>
</tr>
<tr>
<td>THERMO_INPUT_INVALID</td>
<td>39</td>
</tr>
<tr>
<td>THERMO_CONV_FAILURE</td>
<td>40</td>
</tr>
<tr>
<td>TABLE_INDEP_OUT_OF_RANGE</td>
<td>51</td>
</tr>
</tbody>
</table>

**Purpose** – A three-digit number used to identify a specific problem. Each owner group is free to define conventions for these values.

**Id** – A seven-digit number that uniquely identifies an ESORegistry entry. The three attributes owner, category and purpose are combined to form the ESO identifier. The function `ESOMakeId()` implements this relationship. All references to the ESO registry use this identifier.

### 4.14.4 ESORegistry: Changeable Attributes

The model developer can modify the following ESORegistry attributes.

**Severity** – The severity attribute serves two purposes: it controls the execution behavior when an error is encountered and it also determines the severity level that is displayed to the user via an ESI. A zero severity level will not be allowed as it introduces display problems for a resulting ESI. The severity of a registered ESO CANNOT be reduced if it is protected.

The ESO severity levels, together with predefined NPSS variables holding the numerical value corresponding to each level, are shown in the following table. It is desirable that the NPSS team reaches consensus on the warning levels, values (1-7).
Table 21. ESO Severity Levels

<table>
<thead>
<tr>
<th>Description</th>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kill Run</td>
<td>KILL_RUN</td>
<td>99</td>
</tr>
<tr>
<td>Kill Point Error</td>
<td>KILL_POINT</td>
<td>9</td>
</tr>
<tr>
<td>Error, but do not kill point</td>
<td>ERROR</td>
<td>8</td>
</tr>
<tr>
<td>Warning levels, execution proceeds</td>
<td></td>
<td>1-7</td>
</tr>
<tr>
<td>Illegal values</td>
<td></td>
<td>Other</td>
</tr>
</tbody>
</table>

provisional – The provisional attribute indicates whether the ESO is provisional or not. The default value is FALSE. If TRUE, the execution behaviors (severity=KILL_RUN or KILL_POINT) are deferred until after the last pass of a solver sequence, regardless of solver success.

active – The active attribute allows the developer to de-activate a potential ESO without changing any of the other attributes. This capability is needed in situations where the system error no longer makes any sense or where a more detailed mechanism for handling the situation is in place. A registered ESO CANNOT be inactivated if it is protected. The default value is TRUE. An example is to de-activate a solver warning that indicates a constraint was invoked.

protected – The protected attribute prevents a user from making an unauthorized reduction in severity of a registered ESO. The default value is FALSE. The developer should only set protected=TRUE for conditions where recovery from the condition is not possible. Once this attribute is set to TRUE it cannot be reset back to FALSE.

description – A generic description of the potential condition.

4.14.5 ESORegistry Functions

The following ESORegistry functions are available. Note that all functions begin with "ESOreg." They are global top-level functions; they do not operate on an object. Note that only one of these functions, ESOregCreate(), actually produces an error. All the other functions indicate problems via their return values.

void ESOregCreate (int id, int severity, string description, int provisional, int protected, int active)

Adds a new ESO to the registry. The registry will be checked to see if the specified id has already been entered. If it has, then all attributes (except severity, protected, active) are compared. If they do not match, an error is generated. The last three arguments are optional; these attributes will receive default values if not specified.

int ESOregDelete(int id)

Deletes an existing entry. The decision to delete an entry should not be taken lightly. A non-existent id returns FALSE.

int ESOregExists(int id)

Checks for the existence of the specified id. A non-existent id returns FALSE.

int ESOregGetActive(int id)

Gets the active attribute. A non-existent id returns –1.

string ESOregGetDescription(int id)

Gets the description. A non-existent id returns the null string. (Note that the null string is also a legitimate value).

int ESOregGetProtect(int id)

Gets the protected attribute. A non-existent id returns –1.

int ESOregGetProvisional(int id)

This function is used to get the provisional attribute. A non-existent id returns –1.
int ESOregGetSeverity(int id)
Gets the severity attribute. A non-existent id returns –1.

string ESOregList(int id)
Writes all the attributes for the specified id to a string. It consists of a series of attribute=value pairs. A non-existent id returns the null string.

string[] ESOregListAll()
Writes all the attributes for all of the ESOs in the registry to a string array. The first lines contain attribute names, and the following lines are the corresponding attribute values for each entry.

Int[] ESOregListIDs()
Returns a list of all ESOs ids in the registry to an int array. If no ids exist, FALSE is returned.

int ESOregSetActive(int id, int active)
Updates the active attribute. A non-existent id or invalid value returns FALSE.

int ESOregSetDescription(int id, string description)
Updates the description. A non-existent id returns FALSE.

int ESOregSetProtect(int id)
Sets the protected attribute to TRUE. Note that once an attribute is protected, it cannot be unprotected. A non-existent id returns FALSE.

int ESOregSetProvisional(int id, int immediate)
Updates the provisional attribute. A non-existent id or invalid value returns FALSE.

int ESOregSetSeverity(int id, int severity)
This function is used to update the severity attribute. A non-existent id or invalid severity returns FALSE.

4.14.6 ESO Convenience Functions
The following functions do NOT access the ESORegistery. These functions are used to preserve the integrity among the various ESO identification attributes.

int ESOgetCategory(int id)
Returns an ESO category implied by the id. An invalid id returns –1.

int ESOgetOwner(int id)
Returns an ESO owner implied by the id. An invalid id returns –1.

int ESOgetPurpose(int id)
Returns an ESO purpose implied by the id. An invalid id returns –1.

int ESMakeESI(int id, int severity)
Constructs and returns an ESI given the ESORegistry id and severity. The user may overwrite this function. The built-in function prefixes the severity to the id, yielding an eight-digit integer. Invalid inputs return –1.

int ESMakeID(int purpose, int category, int owner)
Constructs and returns an ESORegistry id given the set of identification attributes. Invalid inputs return –1.

\[ \text{Id} = 10,000 \times \text{purpose} + 100 \times \text{cat} + \text{owner} \]

4.14.7 errHandler Object Behavior
In every assembly having a solver, NPSS automatically instantiates an object named errHandler to handle error/warning conditions reported to it.
When the `errHandler` is informed of an error/warning condition, it logs the condition. The `errHandler` stores the ESOreg id, an instance description, and context information (calling object, file name, line number, etc.). The remaining information can be obtained from the ESORegistry.

Provisional messages have meaning when generated during execution of the main solver sequence. Consequently, any condition generated before or after the main solver sequence is automatically promoted to a non-provisional condition. All provisional conditions pertaining to the last execution of the main solver sequence are promoted to non-provisional status regardless of the success of the iteration—even if no iteration was performed.

```c
void ESOreport(int id, string iDescription, int verbose)
```
This VC Interface function is used to inform the `errHandler` that a error/warning condition has been encountered. The `errHandler` performs the behavior as defined in the ESORegistry for the specified `id`. An error is generated if `id` does not exist. `iDescription` is an instance/supplemental description of the condition. The information is also written to the appropriate stream. The final argument, `verbose`, determines whether or not the "general" ESORegistry error message will be displayed or not. Verbose defaults to TRUE, so you do not need to include it in most cases. When you wish to withhold the general error description and simply display the instance error message, this is the place to take that control.

```c
void message(string msg)
```
This VC Interface function is only loosely related to the `errHandler`. The output of this function is written to the stream identified in `errHandler.msgStreamName`. This information is NOT buffered in the `errHandler`.

### 4.14.8 `errHandler` Variables

The `errHandler` object variables are described as follows.

Several string variables exist that control what streams are written to. These variables can be set to the names of any valid output stream (see Section 15.2 on Output Streams).

```c
string errStreamName default="cerr" alternate="errStream"
```
Controls the stream that the error (severity>=9) ESO information is written to.

```c
string msgStreamName default="cout" alternate="msgStream"
```
Controls the stream that the `message()` function writes to.

```c
string warnStreamName default="cerr" alternate="warnStream"
```
Controls the stream that the warning (severity<9) ESO information is written to.

The following two variables control the writing of provisional ESO information to their respective streams. They do NOT influence the buffering of the ESO information in the `errHandler` object. If TRUE (1) is returned, the provisional information is displayed to the stream as it occurs. Note that the information will be displayed for every pass on which it occurs. If FALSE (0) is returned, the provisional information will be displayed only when the solver terminates. The promoted information may be out of chronological order.

```c
int showProvErrors default=TRUE
```
Controls the writing of provisional errors (severity>=9) ESO information to the "errStreamName" stream.

```c
int showProvWarnings default=TRUE
```
Controls the writing of provisional warnings (severity<9) ESO information to the "warnStreamName" stream.

The following variables provide information about errors and warnings.

```c
string[] errors
```
The array of errors contained in the msgHandler.

```c
string[] ESIs
```
Eight-digit Engineering Status Indicators.
4.14.9 errHandler Functions

The functions that operate on the errHandler object follow. Note that the other attributes can be obtained by using the ESOreg functions.

**void clear()**

Function `errHandler.clear()` clears all the saved conditions (errors/warnings) in errHandler. All conditions, including provisional conditions, are accumulated by `errHandler` until the `clear()` function is called. Function `errHandler.clear()` is not automatically called by NPSS. Therefore, it is advisable for the user to call `errHandler.clear()` after the DataViewers have updated for each case.

**int ESIexists(int ESI)**

Returns TRUE (1) if the ESI is found in the errHandler.

**string[] getContexts()**

Returns the list of context strings.

**string[] getDescriptions()**

Returns the list of the generic descriptions.

**int[] getESIs()**

Returns the list of ESIs found in the errHandler.
int[] getIds()
Returns the list of ESOreg ids found in the errHandler.

int getNum()
Returns number of conditions in errHandler.

void postKillPoint(int sequence, int index)
This is actually an assembly function. If an NCPKillPointException is generated, the user-overwriteable function postKillPoint is called.
The sequence specifies in which solver sequence the NCPKillPointException occurred, based on the following enumeration:

NOT_IN_SOLVER 0
PRE_SOLVER 1
SOLVER 2
POST_SOLVER 3

The index specifies in which element the NCPKillPointException occurred. This number can be helpful if attempting to continue the postExecutionSequence at the next element.

If users are overwriting postKillPoint, they must write the function signature to include the two integer arguments. But users need not use either argument in their version of postKillPoint.

The default behavior of postKillPoint is:
• If sequence is PRE_SOLVER or SOLVER, go directly to postExecutionSequence.
• If sequence is POST_SOLVER, skip the current element and continue the postExecutionSequence.
• Following the postExecutionSequence, run the Assembly's postexecute function.
• If a second NCPKillPointException is encountered, the point is abandoned.

### 4.14.10 ESO: Examples
Some sample code fragments follow which illustrate usage of the error handling functionality.

**Example 1: Deactivating a Solver Warning**
```c
// Solver constructor logic
// Create private class variable that holds ESO ID
int idhitConstraint = ESOMakeId( 0, CONSTRAINTUSED, 14); //owner=system=0

// register a potential ESO, make it provisional
ESOregCreate(_id_hitConstraint, WARNING, "Constraint encountered", TRUE);
// Solver constraint handling logic (report condition to errHandler object)
ESOreport(_id_hitConstraint, "Constraint="+constraint[i]+" active on dependent=
+dependent[j]);
// Logic added to a model definition file
ESOregSetActive( 0022014, FALSE); //Deactivate constraint warning
```

**Example 2: Limit Exceedance in Element Class Code**
```c
int idhumidityExceed = ESOMakeId( 2, INPUTINVALIDBUTRESET, 0);
ESOregCreate(_id_humidityExceed, WARNING, "Relative Humidity exceeds FAR25 regulation");

calculate() {
    if( humidity >= WARmax ) {
        ESOreport(_id_humidityExceed, toStr(humidity) + ">=" + toStr(WARmax);
        humidity = WARmax
    }
}
```
Example 3: Use of message() to write context information

```c
message("Processing MPS point 2 of 4\n");
```

### 4.15 Data Matching and Analysis

A common use for computational models of gas turbine engines is to match a model to measured data from a physical engine and use the resulting model to better understand both the data and the engine. Such a process can

- Reveal how engine components are performing relative to design intent.
- Provide good estimates for parameters not measured.
- Shed insight on the accuracy and reliability of measured data.
- Provide good predictions of engine performance at operating conditions other than those measured.

Three phases of the process can be identified, and are often performed sequentially:

1. Run an existing model of the physical engine to the tested operating conditions. Results from this phase can be called "predicted" results since they provide a prediction of the data measured by the test. These results can be used to guide the test during its progress.
2. Run the model to the tested operating conditions, and add independent variables and dependent conditions to the solver to adjust component performance until selected measured data are matched. Results from this phase can be called "as measured" results.
3. Lock the values of the various component performance modifiers solved for in step 2, and run the model at operating conditions other than those tested. This provides a prediction of the physical engine's performance at other conditions. It is often useful to run such a model to standard conditions, in which case the model results can be called "corrected" results. "Corrected" results from several engine tests generally provide a better basis for comparing the tests than direct use of "as measured" results.

These operations assume that the raw instrumentation values (volts, frequencies, etc.) have been converted to engineering units, and that multiple measurements representing a location in the engine have been suitably averaged into a single value that can be properly compared with the single value calculated by the model at that location. Generally these tasks are accomplished by a data acquisition and reduction system outside NPSS. If necessary, however, the powerful features of the NPSS programming language can be used to accomplish these tasks as well.

The process outlined above requires model features that are usually not included in the model used during an engine's design phase. The following steps are typically necessary:

- Add components to simulate pressure losses resulting from the instrumentation.
- Add components to represent the measurements to be matched.
- Establish a means to make the measured data available to the engine model.

The **InstrumentDuct** element is available for modeling instrumentation pressure losses. It is identical to the **Duct** element except for two features. First, it includes a switch to turn the losses on or off. This is convenient for using the same model to predict instrumented and non-instrumented performance, or for modeling different instrumentation configurations. Second, it includes an attribute specifying the number of rakes that serves as a multiplier on the base loss. The loss due to one rake can be modeled using a subelement in the $dP$ socket, and the multiplier used to obtain the total loss. This also facilitates using a single model for different instrumentation configurations. The user should refer to the **NPSS Reference Sheets** for details.

The basic model definition can be modified to include **InstrumentDuct** elements, or the procedure given in Section 4.12 can be used to insert the new elements in an existing model. With either method, modifications will be necessary to the `linkPorts()` commands and to the solver’s `executionSequence` array.

The **Instrument** element is available to represent a measurement to be matched. It has no ports, and thus is easily added to a model without modification to the `linkPorts()` commands. It is generally good practice to group **Instrument** elements together as the final elements defined and executed in a model. Each **Instrument** element
includes a number of sockets in which subelements can be used to adjust measured values for various inaccuracies, including total-to-static calibration and dynamic lag. The element calculates an error term between the adjusted measurement and a specified model variable, but does not create Independent and Dependent objects for addition to the solver. This is the user's responsibility (see Section 4.8.3). Details on the element are found in the NPSS Reference Sheets.

The ExternalDB element is available for making measured data available to the engine model. It too has no ports and can thus be added to a model without modification to the linkPorts() commands. It is unique in that it should not be added to the solver.executionSequence array (although its presence there is not detrimental – it performs no action when its run() command is called). It retrieves data from an external source when the user calls its retrieve() member function (somewhat like calling display() on DataViewers whenever output is desired). The element relies on a file that maps parameter names in the data source with parameter names in the model. Consult the NPSS Reference Sheets for details. A user-written data input subelement may be required for use with element ExternalDB, for which the user should see the Developer Guide. External data can also be read into NPSS variables by direct use of input streams (see Section 4.9.1.1).

Once a suitable model file has been constructed, an NPSS input file could be constructed to carry out the three phases of data matching and analysis listed above on a point-by-point basis.

1. Retrieve the desired data
2. Validate operating condition data and set up the solver for a prediction run.
3. Run a prediction case.
4. Validate measured cycle data and set up the solver for an "as measured" run.
5. Save the initial values of the solver independent variables.
6. Run an "as measured" case.
7. Set standard operating conditions and set up the solver for a "corrected" run.
8. Run the "corrected" case.
9. Restore the values of the solver independent variables stored before the "as measured" case was run.
10. Identify the next desired data record and repeat the process.
5 Building a Model

The following sections describe the basic building blocks of a typical engine simulation model using a two-spool fan engine. Both steady-state and transient simulations are demonstrated, although transient operation is covered in detail in Chapter 6 and in the Developer Guide.

The complete model listing is given in Section 5.4, to which the reader may wish to refer. Each part of the model is discussed in detail, together with partial listings from the model, in the sections below.

The model is read as a single continuous input stream and may be constructed in a single file. This file would contain the complete definition of model components – their inputs and how they are linked together, the tables and maps required, a definition of the output, and the inputs describing each of the conditions at which the model is to be run. The resulting file would be large. In practice, therefore, the model is sectioned into separate files which are brought into the main model file where needed by using the `#include` directive (see Section 2.2.3.1). This subdivides the model into more easily readable (and editable) portions. Some of these files may be reused for several models, others may contain portions of the model that are of less concern to the general user.

When dividing model information among several files, two points are helpful to remember.

- Variable scope is determined by a variable's location in the model's object hierarchy, not by the file in which it resides. For example, variables declared inside files that are included in the model's top-level assembly have global scope. See Section 2.2.4.3 for more information about variable scope.
- The same file may be included multiple times. For example, a pressure loss table that is a function of local Mach number may be included in different ducts within the model. Sometimes including a file more than once is undesirable, and can be prevented using preprocessor commands (see Section 2.2.3.3).

The order of items within the model, the method of separating model parts into files, and the file extensions used in this document are recommendations based on the experience of the program developers. (In the past, interpreted components used the .cmp extension and were `#included` into the model. The current recommendation is to use the .int extension which will in many cases automatically load the component, or `loadResource()` may be used to explicitly load the component. The discussion below is based on the previous recommendations).

The model file types may be identified by their file extensions:

<table>
<thead>
<tr>
<th>File description</th>
<th>Extension</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Study executive</td>
<td>*.run</td>
<td>fanjet.run</td>
</tr>
<tr>
<td>Model description</td>
<td>*.mdl</td>
<td>fanjet.mdl</td>
</tr>
<tr>
<td>Case definition</td>
<td>*.case</td>
<td>desOD.case</td>
</tr>
<tr>
<td>Functions</td>
<td>*.fnc</td>
<td>bleed_macros.fnc</td>
</tr>
<tr>
<td>Interpreted components</td>
<td>*.int</td>
<td>EngPerf.int, controls.int</td>
</tr>
<tr>
<td>Component maps</td>
<td>*.map</td>
<td>fanE3.map, hptE3.map</td>
</tr>
<tr>
<td>Tables</td>
<td>*.tbl</td>
<td>pressLoss.tbl, ramRecovery.tbl</td>
</tr>
<tr>
<td>Output viewers</td>
<td>*.view</td>
<td>ncp.view, Transient.view</td>
</tr>
</tbody>
</table>

A sample turbofan engine is described in the following sections:

| Study file | Section 5.1 |
| Model file | Section 5.2 |
| Case input file | Section 5.3 |
5.1 Study Executive File

A study executive file contains references (#include) to files used to define the model and to the case files where the model execution is defined. These files may be reusable in other simulations, and this is a convenient place to view the major parts of a simulation and to make modifications.

An example of a study executive file, fanjet.run, follows. Its parts are discussed in the sections that follow.

```
//Hi Bypass Fanjet model
//Created: 7/2000

#ifndef THERMO
#define THERMO GasTbl
#endif

//Set the thermo package
setThermoPackage("$THERMO");

//default includes
#include "ncp.view"           //default page viewer
#include "bleed_macros.fnc"   //bleed macros

//local includes
#include "Transient.view"     //transient viewer

//model file
#include "fanjet.mdl"

//case input files
#include "desOD.case"
```

5.1.1 Define Thermo Package

Before any Element or Subelement objects are defined (normally done in the model file), the thermodynamics package must be specified (see Section 4.1).

5.1.2 Include Standard NPSS Utilities and Components

```
//default includes
#include <ncp.view>           //default page viewer
#include <bleed_macros.fnc>   //bleed macros
```

Various standard global functions, or macros, are available to the user as summarized in Chapter 12, Function Summary. The default DataViewer included above (ncp.view) creates a 132-column-wide page of summary output arranged by component type for each case run. A sample output produced by the viewer is shown in Section 5.4.11. The bleed macros file (bleed_macros.fnc) contains functions that can be used to set up bleed flows, as discussed in Section 4.7.5 and illustrated below in Section 5.2.3.2.

As explained in Section 2.2.3.1, the use of angle brackets around the file names in the example above causes NPSS to search the directories in its include path (see Section 2.1) to find the files named. See Section 4.10 for suggestions on setting environment variables used by NPSS to establish the include path, and other paths used by Creation Method Facilities (Chapter 3).

5.1.3 Include Special Utilities and Components

```
//local includes
#include "Transient.view"     //transient viewer
```

The use of quotation marks around the file name in the example above causes NPSS to look only in the current directory for the files named (see Section 2.2.3.1). File must therefore reside in the same directory from which NPSS is started.
In order to obtain output from NPSS, the user must generally include one or more DataViewers (see Section 4.9.2). A standard DataViewer was included earlier (ncp.view), but the user may have created his or her own special DataViewers that can be included here. It may be desirable to use several DataViewers for a single model to obtain output specialized for different uses. It is recommended that each DataViewer be defined in a separate file with an extension of ".view".

This section of the study executive file can be used to include files defining many special objects such as tables, functions, elements, and DataViewers (such as Transient.view in the example above). Special variables can be defined here as well. Any objects defined as illustrated in the study executive file will exist in the top-level assembly and have global scope.

5.1.4 Include Model Definition

```
//model file
#include "fanjet.mdl"
```

This file defines the engine cycle, and will be discussed in detail in Section 5.2. If a single engine is being simulated, including the model definition in the top-level assembly as done above is satisfactory. If multiple engines are to be simulated, each should be included within a uniquely named assembly (e.g. Eng1 and Eng2). See Section 4.6 for more information on using assemblies.

5.1.5 Model Case Description Files

```
//case input files
#include "desOD.case"
```

One or more files are included that contain solver directives, model variable inputs, run commands, and print requests. The structure of these files is described in Section 5.3.

5.2 Model File

This model file contains the definition of the model geometry, including the components used, general inputs, and the connection between components. The model file roughly consists of four sections:

1. Header
2. Instantiating model components
3. Linking elements
4. Solver setup

5.2.1 Header

```
// File Name: fanjet.mdl
// Version: 1.0
// Date(s): 11 November 1998
// Author: John Doe

This fan jet model has been developed to use as example in User Guide document. Model uses E3 compressor and turbine maps.
```

AUTHOR = "J. Doe";
MODELNAME = "fanjet";
string Model_ID = "fanjet for documentation example";
The top of the model file will contain model descriptive information and the definition of objects that require global scope. The information contained at the top of the model file should identify the model, the author, and relevant dates (such as the date of the initial build and of later revisions).

In the example, predefined global variables AUTHOR and MODELNAME were mentioned in Section 2.2.4.10 (Table 7). Variable Model_ID is a user-created NPSS variable. Variables like this can be used to identify the model in output files.

The header comments should include notes about assumptions and modeling restrictions. A simple schematic of the model may also be included.

### 5.2.2 Instantiating Model Components

In the study executive file, the user included definitions of standard and special objects (see Sections 5.1.2 and 5.1.3). The model file is where specific instances of these objects are created. Special tables and functions that may be used more than once in the model can be instantiated first to set them apart from other objects, and insure they have global scope. Otherwise, it is best to instantiate objects in the order of normal flow from the front of the engine to the rear. A logical order of instantiation will improve the readability of the model. It also simplifies solver setup as illustrated below in Section 5.2.4.1.

The fanjet.mdl example file is shown in its entirety in Section 5.4.2, but its main components will be shown and discussed in the sections that follow. To provide an overview, the element types used are listed below.

<table>
<thead>
<tr>
<th>Element Name(s)</th>
<th>Element Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amb</td>
<td>Ambient</td>
<td>Sets ambient conditions, establishes a source of airflow into engine. (does not have an outlet port and requires the InletStart)</td>
</tr>
<tr>
<td>InletStart</td>
<td>InletStart</td>
<td>Sets the flow that enters a following Inlet.</td>
</tr>
<tr>
<td>Inlet</td>
<td>Inlet</td>
<td>Sets conditions at engine inlet, after ram recovery.</td>
</tr>
<tr>
<td>SplitFan</td>
<td>Splitter</td>
<td>Divides flow between the bypass stream and core stream.</td>
</tr>
<tr>
<td>CmpFSec, CmpL, CmpH</td>
<td>Compressor</td>
<td>Compresses flow.</td>
</tr>
<tr>
<td>FusEng</td>
<td>FuelStart</td>
<td>Establishes a source of fuel for the burner.</td>
</tr>
<tr>
<td>BrnPri</td>
<td>Burner</td>
<td>Burns fuel/air mixture.</td>
</tr>
<tr>
<td>B025, B025in, BFanOB, B041, B042, B045</td>
<td>Bleed</td>
<td>Establishes a location for bleed outflow and inflow.</td>
</tr>
<tr>
<td>TrbH, TrbL</td>
<td>Turbine</td>
<td>Extracts work from flow</td>
</tr>
<tr>
<td>Dfan, Duct43</td>
<td>Duct</td>
<td>Defines pressure loss between components.</td>
</tr>
<tr>
<td>NozSec, NozPri</td>
<td>Nozzle</td>
<td>Delivers flow to the atmosphere at engine exit.</td>
</tr>
<tr>
<td>FePri, FeSec, FeFanOB</td>
<td>FlowEnd</td>
<td>Establishes a sink for a fluid stream.</td>
</tr>
<tr>
<td>ShH, ShL</td>
<td>Shaft</td>
<td>Transfers torque between rotating components.</td>
</tr>
<tr>
<td>PERF</td>
<td>EngPerf</td>
<td>Calculates overall performance parameters.</td>
</tr>
</tbody>
</table>

Refer to the *NPSS Reference Sheets* for a complete description of available elements.

### 5.2.2.1 Ambient, InletStart and FlowStart Elements

All fluid streams must start with either an Ambient with InletStart, FlowStart, or FuelStart element.
The Ambient element includes atmospheric tables and is usually used to identify the freestream ambient flow conditions into the inlet of an engine model. A single Ambient element may provide freestream conditions for several InletStart and Inlet streams.

```
Element Ambient Amb {
   switchMode = "ALDTMN";
   alt_in = 35000.;
   dTs_in = 0.;
   MN_in = 0.8;
} // END Amb
```

**Element Ambient** initiates the ambient conditions, however this element does not have an outflow port and requires an InletStart to initiate an airflow stream, and automatically creates an outflow port, Fl_O, to send the stream to another element. The member variables alt, dTs, etc. will normally be redefined in the case input file. They are assigned values here to give the model an initial starting point in the event none is supplied before a run() command is executed, and may correspond to a design condition. When autoSolverSetup() is run in off-design the variable w is made an independent to be controlled by the solver.

A FlowStart element, as opposed to Ambient and InletStart, may be useful when modeling an isolated component or assembly.

### 5.2.2.2 InletStart

```
Element InletStart InletStart {
   AmbientName = "Amb";
   W_in = 100.;
}
```

The InletStart elements begins the freestream conditions. It references ambient conditions from the named Ambient element and defines a flow in it’s fluid out port.

### 5.2.2.3 Inlet Element

```
Element Inlet InEng {
   eRamBase = 0.995;
} // END InEng
```

Once a flow representing freestream conditions has been established, the Inlet element is generally the best way to establish conditions at the engine inlet face. In this case, the engine inlet total pressure will be 0.995 times the freestream total pressure. Details on more advanced uses of the element are found in the NPSS Reference Sheets.

### 5.2.2.4 Splitter Element

```
Element Splitter SpltFan {
   BPRdes = 5.;
} // END SpltFan
```

The Splitter element automatically creates two output flow ports: Fl_O1 (“primary”) and Fl_O2 (“secondary”). Variable BPR specifies the ratio of secondary to primary flow. It is assigned a value here either as a design condition or as an initial guess for the first case run. When autoSolverSetup() is run, BPR is made an independent to be controlled by the solver.

### 5.2.2.5 Modeling Compressors

```
Element Compressor CmpFSec {
   // Load map file.
   // This plugs the CompressorRline subelement into S_map socket,
```

5 Building a Model
// defines S_map tables TB_eff, TB_PR, TB_Wc, and sets values
// for S_map.alphaMapDes, S_map.RlineMapDes, and S_map.NcMapDes
#include "fanE3.map";
S_map.PRdes = 1.5;
S_map.effDes = 0.8589;
Sh_O.inertia = 10.; // slugs-ft**2.
} //END CmpFSec

As will be seen later, Compressor element CmpFSec models the tip portion of the fan and receives the secondary flow from the splitter. Compressors CmpL and CmpH are modeled similarly. The compressor characteristics are supplied using a subelement that returns flow, pressure ratio, efficiency, and other compressor performance parameters. The subelement is defined in file fanE3.map. Storing the compressor characteristic definition (which may be lengthy) in its own file improves the readability of the model file, and allows the characteristic to be conveniently used by other models.

5.2.2.6 Modeling a Compressor Map
The following Compressor map is an example of how the map would be constructed for a variable geometry model, where alphaMap might represent stator vane angle.

Subelement CompressorRlineMap S_map {
    alphaMapDes = 0.0; // alpha at unscaled map design point
    NcMapDes = 1.0; // Nc at unscaled map design point
    RlineMapDes = 2.0; // Rline at unscaled map design point
    RlineStall = 1.0; // Rline at stall line

    Table TB_Wc(real alphaMap, real NcMap, real RlineMap) {
        alphaMap = 0.0 {
            NcMap = 0.4 {
                RlineMap = { 1.0, 1.25, 1.5, 1.75, 2.0 }
                WcMap = { 386.838, 552.308, 670.769, 762.906, 830.598 }
            }
            NcMap = 0.5 {
                RlineMap = { 1.0, 1.25, 1.5, 1.75, 2.0 }
                WcMap = { 544.786, 659.487, 766.667, 855.043, 918.974 }
            }
        }
        alphaMap = 1.0 {
            NcMap = 0.4 {
                RlineMap = { 1.0, 1.25, 1.5, 1.75, 2.0 }
                WcMap = { 386.838, 552.308, 670.769, 762.906, 830.598 }
            }
            NcMap = 0.5 {
                RlineMap = { 1.0, 1.25, 1.5, 1.75, 2.0 }
                WcMap = { 544.786, 659.487, 766.667, 855.043, 918.974 }
            }
        }
        alphaMap.interp = "linear";
        alphaMap.extrap = "linear";
        NcMap.interp = "lagrange3";
        NcMap.extrap = "linear";
        RlineMap.interp = "lagrange3";
        RlineMap.extrap = "linear";
        extrapIsError = 0;
        printExtrap = 0;
    } //end TB_Wc
} //end CmpFSec
The following is an example of how the compressor map would be constructed for a fixed geometry model, where \( \alphaMap \) is not required to determine compressor performance. Notice the change in the placement and as a result the scope of \( \alphaMap \). The tables access functions require three parameters as arguments to each function (\( \alpha, \) \( \text{NcMap}, \) \( \text{RlineMap} \)). In a fixed geometry model \( \alpha \) will always be zero therefore this method allows the user to build more efficient maps by having \( \alpha \) as a fixed value, global to the map.
RlineMap = { 1.0, 1.25, 1.5, 1.75, 2.0 }
Pmap = { 1.091, 1.084, 1.06, 1.005, 0.9419 }
...
} //end TB_PR
} //end S_map

File fanE3.map defines two subelements. By including the file within a block associated with Compressor element Fan, they become children of that element (see Section 2.2.2.4). Subelement CompressorRlineMap must be named S_map to fit in the socket of a Compressor element (see Section 4.5.3). It returns scaled values of compressor performance parameters. The CompressorRlineMap subelement uses three tables to look up compressor map parameters: TB_Wc, TB_eff, and TB_PR, which can be declared in any order. These tables follow the ordinary rules for all tables (see Section 2.2.7). Each gives its dependent parameter as a function of three independent parameters: alphaMap (which can be used to represent variable geometry settings), NcMap (corrected speed), and RlineMap (a speedline location parameter). If the maps are independent of alphaMap (as is normally the case for fans), the table may be constructed without the use of alphaMap so that the table just contains NcMap and RlineMap (alphaMap will be ignored even though it is still an independent parameter in the table declaration).

Variable CmpFSec.S_map.RlineStall identifies the RlineMap value that corresponds to the stall line to enable stall margin calculations.

A variety of scale factors are available in the CompressorRlineMap subelement, although none appear in the example (see the NPSS Reference Sheets). When autoSolverSetup() is called, the CompressorRlineMap variable RlineMap is made an independent to be controlled by the solver. A dependent is also created to cause the Compressor element’s inlet flow (in this case CmpFSec.Wc) to equal the flow scaled from the compressor maps (CmpFSec.WcCalc). Note that even though the solver dependent is located in the S_map subelement, it is still allowed to affect variables that belong to the parent Compressor element (Wc and WcCalc).

5.2.2.7 Modeling Bleeds

This example will demonstrate the two ways to create a bleed flow discussed in Section 4.7. Each involves instantiating a Bleed element at the location where bleed is either to be extracted or reintroduced. In the first method, details about the bleed are included in the Bleed element instantiation.

```
Element Bleed BFanOB (  
  BleedOutPort BFanOB {  
    hscale = 1.0;  
    Pscale = 1.0;  
    fracW = .00;  
  }  
}  

Bleed elements automatically have inlet and outlet ports for the main flow (Fl_I and Fl_O respectively). To extract a bleed, the user must create a BleedOutPort as is the example above (see Section 4.3.1). Variables hscale and Pscale determine the enthalpy and pressure of the bleed relative to inlet values of the bleed element, and variable fracW determines the bleed quantity as a fraction of the Bleed element inlet flow (see Section 4.7.3). Port BFanOB must be linked to another element elsewhere in the model (see Section 5.2.3.2).

The second method accomplishes the same thing, but uses standard NPSS macros to accomplish it (see Section 4.7.5). The Bleed element must still be instantiated at the appropriate location, but no details about the bleed need be given.

```

Element Bleed B025in {  
}
```

Use of the macros will be illustrated in Section 5.2.3.2. File bleed_macros.fnc was included in the study executive file to enable use of the bleed macros (see Section 5.1.2).
5.2.2.8 Nozzle Element

```cpp
Element Nozzle NozSec{
    PsExhName = "Amb.Ps";
} // END NozSec
```

In this model, the bypass stream is followed to its nozzle, which is separate from the core stream nozzle (the cycle is an unmixed turbofan), before instantiating the elements corresponding to the remainder of the core stream. Modeling the core stream nozzle is similar to modeling the bypass nozzle. Variable `PsExhName` is a string variable holding the name of a variable whose value is the nozzle exit static pressure (see the NPSS Reference Sheets). In this case, it is the ambient static pressure as calculated by the `Ambient` element (named `Amb`). If the value of the named variable changes during the course of a simulation, the nozzle exit static pressure will be automatically changed to match.

By default, the `Nozzle` element calculates an ideally expanded converging-diverging nozzle. When `autoSolverSetup()` is called, a dependent is created that causes the flow per unit area at the nozzle throat to equal the value required to either choke the throat or match the specified exit static pressure (depending on nozzle pressure ratio and other options).

5.2.2.9 Burner Modeling

```cpp
Element FuelStart FusEng {
    hFuel = 182.7175;
} // END FusEng
```

```cpp
Element Burner BrnPri {
    dPqPfBase = 0.05;
    effBase = 0.98;
    switchHotLoss = "input";
    switchBurn = "FUEL";
    Wfuel = 0.35;
    tolRayleigh = 0.0001;
} // END BrnPri
```

The burner combines air and fuel, and calculates the result of burning the mixture. The burner’s `F1_I` port will receive the airflow from upstream components (established originally by the `Ambient` element), but the fuel flow has yet to be established. This is done using the `FuelStart` element as shown above. Its outlet port (`Fu_O`) will be linked later to the burner’s fuel inlet port (`Fu_I`).

The various Burner element variables are described in the NPSS Reference Sheets document. The variable `switchBurn` determines the quantity, fuel flow, fuel/air ratio, or exit temperature; it is used as an input to the burner calculation. In the example above, fuel flow is selected, and an initial value is supplied for `Wfuel`. This value is likely to be changed later in the case input file, or possibly made an independent for the solver to control.

5.2.2.10 Turbine Element

```cpp
Element Turbine TrbH {
    // Load map file.
    // This plugs the TurbinePRmap subelement into S_map socket,
    // defines S_map tables TB_eff and TB_Wp, and sets values
    // for S_map.PRmapDes and S_map.NpMapDes
    #include "hptE3.map";
```
5 Building a Model

effDes = 0.89;
PRbase = 5.0; // pressure ratio initial guess

Subelement ThermalMass S_Qhx {
    Ax = 3574.35;
    Cpmat = 1.0;
    ChxDes = .0003858;
    massMat = 100;
    wtdAvg_Fl = .5;
}

} //END TrbH

The turbine characteristics are modeled much like the compressors', and are defined in file hptE3.map. Like the Compressor element, the Turbine element contains an S_map socket that supplies scaled values. The TurbinePRmap subelement is designed to plug into the S_map socket and it contains two tables, TB_Wp and TB_eff, that give corrected flow and efficiency as functions of a corrected speed, and expansion ratio. As with the CompressorRlineMap subelement, the TurbinePRMap subelement creates both an independent and a dependent when autoSolverSetup() is called. The expansion ratio (PRbase) is put under the control of the solver as an independent, and a dependent is created such that the scaled map flow equals the inlet flow to the Turbine element at convergence (Wp = WpCalc).

The example above further shows use of a ThermalMass subelement, which can also be used with compressors (in fact it is used in the CmpH compressor object, though not discussed above). The subelement returns variable Qhx which is used to adjust the element's exit temperature to account for heat transferred to or from a thermal mass (such as the engine case). When autoSolverSetup() is called, the subelement makes material temperature Tmat an independent under the control of the solver. It also creates a dependent such that the gas path temperature, TgasPath, equals the material temperature at convergence. Details are given in the NPSS Reference Sheets.

5.2.2.11 Duct Element

Element Duct D043 {
    switchDP = "INPUT";
    void preexecute() {
        dpqP_in = 0.25 * Fl_I.MN * Fl_I.MN;
    } 
    Fl_I.MNdes = 0.4;
} //END D043

Duct elements are usually used to introduce pressure losses between components. The element's principal means of doing this is through variable dpqPbase (scale factors are also available for modification of this value). Subelement dpqP or dpqPm can be used to set dpqPbase, but in the example above, a preexecute() function is used instead (see Section 4.13). Before each execution of Duct43's main calculations, dpqPbase is set to ¼ of the inlet Mach number squared. When the duct's inlet fluid port's switchDes variable is set to "DESIGN", the area of the port will be calculated such that its Mach number equals 0.4.

5.2.2.12 FlowEnd Elements

All fluid streams must end in a FlowEnd element. The following FlowEnd elements will be linked later to terminate the core stream, bypass stream, and an overboard bleed stream.

Element FlowEnd FePri {
}

Element FlowEnd FeSec {
}

Element FlowEnd FeFanOB {
}
5.2.2.13 Shaft Elements

Element Shaft ShH {
    // Mechanical Ports. These are created as needed on the shaft.
    ShaftInputPort MeCmpH, MeTrbH;
    Nmech = 8997.43;
    inertia = .93243; //30. lbm-ft2;
} //END ShH

Element Shaft ShL {
    ShaftInputPort MeCmpFSec, MeCmpL, MeTrbL;
    Nmech = 3497.40;
    inertia = 2.73513; //88. lbm-ft2;
} //END ShL

The compressor components are mechanically connected to their respective turbine components by Shaft elements. The required number of shaft input ports are created by the user on each Shaft element, and these are later linked in such a way as to connect the proper components. The names chosen for the ports above suggest that the HP turbine will drive the HP compressor, and the LP turbine will drive both the LP compressor and the fan. The values specified for Nmech may correspond to a design condition, or provide initial values for the first case run. When autoSolverSetup() is called, each Shaft element makes its Nmech variable an independent under the control of the solver. When the solver is run in steady-state mode, autoSolverSetup() also causes dependents to be set up such that the net torque on each shaft is zero when the solver converges.

5.2.2.14 EngPerf Element

Element EngPerf PERF {
} //END PERF

Element EngPerf calculates summary engine performance parameters such as total net thrust, total airflow, and thrust specific fuel consumption. As shown in Section 5.2.4.1, the model will be set up to execute an object of type EngPerf at the end of each pass through the engine model components.

The performance variables calculated in the sample EngPerf element are as follows:

**Table 24. Variables Calculated in Sample EngPerf Element**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt</td>
<td>altitude</td>
</tr>
<tr>
<td>dTs</td>
<td>delta static temperature from standard day</td>
</tr>
<tr>
<td>Fg</td>
<td>total gross thrust</td>
</tr>
<tr>
<td>Fn</td>
<td>total net thrust</td>
</tr>
<tr>
<td>Fram</td>
<td>total ram drag</td>
</tr>
<tr>
<td>MN</td>
<td>Mach number</td>
</tr>
<tr>
<td>OPR</td>
<td>overall core engine pressure ratio</td>
</tr>
<tr>
<td>TSFC</td>
<td>thrust specific fuel consumption</td>
</tr>
<tr>
<td>Vtas</td>
<td>true airspeed</td>
</tr>
<tr>
<td>WAR</td>
<td>water to air ratio of freestream air</td>
</tr>
<tr>
<td>Wtot</td>
<td>total inlet airflow</td>
</tr>
</tbody>
</table>
If the user wishes to define additional summary variables, a good practice is to add them in a `postexecute()` function of the `EngPerf` element.

### 5.2.2.15 Control Element

```c
#include "controls.cmp"
```

This model implements a simple control. File `Controls.cmp` (Section 5.4.9) contains interpreted code that defines a new element of type `Control`, and creates an instance of it named `CONTROL`. When executed, this element gathers specific values from the model including ambient total temperature, fan mechanical shaft speed, and TrbH inlet total temperature. It contains a table of corrected percent fan speed versus a power lever angle, contained in variable `CONTROL.PLACS`. It calculates an error, contained in variable `CONTROL.ERR1`, that represents the difference between the actual corrected fan speed (calculated from the model’s mechanical speed and the ambient total temperature) and the scheduled fan speed (determined from the value of `CONTROL.PLACS`). It modifies this error, if necessary, to accomplish three things:

- Limit the corrected fan speed to a maximum value
- Limit the mechanical fan speed to a maximum value
- Limit the TrbH inlet total temperature to a maximum value

The model will later use the solver to vary burner fuel flow such that `CONTROL.ERR1` is zero. The power level of the model will be controlled by specifying values for `CONTROL.PLACS`.

### 5.2.3 Linking Elements

The declarations of the elements do not contain any connection information. All connection information is given in separate link statements. How to do this is discussed in Section 4.3.2.

#### 5.2.3.1 Main Fluid Ports

The following are the main flow element link statements required for the fanjet example.

```c
linkPorts( "InletStart.Fl_O", "InEng.Fl_I",        "F0"    );
linkPorts( "InEng.Fl_O",      "SpltFan.Fl_I",      "F01A"  );
// Primary section
linkPorts( "SpltFan.Fl_O1",   "CmpL.Fl_I",         "F025"  );
linkPorts( "CmpL.Fl_O",       "B025.Fl_I",         "F0251" );
linkPorts( "B025.Fl_O",       "D025.Fl_I",         "F0252" );
linkPorts( "D025.Fl_O",       "CmpH.Fl_I",         "F0253" );
linkPorts( "CmpH.Fl_O",       "BrnPri.Fl_I",        "F03"  );
linkPorts( "FusEng.Fu_O",     "BrnPri.Fu_I",        "FU036" );
// Primary Hot Section Connect Statements:
linkPorts( "BrnPri.Fl_O",     "B041.Fl_I",         "F041"  );
linkPorts( "B041.Fl_O",       "TrbH.Fl_I",         "F041a" );
linkPorts( "TrbH.Fl_O",       "B042.Fl_I",         "F042"  );
linkPorts( "B042.Fl_O",       "D043.Fl_I",         "F043"  );
linkPorts( "D043.Fl_O",       "TrbL.Fl_I",         "F044"  );
linkPorts( "TrbL.Fl_O",       "B045.Fl_I",         "F045"  );
linkPorts( "B045.Fl_O",       "NozPri.Fl_I",        "F07"  );
linkPorts( "NozPri.Fl_O",     "FePri.Fl_I",         "F09"  );
// Fan duct section:
linkPorts( "SpltFan.Fl_O2",   "CmpFSec.Fl_I",       "F12"  );
```
\begin{verbatim}
linkPorts("CmpFSec.Fl_O",  "B025in.Fl_I",       "F11"   
linkPorts("B025in.Fl_O",     "Dfan.Fl_I",         "F16"   
linkPorts("Dfan.Fl_O",       "BFanOB.Fl_I",       "F165"  
linkPorts("BFanOB.Fl_O",     "NozSec.Fl_I",       "F17"   
linkPorts("NozSec.Fl_O",     "FeSec.Fl_I",        "F19"   
\end{verbatim}

Note the following points:

- The splitter primary output flow (\texttt{SplitFan.Fl_O1}) is linked to the \texttt{CmpL} inlet (\texttt{CmpL.Fl_I}), and its secondary output flow (\texttt{SplitFan.Fl_O2}) is linked to the fan inlet (\texttt{Fan.Fl_I}).
- The main stream input and output ports of \texttt{Bleed} elements (e.g. \texttt{B025}) are linked to other main stream components just like any other element would be.
- The burner's fluid input port (\texttt{BrnPri.Fl_I}) is linked to its upstream component, and its fuel input port (\texttt{BrnPri.Fu_I}) is linked to the \texttt{FuelStart} element (\texttt{FusEng.Fu_O}).
- The nozzle fluid output ports are linked to the fluid input ports of \texttt{FlowEnd} elements.

Two special elements in the model, \texttt{PERF} and \texttt{CONTROL}, do not appear anywhere in the \texttt{linkPorts()} statements because they do not have ports.

### 5.2.3.2 Bleed Ports

Recall that \texttt{Bleed} element \texttt{BFanOB} was created with full bleed definition in the element instantiation (see Section 5.2.2.7). A \texttt{BleedOutPort}, also named \texttt{BFanOB}, was created on this element. This port is linked to another element just as any other fluid port would be. In this example, the bleed goes overboard, and therefore the port is linked to a \texttt{FlowEnd} element as follows.

\begin{verbatim}
linkPorts("BFanOB.BFanOB",   "FeFanOB.Fl_I",  "FL166"
\end{verbatim}

The other bleeds are created by means of the macros found in file \texttt{bleed_macros.fnc} and discussed in Section 4.7.5. Two of the macros are used in our example. Bleeds from a \texttt{Compressor} to a \texttt{Bleed} element require the name of the \texttt{Compressor} object (source), the name of the \texttt{Bleed} object (sink), the fraction of compressor inlet flow extracted as bleed, the fractional pressure of the bleed (0 indicates the pressure at compressor inlet, 1 the pressure at compressor exit), the fractional enthalpy of the bleed (analogous to fractional pressure), and the name given to the bleed.

\begin{verbatim}
linkBleedCT("CmpH", "TrbH",  .10, 1., 1., 1., 0, "ca1HPT" 
linkBleedCB("CmpH", "B042",  .04, 1., 1., 1., 0, "ca2HPT" 
\end{verbatim}

Bleeds from one \texttt{Bleed} element to another require the name of the source \texttt{Bleed} object, the name of the sink \texttt{Bleed} object, the fraction of source \texttt{Bleed} object flow extracted as bleed, the enthalpy of the bleed relative to the source \texttt{Bleed} object inlet enthalpy, the pressure of the bleed relative to the source \texttt{Bleed} object inlet pressure, and the name given to the bleed.

\begin{verbatim}
linkBleedBB("B025", "B025in", .00, 1., 1., "surge" 
\end{verbatim}

### 5.2.3.3 Shaft Ports

\texttt{Compressor} and \texttt{Turbine} elements have shaft outlet ports named \texttt{Sh_O} created automatically. The user must connect the shaft inlet ports created on the \texttt{Shaft} elements to the appropriate shaft outlet ports on the other elements.

\begin{verbatim}
linkPorts("CmpL.Sh_O",      "ShL.MeCmpL",   "MeCmpL" 
linkPorts("CmpFSec.Sh_O",   "ShL.MeCmpFSec","MeCmpFSec" 
linkPorts("TrbL.Sh_O",      "ShL.MeTrbL",   "MeTrbL" 
linkPorts("CmpH.Sh_O",      "ShH.MeCmpH",   "MeCmpH" 
linkPorts("TrbH.Sh_O",      "ShH.MeTrbH",   "MeTrbH" 
\end{verbatim}

Notice that a port is never connected to more than one other port. Shafts are normally connected to more than one component (in the example, the LP shaft is connected to the \texttt{TrbL}, the \texttt{CmpL}, and the fan, and the HP shaft is
connected to the TrbH and the CmpH). This requires multiple ports to be created on each shaft element (see sections 4.3.1 and 5.2.2.13).

5.2.4 Solver Setup
Two solver setup items are appropriate for inclusion in the model file:

- Definition of the execution sequence
- Definition of Independent and Dependent objects.

5.2.4.1 Execution Sequence
The execution sequence is the order in which a model's objects are executed. Usually only the model's elements are considered since subelements are automatically executed by the elements to which they belong. It is permissible to specify other objects, such as functions, in the execution sequence to cause them to execute every pass through the cycle. It is often possible, however, to achieve the desired end using other means.

As mentioned, the default execution sequence is the Element objects of the model in the order they were instantiated. For our example, the default sequence is:

1. Ambient element Amb
2. InletStart element InletStart
3. Inlet element Inlet
4. Splitter element SplitFan
5. Compressor element CmpFSec
6. Bleed element B025in
7. Duct element Dfan
8. Bleed element BFanOB
9. Nozzle element NozSec
10. Compressor element CmpL
11. Bleed element B025
12. Compressor element CmpH
13. FuelStart element FusEng
14. Burner element BrnPri
15. Bleed element B041
16. Turbine element TrbH
17. Bleed element B042
18. Duct element Duct43
19. Turbine element TrbL
20. Bleed element B045
21. Nozzle element NozPri
22. FlowEnd element FePri
23. FlowEnd element FeSec
24. FlowEnd element FeFanOB
25. Shaft element ShH
26. Shaft element ShL
27. EngPerf element PERF
28. Control element CONTROL

The default execution sequence for our example does not need to be modified – a benefit of instantiating objects in a sensible order. However, had it been necessary to alter the execution sequence, string array `solver.executionSequence` could have been assigned values as follows:

```c
```
If objects must be executed before the solver runs or after it converges, string array variables solver.preExecutionSequence and solver.preExecutionSequence can be defined similarly.

5.2.4.2 Definition of Independents and Dependents

Generally, Independent and Dependent objects should only be defined in the model file, not added to the solver. This is because different modes of operation, as defined in the case input file, may change which independents and dependents should be active.

Various elements and subelements flag certain variables to be made part of a solver Independent or Dependent object, and added to the solver, when autoSolverSetup() is run. In our example, the following solver setup would result for the off design case.

**Independents**
1. InletStart.W
2. SplitFan.BPR
3. CmpFSec.S_map.RlineMap
4. Cmpl.S_map.RlineMap
5. CmplH.S_map.RlineMap
6. CmplH.S_Qhx.Tmat
7. TrbH.PRbase
8. TrbH.S_Qhx.Tmat
9. TrbL.PRbase
10. ShH.Nmech
11. ShL.Nmech

**Dependents**
1. CmpFSec.Wc = CmpFSec.WcCalc
2. Cmpl.Wc = CmpL.WcCalc
3. CmplH.Wc = CmpH.WcCalc
5. TrbH.Wp = TrbH.WpCalc
6. TrbH.S_Qhx.TgasPath = TrbH.S.Qhx.Tmat
7. TrbL.Wp = TrbL.WpCalc
8. NozSec.WqAE = NozSec.WqAEdem
9. NozPri.WqAE = NozSec.WqAEdem
10. ShH.trqNet = 0
11. ShL.trqNet = 0

The number of independents equals the number of dependents, and all the physics of the model are captured, so there is no need to create additional solver quantities. This is normally the case when autoSolverSetup() is used (see Section 4.8.1). If a model has special features not captured by autoSolverSetup(), the additional Independent and Dependent objects can be defined in the model file, with their autoSetup attribute set to TRUE (see Section 4.8.3). This way, the additional quantities will be added to the solver when autoSolverSetup() is run.

Special purpose independents and dependents, needed only for certain run cases, can be defined and added to the solver in the case input file.

5.3 Case Input File

After the model has been defined the next grouping of inputs are referred to as case input. Case input contains assignments to model parameters, calls to built-in and user defined global functions, conditional logic, and other input required to control the execution of the model. The inclusion of independents and dependents to the solver are also part of the case definition. In special situations, new elements can be instantiated, links can broken and made, and the execution sequences changed (see sections 4.11 and 4.12).
5.3.1 Design Point

Our example runs the cases defined in file desOD.case (see Section 5.1.5). It begins with a design point case.

```plaintext
SetOption("switchDes", "DESIGN");
autoSolverSetup();
CASE = CASE + 1;
run();
ncpView.display();
```

By calling `setOption()` from global scope, as above, the `switchDes` variable in all objects throughout the model (where it exists) is set to `DESIGN` (see Section 4.8.1). This mode calculates geometry and scale factors to match specified aerothermodynamic conditions. In design mode, all gas flow stations (`InletStart.Fl_O`, `CmpFSec.Fl_I`, etc.) will calculate an area corresponding to the value of the station's `MNdes` variable, if it was specified. CompressorMap and TurbineNeppMap subelements use specified design values to calculate map scalers. Each Nozzle element calculates a throat area to match the design point flow.

As discussed in Section 4.8.1, the value of an object's `switchDes` variable often affects which independents and dependents it signals to be given to the solver. Consequently, `autoSolverSetup()` should always be called after the call to `setOption()`.

The predefined global variable `CASE` (an integer) is often used as a numeric label for cases run (see Section 4.10). The system initializes it to zero, and it must be manually incremented by the user.

The call to `run()` from the top-level assembly executes the entire model, including the solver. Since no operating conditions have been defined in the case input file, those defined in the instantiation of the model will be used. These are:

- Altitude = 35000 ft, standard day (dTs = 0), Mach number = 0.8 (see Section 5.2.2.1).
- Total inlet flow of 100 lbm/sec (not changed from input value because of design mode).
- Bypass ratio of 5.0 (not changed from input value because of design mode; see Section 5.2.2.4).
- Fuel flow of 0.25 lbm/sec = 900 lbm/hr (see Section 5.2.2.9).
- HP shaft mechanical speed of 8997.43 rpm, LP shaft mechanical speed of 3497.40 rpm (not changed because of design mode; see Section 5.2.2.13).

The study executive file included file `ncp.view` (see Section 5.1.2). This file defines an object of type `PageViewer` (see Section 4.9.2.4) named `ncpView`. `PageViewer ncpView` sends its output to `OutFileStream pvStream`. File `ncp.view` associates this stream with file `viewOut`. Thus, the command `ncpView.display()` causes output from the design case run to be written to a file named `viewOut` in the same directory from which the NPSS run was initiated (this output is shown in Section 5.4.11).

5.3.2 Offdesign Steady-state Points

```plaintext
setOption( "switchDes", "OFFDESIGN" );
autoSolverSetup();
CASE = CASE + 1;
run();
ncpView.display();
```

Function `setOption()` is run again to change all objects to offdesign mode. Function `autoSolverSetup()` is run again to make any changes to the solver necessitated by the new mode. Variable `CASE` is manually indexed and then the model itself is executed. Finally, the results are written to file `viewOut` using `DataViewer ncpView`.

More advanced offdesign running is accomplished next using the special element instance named `CONTROL` (see Section 0).
5 Building a Model

```cpp
varName = "BrnPri.Wfuel";
}

Dependent ControlErr {
    eq_lhs = "CONTROL.ERR1";
    eq_rhs = "0.0";
}
solver.addIndependent( "FuelControl" );
solver.addDependent( "ControlErr" );

CONTROL.PLACS = 50;
solver.maxIterations = 50;
run();
ncpView.display();
CONTROL.PLACS = 45;
run();
ncpView.display();
CONTROL.PLACS = 40;
run();
ncpView.display();
CONTROL.PLACS = 35;
run();
ncpView.display();
CONTROL.PLACS = 30;
run();
ncpView.display();

The solver is given control of the burner fuel flow, and is instructed to converge such that CONTROL.ERR1 is zero. Variable CONTROL.PLACS is then set to various values causing element CONTROL to seek to run to a schedule of corrected fan speed, subject to certain limits (see Section 0). After each point, the results are written to the output file using ncpView. Since CASE was not manually indexed after each point, its value remains constant at 2.

5.3.3 Transient Points

```cpp
solver.removeIndependent( "FuelControl" );
solver.removeDependent( "ControlErr" );

CONTROL.PLACS = 50;
setOption( "solutionMode", "TRANSIENT" );

transient.stopTime = 10;
transient.baseTimeStep = .01;
solver.postExecutionSequence = { "transientTrace" };
run();
transientTrace.display();
```

The solver items used with the control element in steady-state are first removed. The control element power lever angle is then set to 50, corresponding to 100% corrected fan speed. Function setOption() is then called to set all option variables named solutionMode throughout the model to TRANSIENT. This sets the solver to run in transient mode (see Chapter 6). The special control element instantiated as object CONTROL also contains an option variable named solutionMode that has TRANSIENT as a valid value. In this mode, the element changes fuel flow over time.
toward the goal condition specified by CONTROL.PLACS. The transient is set to end at a time of 10 seconds, and to increase time at 0.01 second intervals.

The study executive file included a local file named Transient.view (see sections 5.1.3 and 5.4.2). It defines a CaseColumnViewer (see Section 4.9.2.3) named transientTrace that stores time, engine inlet airflow, CmpH exit total pressure and total temperature, HP and LP shaft mechanical speeds, and CmpH material temperature in columns whenever it is executed. When its display() function is called, its output is sent to OutFileStream transientStream which file Transient.view connects to a file named tout. Viewer transientTrace is made a member of string array solver.postExecutionSequence which will cause it to execute whenever the solver converges. These executions simply store the data from each converged case (each time step) specified by the viewer. After the transient run is complete, function transientTrace.display() produces the output file (a portion of which is shown in Section 5.4.12).

5.4 fanjet Source Files

The following sections contain listings of the files used in the fanjet example discussed above.

5.4.1 fanjet.run

//Hi Bypass Fanjet model
//Created: 7/2000

 ifndef THERMO
 define THERMO GasTbl
 endif

 //Set the thermo package
 setThermoPackage("$THERMO");

 //default includes
 include "nasa.view"       //default page viewer
 include "bleed_macros.fnc" //bleed macros

 //local includes
 include "Transient.view"      //transient viewer

 //model file
 include "fanjet.mdl"

 //case input files
 include "desOD.case"

5.4.2 Transient.view

// **** T R A N S I E N T   C A S E   C O L U M N   V I E W E R  ****

OutFileStream transientStream { filename = "tout"; }

DataViewer CaseRowViewer transientTrace {

titleBody = " ";
titleVars = { };
variableList = {
 "time: ??..???? " ,
 "Amb.W : ???.?? = Air Flow", 
 "CmpH.Fl.O.Pt : ????..?? = Pt3", 
 "CmpH.Fl.O.Tt : ????..?? = Tt3", 
 "ShH.Nmech : ?????..?? = HPSpeed", 
 "ShL.Nmech : ?????..?? = LPspeed", 
 "CmpH.S_Qhx.Tmat : ????..?? = Tmat" 
}
5.4.3 fanjet.mdl

// F A N J E T E N G I N E

// File Name: fanjet.mdl
// Version: 2.0
// Date(s): 11 November 1998
// 5 Dec 2006 - ARP5571 standard nomenclature
// Aug 2011 - New Turbine and Compressor element
// and subelement architecture

AUTHOR = " ";
MODELNAME = "fanjet";
string Model_ID = "fanjet for documentation example";

Element Ambient Amb {
    switchMode = "ALDTMN";
    alt_in    = 35000.;
    dTs_in    = 0.;
    MN_in     = 0.8;
} // END Amb

Element InletStart InletStart{
    AmbientName = "Amb";
    W_in = 100.;
}

Element Inlet InEng {
    eRamBase = 0.995;
} // END InEng

Element Splitter SpltFan

//
Element Splitter SpltFan {
  BPRdes = 5.;
} // END SpltFan

// -----------------------------------------------
//      Compressor CmpFSec
// -----------------------------------------------

Element Compressor CmpFSec {
  // Load map file.
  // This plugs the CompressorRline subelement into the S_map socket,
  // defines S_map tables TB_eff, TB_Pr, TB_Wc, and sets values
  // for S_map.alphaMapDes, S_map.RlineMapDes, and S_map.NcMapDes
  #include "fanE3.map";
  PRdes = 1.5;
  effDes= 0.8589;
  Sh_O.inertia = 10.; // slugs-ft**2.

} //END CmpFSec

// -----------------------------------------------
//      Bleed B025in
// -----------------------------------------------

Element Bleed B025in {

} // END Bleed B025in

// -----------------------------------------------
//      Duct Dfan
// -----------------------------------------------

Element Duct Dfan {

} // END FanDuct

// -----------------------------------------------
//      Bleed BFanOB
// -----------------------------------------------

Element Bleed BFanOB {
  BleedOutPort BFanOB {
    hscale = 1.0;
    Pscale = 1.0;
    fracW = .00;
  }

} // END Bleed BFanOB

// -----------------------------------------------
//      Nozzle NozSec
// -----------------------------------------------

Element Nozzle NozSec {
  PsExhName = "Amb.Ps";
} //END NozSec

// -----------------------------------------------
//      Compressor CmpL
// -----------------------------------------------
Element Compressor Cmpl { 
    // Load map file.
    // This plugs the CompressorRline subelement into the S_map socket,
    // defines S_map tables TB_eff, TB_PR, TB_Wc, and sets values
    // for S_map.alphaMapDes, S_map.RlineMapDes, and S_map.NcMapDes
    #include "lpcE3.map";
    PRdes  = 3.0;
    effDes = 0.872;
    Sh_O.inertia = 10.;   // slugs-ft**2.
} // END Cmpl

// ------------------------------
//      Bleed B025
// ------------------------------

Element Bleed B025 { 
} // END Bleed

// <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
//      Duct D025
// <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Element Duct D025{
    switchDP="OFF";
    dPqP_in=0.002;
    //numRakes=1;
} // END D025

// ------------------------------
//      Compressor CmpH
// ------------------------------

Element Compressor CmpH { 
    // Load map file.
    // This plugs the CompressorRline subelement into the S_map socket,
    // defines S_map tables TB_eff, TB_PR, TB_Wc, and sets values
    // for S_map.alphaMapDes, S_map.RlineMapDes, and S_map.NcMapDes
    #include "hpcE3.map";
    PRdes  = 10.0;
    effDes = 0.8522;
    Sh_O.inertia = 10.;   // slugs-ft**2.

    Subelement ThermalMass S_Qhx { 
        Ahx = 35743.5;
        CpMat = 1.0;
        ChxDes = .0003858;
        massMat = 500;
        wtdAvg_Fl = .5;
    }
} //END CmpH

// ------------------------------
//      FuelStart FusEng
// ------------------------------
Element FuelStart FusEng {
    hFuel = 182.7175;
} // END FusEng

// ------------------------------
//      Burner BrnPri
// ------------------------------
Element Burner BrnPri {
    dPqPfBase = 0.05;
    effBase = 0.98;
    switchHotLoss = "input";
    switchBurn = "FUEL";
    Wfuel = .35;
    tolRayleigh = 0.0001;
} // END BrnPri

// ------------------------------
//      Bleed B041
// ------------------------------
Element Bleed B041 {
}

// ------------------------------
//      Turbine TrbH
// ------------------------------
Element Turbine TrbH {
    // Load map file.
    // This plugs the TurbinePRmap subelement into the S_map socket,
    // defines S_map tables TB_eff and TB_Wp, and sets values
    // for S_map.PRmapDes and S_map.NpMapDes
    #include "hptE3.map";
    PRbase = 4.984; // pressure ratio initial guess
    effDes = 0.89;
    Subelement ThermalMass S_Qhx {
        Ahx = 3574.35;
        CpMat = 1.0;
        ChxDes = .0003858;
        massMat = 100;
        wtdAvg_Fl = .5;
    }
} //END TrbH

// ------------------------------
//      Bleed B042
// ------------------------------
Element Bleed B042 {
}

// ------------------------------
//      Duct D043
Element Duct D043 {
    switchDP = "INPUT";
    void preexecute() {
        dPqP_in = 0.25 * Fl_I.MN * Fl_I.MN;
    }
    Fl_I.MNdes = 0.4;
} //END D043

Element Turbine TrbL {
    // Load map file.
    // This plugs the TurbinePRmap subelement into the S_map socket,
    // defines S_map tables TB_eff and TB_Wp, and sets values
    // for S_map.PRmapDes and S_map.NpMapDes
    #include "lptE3.map";
    PRbase = 4.2915; // pressure ratio initial guess
    effDes = 0.8777;
} //END TrbL

Element Bleed B045 {
}

Element Nozzle NozPri {
    PsExhName = "Amb.Ps";
} //END NozPri

Element FlowEnd FePri {
}

Element FlowEnd FeSec {
}

Element FlowEnd FeFanOB {
}

Element Shaft ShH {
    // Mechanical Ports. These are created as needed on the shaft.
ShaftInputPort MeCmpH, MeTrbH;
  Nmech = 8997.43;
inertia = .93243;       //30. lbm-ft2;
}   //END ShH

// ------------------------------
//      Shaft ShL
// ------------------------------

Element Shaft ShL {
  ShaftInputPort MeCmpFSec, MeCmpL, MeTrbL;
  Nmech = 3497.40;
inertia = 2.73513;     //88. lbm-ft2;
}   //END ShL

// ------------------------------
//      PerfNASA Perf
// ------------------------------

Element EngPerf Perf {
}   //END Perf

// ------------------------------
//      Cycle Cycle
// ------------------------------

Element Cycle Cycle {
  EPR_numName = "CmpH.Fl_O";
  EPR_denName = "InEng.Fl_O";
  FPR_numName = "CmpFSec.Fl_O";
  FPR_denName = "InEng.Fl_O";
}   //END Cycle

// ------------------------------
//      Controls
// ------------------------------
#include "controls.cmp"

// ------------------------------
//      linkPorts
// ------------------------------
linkPorts( "InletStart.Fl_O", "InEng.Fl_I",        "F0"    );
linkPorts( "InEng.Fl_O",      "SpltFan.Fl_I",      "F01A"  );
// Primary section
linkPorts( "SpltFan.Fl_O1",   "CmpL.Fl_I",         "F025"  );
linkPorts( "CmpL.Fl_I",       "B025.Fl_I",         "F0251" );
linkPorts( "B025.Fl_I",       "D025.Fl_I",         "F0252" );
linkPorts( "D025.Fl_I",       "CmpH.Fl_I",         "F0253" );
linkPorts( "CmpH.Fl_O",       "BrnPri.Fl_I",       "F03"   );
linkPorts( "FusEng.Fu_O",     "BrnPri.Fu_I",       "FU036" );
// Primary Hot Section Connect Statements:
linkPorts( "BrnPri.Fl_O",     "B041.Fl_I",         "F041"  );
linkPorts( "B041.Fl_I",       "TrbH.Fl_I",         "F041a" );
linkPorts( "TrbH.Fl_I",       "B042.Fl_I",         "F042"  );
linkPorts( "B042.Fl_I",       "D043.Fl_I",         "F043"  );
linkPorts( "D043.Fl_I",       "TrbL.Fl_I",         "F044"  );
```csharp
linkPorts("TrbL.Fl_O", "B045.Fl_I", "F045" );
linkPorts("B045.Fl_O", "NozPri.Fl_I", "F07" );
linkPorts("NozPri.Fl_O", "FePri.Fl_I", "F09" );

// Fan duct section:
linkPorts("SpltFan.Fl_O", "CmpFSec.Fl_I", "F12" );
linkPorts("CmpFSec.Fl_O", "B025in.Fl_I", "F11" );
linkPorts("B025in.Fl_O", "Dfan.Fl_I", "F16" );
linkPorts("Dfan.Fl_O", "BFanOB.Fl_I", "F165" );
linkPorts("BFanOB.Fl_O", "NozSec.Fl_I", "F17" );
linkPorts("NozSec.Fl_O", "FeSec.Fl_I", "F19" );

// -------------------------------
//      Bleed Connections
// -------------------------------
// Component to Component Bleed
linkBleedCT("CmpH", "TrbH", .10, 1., 1., 1., 0, "ca1HPT");
linkBleedCB("CmpH", "B042", .04, 1., 1., "ca2HPT");
linkBleedCB("CmpH", "B045", .01, .5, .5, "ca3HPT");
linkBleedBB("B025", "B025in", .00, 1., 1., "surge");

//Overboard bleed
linkPorts("BFanOB.BFanOB", "FeFanOB.Fl_I", "FL166");

// -------------------------------
//      Shaft Connect Statements
// -------------------------------
linkPorts("CmpL.Sh_O", "ShL.MeCmpL", "MeCmpL");
linkPorts("CmpFSec.Sh_O", "ShL.MeCmpFSec","MeCmpFSec");
linkPorts("TrbL.Sh_O", "ShL.MeTrbL", "MeTrbL");
linkPorts("CmpH.Sh_O", "ShH.MeCmpH", "MeCmpH");
linkPorts("TrbH.Sh_O", "ShH.MeTrbH", "MeTrbH");

// <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
//        Instrumentation Elements
// <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Element Instrument HUMSE {
  baseName="USERSET";
  base=0.;
  switchError="DELTA";
  tol=1.;

  measName = "DB.HUMSE";
  switchSolverActive = "Never";
}

Element Instrument PAMBSE {
  switchError="RATIO";
  tol=0.1;
  baseName="Amb.Ps";

  measName = "DB.PAMBSE";
  indName = "Amb.alt_in";
  indRef = "10000";
  switchSolverActive = "Never";
}
```

---

5 Building a Model
5 Building a Model

Element Instrument TAMBSE {
    switchError="DELTA";
    tol=10.;
    basePathName="Amb.Ts";

    measName = "DB.TAMBSE";
    indName = "Amb.dTs_in";
    switchSolverActive = "Never";
    indRef = "1000";
} // END TAMBSE Instrument

Element Instrument P25SE {
    switchError="RATIO";
    tol=0.1;
    basePathName="F025.Pt";

    measName = "DB.P25SE";
    indName = "InEng.eRamBase";
    switchSolverActive = "Never";
    indRef = "1";
} // END P25SE Instrument

Element Instrument T25SE {
    switchError="DELTA";
    tol=20.;
    basePathName="F025.Tt";

    measName = "DB.T25SE";
    indName = "Amb.MN_in";
    switchSolverActive = "Never";
    indRef = "1";
} // END T25SE Instrument

Element Instrument P3SE {
    switchError="RATIO";
    tol=0.1;
    basePathName="F03.Pt";

    measName = "DB.P3SE";
    indName = "CmpH.S_map.TB_Wc.s_rtn";
} // END P3SE Instrument

Element Instrument T3SE {
    switchError="DELTA";
    tol=20.;
    basePathName="F03.Tt";

    measName = "DB.T3SE";
    indName = "CmpH.S_map.TB_eff.s_rtn";
} // END T3SE Instrument

Element Instrument WF36SE {
switchError="RATIO";
tol=0.15;
baseName="USERSET";
measName="DB.WF36SE";
indName = "TrbH.S_map.TB_eff.s rtn";
void preexecute(){
    base=BrnPri.Wfuel * 3600.;
}
}  //  END WF36SE Instrument

Element Instrument P42SE {
    switchError="RATIO";
tol=0.2;
baseName="F042.Pt";
measName = "DB.P42SE";
indName = "NozPri.CdTh_in";
}  //  END P42SE Instrument

Element Instrument T42SE {
    switchError="DELTA";
tol=100.;
baseName="DB.T42SE";
measName="USERSET";
indName = "T42SE.a_measAdj";
indRef = "1";
void preexecute() { meas=F042.Tt; }
}  //  END T42SE Instrument

Element Instrument N25SE {
    switchError="RATIO";
tol=0.1;
baseName="ShH.Nmech";
measName = "DB.N25SE";
indName = "BrnPri.Wfuel";
switchSolverActive = "Never";
}  //  END N25SE Instrument

// <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
//        DataBase Read Element
// <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Element ExternalDB DB {
    Subelement wsfr S_ExtDB {
    }
}
solver.debugLevel = "ITERATIONDETAILS";
solver.diagnosticFile = "solver.diag";

5.4.4 fanE3.map
Excerpts from this file are shown and discussed in Section 5.2.2.6.

5.4.5 lpcE3.map
This file is quite similar to fanE3.map excerpted and discussed in Section 5.2.2.6.

5.4.6 hpcE3.map
This file is quite similar to fanE3.map excerpted and discussed in Section 5.2.2.6.

5.4.7 hptE3.map
Because of the length of this file, excerpts are shown as follows.

Subelement TurbinePRmap S_map {
    PRmapDes = 4.975; // unscaled map design point pressure ratio
    NpMapDes = 100.0; // unscaled map design point speed
    Table TB_Wp (real SPED, real PR) {
        SPED = 80.0 {
            PR = { 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 4.75, 4.96, 5.25, 5.5, 5.75, 6.0 }
            FLOW = { 15.249, 15.749, 15.894, 15.907, 15.907, 15.907, 15.907, 15.907, 15.907, 15.907, 15.907, 15.907 }
        }
        SPED = 90.0 {
            PR = { 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 4.75, 4.96, 5.25, 5.5, 5.75, 6.0 }
            FLOW = { 15.226, 15.674, 15.828, 15.862, 15.863, 15.863, 15.863, 15.863, 15.863, 15.863, 15.863, 15.863 }
        }
        SPED = 100.0 {
            PR = { 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 4.75, 4.96, 5.25, 5.5, 5.75, 6.0 }
            FLOW = { 15.208, 15.581, 15.720, 15.761, 15.765, 15.765, 15.765, 15.765, 15.765, 15.765, 15.765, 15.765 }
        }
    }
    SPED_interp = "lagrange3";
    SPED_extrap = "none";
    PR_interp = "lagrange3";
    PR_extrap = "linear";
    extrapIsError = 0;
    printExtrap = 0;
} // end TB_Wp

Table TB_eff (real SPED, real PR) {
    SPED = 80.0 {
        PR = { 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 4.75, 4.96, 5.25, 5.5, 5.75, 6.0 }
        EFF = { 0.86, 0.889, 0.903, 0.908, 0.91, 0.91, 0.909, 0.909, 0.908, 0.907, 0.906, 0.905 }
    }
    SPED = 90.0 {
        PR = { 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 4.75, 4.96, 5.25, 5.5, 5.75, 6.0 }
        EFF = { 0.843, 0.883, 0.903, 0.912, 0.916, 0.919, 0.919, 0.919, 0.918, 0.918, 0.918, 0.918 }
    }
    SPED = 100.0 {
        PR = { 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 4.75, 4.96, 5.25, 5.5, 5.75, 6.0 }
    }
}
EFF = { 0.82, 0.869, 0.896, 0.908, 0.916, 0.92, 0.921, 0.922, 0.922, 0.923, 0.923, 0.924 }

5.4.8 IptE3.map
This file is quite similar to hptE3.map, excerpted in Section 5.4.7.

5.4.9 Controls.cmp

```java
class Control extends Element {

    Element Control CONTROL {

        // Variables and limits
        real PLACS;
        real T1ACS;
        real T25CS;
        real P1ACS;
        real P25CS;
        real P53CS;
        real T41CS;
        real P7CS;
        real T7CS;
        real WF36CS;
        real TH1A;

        real T41H = 3206;
        real WFEL = 430;

        real PCNHR = 105;
        real PCNH = 105;

        real PCNHRH;
        real XN25CS;
        real PCNHRS;
        real EPCNHR;
        real XMDHP;
        real EPCNH;
        real EPNHRH;

        real ET41;
        real ERR1;

        real XNHRED = 13000;
        real WFED;
        // Frame rate
        real dtime = .01;

        // Execute time
        real execTime = 0;

        real C_TSTD = 518.6700;
        void variableChanged(string name, string value) {
            // do nothing...
        }

        Option switchDes {
            allowedValues = { "DESIGN", "OFFDESIGN" }
        }
    }
}
```

// end TB_eff
} // end S_map
value = "DESIGN";
}

Option solutionMode {
  allowedValues = { "TRANSIENT", "STEADY_STATE", "ONE_PASS" }
  value = "STEADY_STATE";
}

// Get data from engine to control
void mapIn() {
  XN25CS = ShL.Nmech;
  T1ACS = FL0.Tt;
  T25CS = FL1A.Tt;
  P1ACS = FL0.Pt;
  P25CS = FL1A.Pt;
  PS3CS = FL3.Ps;
  T41CS = F41.Tt;
  P7CS = FL7.Pt;
  T7CS = FL7.Tt;
  WF36CS = BrnPri.Wfuel;
}

// Get data from control to engine
void mapOut() {
  if ( solutionMode == "TRANSIENT" ) { BrnPri.Wfuel = WFED * BrnPri.Wfuel;}
}

void calculate() {
  //steady-state mode
  if ( solutionMode == "STEADY_STATE" ) {
    mapIn();
    steadyState();
  }
}

// Steady-state
// Schedule fuel flow based on limits (from GE)
void steadyState() {
  TH1A = T1ACS / C_TSTD;
  if ( switchDes == "DESIGN" ) {
    XNDHP = XN25CS/ sqrt( TH1A );
  }

  // Calculate corrected speed
  TH1A = T1ACS / C_TSTD;
  PCNHRS = 100 * XN25CS / sqrt( TH1A ) / XNDHP;

  // Read demand fan speed from PLA
  PCNHR = PC_SKED( PLACS );

  // Calculate error between demand fan speed mand sensed fan speed
  EPCNHR = ( PCNHR – PCNHRS )/ 100.0;

  // Check max corrected fan speed limit
  EPCNHH = ( PCNHR – PCNHRS )/ 100;
  if ( EPCNHR > EPCNHH ) {
    EPCNHR = EPCNHH;
  }

  // Check max corrected fan speed as a function of physical speed
PCNHRH = PCNHH * XNHRD / sqrt( TH1A )/ XNDHP;
EPNHRH = ( PCNHRH - PCNHRS )/ 100;
if ( EPCNHR > EPNHRH ) {
  EPCNHR = EPNHRH;
}

// Calculate error due to T41 max
ET41 = ( T41H - T41CS )/ T41H;

ERR1 = EPCNHR;
if ( EPCNHR > ET41 ) {
  ERR1 = ET41;
}

// Discrete calcs (done once per time step)
int runDiscreteCalcs( real time, real nextCalcTime ) {
  if ( abs ( nextCalcTime - execTime ) > .000001 ) {
    return FALSE;
  }

  // Check steady-state limits
  steadyState();
  real step = min( abs ( ERR1 ), .005 );
  WFED = ( 1 + ERR1 / abs ( ERR1 ) * step );

  // Map data from control to engine
  mapOut();
  return TRUE;
}

// Schedule of comp speed vs power code
Table PC_SKED( real PL ) {
  PL = { 25, 40, 50 }
  PC_SKED = { 80, 90, 100 }
}

// Calculations performed after the cycle converges
void updateHistory( ) {
  if ( abs ( time - execTime ) > .000001 ) {
    return;
  }

  // Determine the next time this executes
  execTime = execTime + dtime;

  // Get the data from the engine to the control
  mapIn();
}

// Initialize
void initializeHistory() {
  execTime = time + dtime;
  // Get the data from the engine to the control
  mapIn();
}
5.4.10 desOD.case

//-------------------------------------------------
// Run a design and off design case
//-------------------------------------------------

setOption( "switchDes", "DESIGN" );
autoSolverSetup();

CASE = CASE + 1;
run();
ncpView.display();

setOption( "switchDes", "OFFDESIGN" );
autoSolverSetup();

CASE = CASE + 1;
run();
ncpView.display();

//-------------------------------------------------
// Run a throttle hook using the controller
//-------------------------------------------------

Independent FuelControl {
    varName = "BrnPri.Wfuel";
}

Dependent ControlErr {
    eq_lhs = "CONTROL.ERR1";
    eq_rhs = "0.0";
}

solver.addIndependent( "FuelControl" );
solver.addDependent( "ControlErr" );

CONTROL.PLACS = 50;
solver.maxIterations = 50;
run();
ncpView.display();

CONTROL.PLACS = 45;
run();
ncpView.display();

CONTROL.PLACS = 40;
run();
ncpView.display();

CONTROL.PLACS = 35;
run();
ncpView.display();

CONTROL.PLACS = 30;
run();
ncpView.display();

//-------------------------------------------------
// Transient: Control-governed snap accel to full power
//-------------------------------------------------

solver.removeIndependent( "FuelControl" );
solver.removeDependent( "ControlErr" );

CONTROL.PLACS = 50;
setOption( "solutionMode", "TRANSIENT" );
transient.stopTime = 10;
transient.baseTimeStep = .01;
solver.postExecutionSequence = { "transientTrace" };

run();
transientTrace.display();

5.4.11 viewOut
This file contains output from the default output viewer (see the following pages). Only the first two points are shown (the design point and the first offdesign point, which are identical).
**INPUT FLOW**

<table>
<thead>
<tr>
<th>MN</th>
<th>alt</th>
<th>dTs</th>
<th>W</th>
<th>Fg</th>
<th>Fn</th>
<th>SFC</th>
<th>Wfuel</th>
<th>WAR</th>
<th>OPR</th>
</tr>
</thead>
</table>
| 0.800    | 35000.0| 0.00       | 100.0    | 4270.0    | 1850.0   | 0.6811    | 1260.00   | 0.0000    | 29.85 |}

**INPUTS**

<table>
<thead>
<tr>
<th>eRam</th>
<th>Afs</th>
<th>Fram</th>
<th>Wc</th>
<th>Wp</th>
<th>PR</th>
<th>TR</th>
<th>effPoly</th>
<th>eff</th>
<th>Nc</th>
<th>Np</th>
<th>pwr</th>
</tr>
</thead>
<tbody>
<tr>
<td>InEng</td>
<td>0.9950</td>
<td>780.44</td>
<td>2420.1</td>
<td>CmpFSec</td>
<td>216.11</td>
<td>1.500</td>
<td>1.1433</td>
<td>0.8668</td>
<td>0.8589</td>
<td>3778.2</td>
<td>-1797.9</td>
</tr>
</tbody>
</table>
NCP                   NPSS_2.4    model:         fanjet     run by:     John Doe    solutionMode= STEADY_STATE    converge=    1    case:    2
time:  0.000    timeStep:0.0500     therm_package:GasTbl      Mode: OFFDESIGN         iter/pas/Jac/Broy=  1/  1/ 0/ 0     run: 08/08/11 14:26:11

Summary Output Data
MN       alt    dTs        W        Fg        Fn     SFC       Wfuel      WAR      OPR
0.800   35000.0     0.00    100.0    4270.0    1850.0   0.6811    1260.00   0.0000   29.850

INPUT FLOW
W        Pt        Tt       ht     FAR       Wc        Ps        Ts     Aphy      MN      gamt
F0     InEng.Fl_I   100.00     5.272    444.43   106.17  0.0000   258.04     3.458    393.85    780.4  0.8000   1.40134
F12    CmpFSec.F>    83.33     5.246    444.43   106.17  0.0000   216.11     0.000      0.00      0.0  0.0000   1.40134
F11    B025in.Fl>    83.33     7.868    508.11   121.42  0.0000   154.05     0.000      0.00      0.0  0.0000   1.40069
F16    Dfan.Fl_I     83.33     7.868    508.11   121.42  0.0000   154.05     0.000      0.00      0.0  0.0000   1.40069
F165   BFanOB.Fl>    83.33     7.868    508.11   121.42  0.0000   154.05     0.000      0.00      0.0  0.0000   1.40069
F17    NozSec.Fl>    83.33     7.868    508.11   121.42  0.0000   154.05     0.000      0.00      0.0  0.0000   1.40069
F025   CmpL.Fl_I     16.67     5.246    444.43   106.17  0.0000    43.22     0.000      0.00      0.0  0.0000   1.40134
F0251  B025.Fl_I     16.67    15.737    632.47   151.28  0.0000    17.19     0.000      0.00      0.0  0.0000   1.39836
F0252  D025.Fl_I     16.67    15.737    632.47   151.28  0.0000    17.19     40.120   1980.97     52.5  0.3999   1.39836
F0253  CmpH.Fl_I     16.67    15.737    632.47   151.28  0.0000    17.19     0.000      0.00      0.0  0.0000   1.39836
F041   B041.Fl_I     14.52   149.499   2770.36   747.72  0.0247     3.30     0.000      0.00      0.0  0.0000   1.29138
ca1HPT TrbH.ca1H>     1.67   157.368   1296.48   316.04  0.0000     0.25     0.000      0.00      0.0  0.0000   1.36211
F041a  TrbH.Fl_I     14.52   149.499   2770.36   747.72  0.0247     3.30     0.000      0.00      0.0  0.0000   1.29138
F042   B042.Fl_I     16.18    44.508   2058.73   534.42  0.0221    10.65     0.000      0.00      0.0  0.0000   1.31068
F043   D043.Fl_I     16.85    44.508   2030.61   525.78  0.0212    11.01   40.120   1980.97     52.5  0.3999   1.31223
F07    NozPri.Fl>    17.02    13.916   1599.63   404.07  0.0210    31.56     0.000      0.00      0.0  0.0000   1.33072
F09    FePri.Fl_I    17.02    13.916   1599.63   404.07  0.0210    31.56     3.458   1119.32    115.8  1.5705   1.33072
F19    FeSec.Fl_I    83.33     7.868    508.11   121.42  0.0000   154.05     3.458    401.52    456.7  1.1503   1.40069
FL166  FeFanOB.F>    0.00    7.868    508.11   121.42  0.0000   154.05     0.000      0.00      0.0  0.0000   1.40069

COMPRESSION & TURBINES
Wc|WpMap   PRmap                     effMap   Nc|NpMap    Rline
SPLITTERS
BPR   dP/P1   dP/P2                                    ADDERS AND SCALARS
BrnPri        2770.36   0.9800   0.0500    0.35000   0.02471                      s_Wc|WpAud  a_Wc|WpAud  s_PRaud  a_PRaud  s_effAud  a_effAud
CmpFSec       1.0000      0.0000   1.0000   0.0000    1.0000    0.0000
CmpL             1.0000      0.0000   1.0000   0.0000    1.0000    0.0000
CmpH             1.0000      0.0000   1.0000   0.0000    1.0000    0.0000
TrbH       1.0000   0.0000   1.0000   0.0000    1.0000    0.0000
TrbL       1.0000   0.0000   1.0000   0.0000    1.0000    0.0000
CmpH.ca1HPT         0.0400   1.0000   1.0000    1296.48     316.04
CmpH.ca2HPT         0.0100   0.5000   0.5000     970.50     233.66
CmpH.ca3HPT         0.0200   0.8000   0.8000     970.50     233.66

5 Building a Model
### 5.4.12 tout

Because of the large number of transient points run, this file is lengthy. Excerpts are shown here.

<table>
<thead>
<tr>
<th>Time</th>
<th>Air Flow</th>
<th>Pt3</th>
<th>Tt3</th>
<th>HPspeed</th>
<th>LPspeed</th>
<th>Tmat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 2</td>
<td>0.0500</td>
<td>100.00</td>
<td>157.10</td>
<td>1298.57</td>
<td>8997.25</td>
<td>3497.28</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.1000</td>
<td>100.00</td>
<td>156.91</td>
<td>1299.05</td>
<td>8997.07</td>
<td>3497.16</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.1500</td>
<td>99.99</td>
<td>156.70</td>
<td>1299.62</td>
<td>8993.86</td>
<td>3495.12</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.2000</td>
<td>99.98</td>
<td>156.49</td>
<td>1300.20</td>
<td>8993.68</td>
<td>3494.68</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.2500</td>
<td>99.97</td>
<td>156.29</td>
<td>1300.76</td>
<td>8995.00</td>
<td>3495.85</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.3000</td>
<td>99.95</td>
<td>156.06</td>
<td>1301.42</td>
<td>8993.86</td>
<td>3495.12</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.3500</td>
<td>99.94</td>
<td>155.84</td>
<td>1302.08</td>
<td>8992.74</td>
<td>3494.41</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.4000</td>
<td>99.92</td>
<td>155.62</td>
<td>1302.74</td>
<td>8991.62</td>
<td>3493.70</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.4500</td>
<td>99.90</td>
<td>155.38</td>
<td>1303.48</td>
<td>8990.11</td>
<td>3492.74</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.5000</td>
<td>99.88</td>
<td>155.16</td>
<td>1304.16</td>
<td>8988.73</td>
<td>3491.86</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.5500</td>
<td>99.86</td>
<td>154.92</td>
<td>1304.92</td>
<td>8987.11</td>
<td>3490.83</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.6000</td>
<td>99.84</td>
<td>154.67</td>
<td>1305.71</td>
<td>8985.38</td>
<td>3489.72</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.6500</td>
<td>99.81</td>
<td>154.42</td>
<td>1306.52</td>
<td>8983.45</td>
<td>3488.49</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.7000</td>
<td>99.78</td>
<td>154.17</td>
<td>1307.32</td>
<td>8981.53</td>
<td>3487.27</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.7500</td>
<td>99.75</td>
<td>153.92</td>
<td>1308.11</td>
<td>8979.61</td>
<td>3486.04</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.8000</td>
<td>99.72</td>
<td>153.66</td>
<td>1308.97</td>
<td>8977.41</td>
<td>3484.62</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.8500</td>
<td>99.69</td>
<td>153.41</td>
<td>1309.79</td>
<td>8975.30</td>
<td>3483.27</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.9000</td>
<td>99.66</td>
<td>153.14</td>
<td>1310.66</td>
<td>8973.02</td>
<td>3481.81</td>
</tr>
</tbody>
</table>

...
6 Solver

6.1 Introduction

The function of the NPSS Solver is to drive a model created with the NPSS software to a self-consistent, or converged state. Models of complex systems such as gas turbine engines generally consist of a number of interdependent equations that typically cannot be solved explicitly. An iterative approach must be taken in which an initial guess at the solution is progressively refined until a satisfactory solution is obtained. This is the job of the solver.

As discussed in Section 4.8, it is often possible for a user to build and execute an NPSS model without any detailed knowledge of the solver or its operation. Some users, however, will need to customize the solver configuration for their model, and the NPSS solver was designed to provide a high degree of flexibility and user control. The following sections provide details for those users who wish to exert more control over the solver setup and its operating parameters. Complete listings of solver attributes and functions is given in Chapter 16.

The solver provides a solution to the model for each steady-state case or transient simulation time-step. Multi-point solutions, such as transient simulations, are controlled by other objects external to the solver. The TransientExecutive object that controls transient simulations is discussed in Chapter 6.

6.2 Basic Theory of Solver Operation

A set of equations can be considered to consist of a set of independent variables, the values of which can be set however desired, and a set of dependent conditions, the state of which is completely determined by the values of the independent variables. For example, consider the following single equation:

\[ 4x^2 - 7.25 = x^3 + \sin(x/4\pi) \]

In this case, there is one independent variable named \( x \). For any given value of \( x \), the value of the equation's left-hand side is determined \( 4x^2 - 7.25 \), as is the value of the equation's right-hand side \( x^3 + \sin(x/4\pi) \). The dependent condition is that the left-hand side equal the right-hand side. It is customary to express this condition as an error term whose value is zero when the condition is satisfied. For example, if the equation left-hand side is denoted by \( \text{eq}_\text{lhs} \) and the right-hand side by \( \text{eq}_\text{rhs} \), an error term could be defined as follows:

\[ \text{error} = \text{eq}_\text{lhs} - \text{eq}_\text{rhs} \]

Some tolerance would be specified, meaning that the equation is considered satisfactorily solved when the error term is less than the tolerance value. In the example above, the tolerance specified would be called an absolute tolerance since it is compared to the absolute difference between \( \text{eq}_\text{lhs} \) and \( \text{eq}_\text{rhs} \).

Absolute tolerances are often undesirable. If the equation were solved by its left- and right-hand sides both equalling 1, a reasonable absolute tolerance might be \( 10^{-5} \). If, however, the equation were solved by its left- and right-hand sides both equalling 10000, an absolute tolerance of \( 10^{-5} \) might be unattainable with the precision of the computer being used, and in most cases would be unnecessarily tight. A tolerance of 1 might be more appropriate. Often what is desired is a tolerance expressed as a certain fraction of the converged left- and right-hand sides. This gives rise to the fractional, or nondimensional, error term and tolerance:

\[ \text{errorNonDim} = \left( \frac{\text{eq}_\text{lhs} - \text{eq}_\text{rhs}}{\text{eq}_\text{rhs}} \right) \]

If error terms are defined this way, a single fractional tolerance can be appropriate for all equations, regardless of the magnitude of the converged left- and right-hand sides.

The error term definition given above is suitable for equations in which \( \text{eq}_\text{rhs} \) never equals zero. For cases in which \( \text{eq}_\text{rhs} \) may equal or approach zero, a different term should be used in the denominator of the error term.
definition. The term in the denominator can be called the \textit{reference term}. The error term definition can be generalized to take the following form:

\[
\text{errorNonDim} = \frac{\text{eq}_\text{lhs} - \text{eq}_\text{rhs}}{\text{eq}_\text{Ref}}
\]

If the term \text{eq}_\text{Ref} equals 1, the error term reduces to an absolute difference. If \text{eq}_\text{Ref} equals \text{eq}_\text{rhs}, the error term is a true fractional error as discussed above. In general, \text{eq}_\text{Ref} can be any expression that is always nonzero. Best results will be had if \text{eq}_\text{Ref} is of the same order of magnitude as the converged values of \text{eq}_\text{lhs} and \text{eq}_\text{rhs}.

If \text{errorNonDim} is defined as a fractional error with \text{eq}_\text{rhs} as the reference term, the variation of the error term with the independent variable is as follows:

Suppose an initial guess of \(x = 1\) is made. The error term is about 4.0. \textit{Newton's method} of iteration calculates the slope of the curve at this point, and finds the value of \(x\) at which the error term would equal zero if the curve were linear (i.e., if the slope calculated at the initial \(x\) value were the same at all \(x\) values). The true error term is calculated again at the new \(x\) value. The process is illustrated as follows:

Notice that if the initial guess for \(x\) had been 2.4, the next guess on \(x\) could have been a very large negative number--far from the solution. For this reason some form of \textit{damping} is normally applied to Newton's method. For example, the value of \(x\) might be allowed to only change by a specified maximum fraction on each iteration.

For the sorts of general equations arising from modeling systems like gas turbine engines, it is not possible to analytically determine the slope at a guessed value of an independent variable. The slope is approximated.
numerically, therefore, by finding the change in the error term due to a small perturbation of the independent variable. In the example above, a value of the error term would be found at a value of $x$ near the initial guess. For example:

$$\text{slope} \approx \frac{\text{errorNonDim}(1.0) - \text{errorNonDim}(0.99)}{1.0 - 0.99}$$

Once a new value of $x$ has been determined as discussed above, the procedure can be repeated to find another value of $x$ until the error term is less than the specified tolerance. A true Newton's method iteration recalculates the slope of the curve at every new value of $x$ as illustrated below.

This procedure converges in relatively few iterations, but each iteration can be time-consuming since the independent variable must be perturbed, the resulting value of the error term found, and the numerical slope calculated on every iteration. Modified Newton's method simply reuses the slope found for the initial guess of the independent variable, as follows:

Modified Newton's method makes each iteration fast because it avoids the overhead of recalculating slopes, but it can require many iterations to converge since the slope is never updated. Quasi-Newton methods use information from each iteration to update the slope without taking the time to calculate it exactly. These methods are usually the most successful: fewer iterations are required than with modified Newton, and each iteration is faster than with pure Newton.

The methods illustrated above for a single equation can be generalized to a system of any number of equations. They are often called Newton-Raphson methods in such cases. The methods can be applied when a set of $n$ equations contains exactly $n$ independent variables and $n$ dependent conditions, which are expressed as $n$ error
terms. Note that the solver must be able to vary each of the independent variables; their values must not be controlled by the model itself. There is no longer a single slope value, but rather a matrix of slopes quantifying the change in each error term with respect to each independent variable perturbed individually. This matrix is called the Jacobian matrix. It is a matrix of partial derivatives. If each partial derivative is approximated with a one-sided perturbation from the initial guess (rather than a centered difference, for example, which would require two perturbations from the initial guess), then it requires \( n \) evaluations of each equation to determine the Jacobian matrix. This makes quasi-Newton methods especially attractive over a pure Newton method for systems of several equations. A popular method for updating the Jacobian matrix on each iteration of a quasi-Newton method is Broyden's method. Generally, some criterion is applied to determine when another Jacobian update is appropriate, and when a completely new Jacobian must be formed by independent variable perturbations. Some provision is also made to damp each iteration. More details on the normal solver algorithm are given in Section 6.7.

The following, then, are the primary areas that concern a user of a quasi-Newton solver:

- Setting up the correct set of equations defined by an equal number of independent variables and dependent conditions.
- Specifying appropriate error terms and tolerances for each dependent condition.
- Controlling the way the Jacobian matrix is formed.
- Controlling the way the Jacobian matrix is updated.
- Controlling the damping applied to each iteration.
- Controlling the conditions under which the iteration is terminated if a solution has not been found.

The most common user-controlled parameters to address these concerns are discussed in this chapter. Full details on all parameters under the user's control are given in Chapter 16.

This chapter also discusses placing constraints on the solution to a system of equations (see Section 6.8). The basic equations of the system are equalities, in which the left-hand side of each equation must equal its right-hand side. Constraints are inequalities. They express mathematically an intention such as the following: "Find an engine operating condition as specified by these equations, but do not let burner exit temperature exceed 2200º F."

Other advanced uses of the NPSS solver, such as using discrete state variables, are discussed in Chapter 16.

### 6.3 General Structure of the NPSS Solver

The NPSS solver consists of several complementary object classes designed to achieve a solution to any given system model. The Solver class provides central control for all independent and dependent parameters assigned to it in the model. Each Solver object contains a single Jacobian object, which handles most of the mathematical operations applied to the matrix. The Solver object also contains lists of pointers to the Independent and Dependent objects distributed throughout the model. These objects perform many of the low-level calculations required by the solver, and act as the interfaces between the solver, the Jacobian and the model.

Data and methods in the solver are distributed among the various classes described above. The Solver object holds user-settable attributes that apply to all independents and dependents collectively (such as default values), and to the central control of the solver's operation. The Independent and Dependent objects hold attributes that apply to each of these objects separately, including override values to the defaults specified in the Solver object. For example, the maximum allowed number of solver iterations is set in the Solver object, as is the default perturbation size for all independents. A different value of perturbation size for a specific independent variable would be set in the appropriate Independent object, not in the Solver object.

### 6.4 Creating a Solver

The commands used to create a solver in an assembly are discussed and illustrated in Section 4.8.5. The top-level assembly has a Solver object named solver created automatically for the user. Each Solver object in the model is distinct, and can have its own set of attribute values that differ from those in other Solver objects. When a Solver object is created, reasonable values for all its attributes are set by default. The user need not assign values to any attributes unless there is reason to do so. Solver attributes that are commonly given values other than their
defaults are discussed in Section 6.7. A solver object must exist before Independent and Dependent objects can be added to it.

### 6.5 Independent Variables

The creation of Independent objects is discussed and illustrated in Section 4.8.3. Independent objects must both be created and added to a specific solver to be actively controlled. Element and Subelement objects automatically create Independent and Dependent objects likely to be needed to satisfy the physics of the model. The UserGuide Reference Sheets indicate which solver-related objects are created. As noted in Section 4.8.1, global command autoSolverSetup() normally adds all the Independent and Dependent objects to the appropriate solvers to satisfy the basic physics of the model. Section 4.8.2 shows how to verify this. Section 4.8.3 shows how additional Independent and Dependent objects can be created and added to a solver to satisfy special requirements. Independent objects can be created at the assembly level, or as children of Element and Subelement objects. As illustrated in Section 4.8.2, Independent objects created automatically by standard NPSS elements and subelements are created as children of those objects.

The user can give new values to the attributes contained in Independent objects, including those automatically created by the system. As discussed in Section 4.8.3, the only attribute the user must set is varName, which identifies the model variable that the Independent object puts under control of a solver. The following additional Independent input attributes are among those most often set by users.

**Table 25. Commonly Used Independent Input Attributes**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoSetup</td>
<td>Boolean</td>
<td>Determines whether or not the Independent should be automatically added to the solver when autoSolverSetup() is called.</td>
<td>FALSE</td>
</tr>
<tr>
<td>dxLimit</td>
<td>real</td>
<td>Specifies the maximum allowable change for the independent variable on any single iteration. Whether the maximum change specified is fractional or absolute is determined by dxLimitType. If not set, the solver default value is used.</td>
<td>solver.defaultDxLimit</td>
</tr>
<tr>
<td>dxLimitType</td>
<td>Option</td>
<td>Allowable values are &quot;FRACTIONAL&quot; and &quot;ABSOLUTE&quot;. If not set, the solver default value is used.</td>
<td>solver.defaultDxLimitType</td>
</tr>
<tr>
<td>indepRef</td>
<td>string</td>
<td>Contains an expression (that can be simply a numeric value or a variable name) evaluated to yield a reference value for the Independent. The result of the expression is stored in attribute xRef. If not set, the value of the independent variable x is used unless the value of x is zero, then xRef is set to 1.0. When specified, indepRef should evaluate to a value of the same order of magnitude as the converged value of the independent.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>perturbation</td>
<td>real</td>
<td>Specifies the amount by which the independent variable is varied when forming a Jacobian. Whether the perturbation specified is fractional or absolute is determined by perturbationType. If not set, the solver default value is used.</td>
<td>solver.defaultPerturbation</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Type</td>
<td>Description</td>
<td>Default Value</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>perturbationType</td>
<td>Option</td>
<td>Allowable values are &quot;FRACTIONAL&quot; and &quot;ABSOLUTE&quot;. If not set, the solver default value is used. That value defaults to &quot;FRACTIONAL&quot;.</td>
<td>solver.defaultPerturbationType</td>
</tr>
<tr>
<td>perturbationTrip</td>
<td>string</td>
<td>Evaluates to a number which is a perturbation tripping point for independent variable ‘x’ such that: If x&lt;=Tripping point then perturbation is positive, it x&gt;Tripping point then the perturbation temporarily changes sign.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>varName</td>
<td>string</td>
<td>The name of the independent variable. Variable scoping rules apply (see Section 2.2.4.3). Thus, if the Independent object resides within the object containing the independent variable, simply the variable's name can be specified. If the Independent object resides in global scope, the full path name of the independent variable must be specified.</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

With reference to the concepts discussed in Section 6.2:

- The formation of the Jacobian is affected by attributes `indepRef`, `perturbation`, and `perturbationType`.
- The damping is affected by attributes `indepRef`, `dxLimit`, and `dxLimitType`.

Notice that, if desired, each independent variable in the system can have a different perturbation and damping, although by default all of them share common values set in the Solver object.

By default, *fractional* perturbations are applied to each independent variable to form the Jacobian, and damping is achieved by only allowing independent variables to change a certain *fractional* amount per iteration. These fractions are fractions of each independent variable's *reference value*, determined by the expression in each Independent object's `indepRef` attribute. If an Independent object's `indepRef` expression yields a value much larger than the independent variable value, the quality of the partial derivatives involving that variable in the Jacobian will be poor because the perturbation of the variable will be large. In addition, little damping will be applied to changes of the variable. Conversely, if the `indepRef` expression yields a value much smaller than the independent variable value, so much damping may be applied that a solution cannot be had in a reasonable number of iterations.

If `indepRef` is left blank (its default setting), the independent variable's reference value becomes the value of the independent variable unless the independent variable value is zero, in which case a value of 1.0 is used.

As indicated in Table 25, some Independent object attributes (such as `dxLimit` and `perturbation`) have default values set by attributes in the Solver object. When the value of such an attribute is explicitly set by the user in the Independent object, that value is *locked*. This means that any subsequent change to the default value in the Solver object will not change the value explicitly set in the Independent object. Chapter 16 explains how the value of an Independent object attribute can be unlocked so that it reverts to the default value set in the Solver object.

Other Independent object attributes are described in Chapter 16. These options allow such actions as:

- Setting minimum and maximum limits on an independent variable so that either a warning or an error is generated if the variable is outside its limits upon solver termination.
- Establishing a mapping function between the independent variable value and the value used by the solver.
6.6 Dependent Conditions

The creation of Dependent objects is discussed and illustrated in Section 4.8.3. As with Independent objects, Dependent objects must both be created and added to a specific solver to be actively controlled. Element and Subelement objects automatically create Dependent (as well as Independent) objects likely to be needed to satisfy the physics of the model. See the NPSS Reference Sheets to find which solver-related objects are created by each Element and Subelement object. As noted in Section 4.8.1, global command autoSolverSetup() normally adds all the Independent and Dependent objects to the appropriate solvers to satisfy the basic physics of the model, as can be verified using a procedure like that illustrated in Section 4.8.2. Section 4.8.3 shows how additional Independent and Dependent objects can be created and added to a solver to satisfy special requirements. Dependent objects can be created at the assembly level, or as children of Element and Subelement objects. As illustrated in Section 4.8.2, Dependent objects created automatically by standard NPSS elements and subelements are created as children of those objects.

The user can give new values to the attributes contained in Dependent objects, including those automatically created by the system. As noted in Section 4.8.3, the only attributes the user must set are eq_lhs and eq_rhs. From these the solver forms an error term, the value of which must be less than a specified tolerance at convergence (see Section 6.2). For eq_lhs, eq_rhs and eq_Ref any valid expression that evaluates to a single real value can be given. The expression can be a simple numerical value, a variable name, an algebraic expression referencing multiple model variables and including any supported math operator, or can even be a call to a function which returns a real number. The following additional Dependent input attributes are among those most often set by users.

### Table 26. Commonly Used Dependent Input Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoSetup</td>
<td>Boolean</td>
<td>Determines whether or not the Dependent should be automatically added to the solver when autoSolverSetup() is called.</td>
<td>FALSE</td>
</tr>
<tr>
<td>eq_lhs</td>
<td>string</td>
<td>Any valid expression that when evaluated yields a real value for the left-hand side of the dependent condition. The results of the expression evaluation are stored in y1. The expression can basically be anything that yields a real number when evaluated. This includes simple constants or single variable references. It can also be any mathematical expression referencing one or more variables. The expression can even be a function call that performs any arbitrarily complex calculations and returns a single real value.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>eq_rhs</td>
<td>string</td>
<td>Any valid expression that when evaluated yields a real value for the right hand side of the dependent condition. The result of the expression evaluation is stored in attribute y2.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Type</td>
<td>Description</td>
<td>Default Value</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>eq_Ref</td>
<td>string</td>
<td>Any valid expression that when evaluated yields a real number reference value. The absolute value of the results of the expression evaluation is stored in attribute yRef. The error term, stored as attribute errorCon, is calculated as ((y1-y2)/yRef), and is used to check convergence. The value of yRef at the point where a Jacobian is generated is stored in attribute yRefLock and is used to calculate the error term errorIter. When an expression is not given for eq_Ref (or it evaluates to zero), the value of eq_rhs (y2) is used as the reference value. If eq_Ref is not set and y2 is equal to zero, then a yRef value of 1 is used.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>tolerance</td>
<td>real</td>
<td>Compared with the error term each iteration to determine convergence. Whether the tolerance specified is interpreted as fractional or absolute is determined by toleranceType. If not set, the solver default value is used.</td>
<td>solver. DefaultTolerance</td>
</tr>
<tr>
<td>toleranceType</td>
<td>Option</td>
<td>Allowable values are &quot;FRACTIONAL&quot; and &quot;ABSOLUTE&quot;. If not set, the solver default value is used. That value defaults to &quot;FRACTIONAL&quot;. When toleranceType = &quot;FRACTIONAL&quot;, the value stored in attribute errorCon is compared to tolerance to determine convergence. When toleranceType = &quot;ABSOLUTE&quot;, the given absolute tolerance value is divided by yRef to yield a fractional tolerance for internal use. This internal tolerance is recalculated each iteration so that the convergence is to the given absolute tolerance value.</td>
<td>solver. DefaultToleranceType</td>
</tr>
</tbody>
</table>

With reference to the concepts discussed in Section 6.2:

- The formation of the error term is affected by attributes eq_lhs, eq_rhs, eq_Ref
- The convergence tolerance is affected by attributes tolerance and toleranceType

Notice that, if desired, each dependent condition in the system can have a different tolerance value and type, although by default all of them share common values set in the Solver object.

Two fractional error terms, errorCon and errorIter, are calculated on each pass. The difference between the two error terms is that errorCon is normalized by yRef which is recalculated from eq_Ref each iteration and errorIter is normalized by yRefLock, which is the value of yRef when a new Jacobian matrix is generated and remains fixed until a new matrix is generated. The errorCon value is used for convergence checking and constraint handling. The errorIter value is used when generating a Jacobian and when using the solution of the Jacobian to calculate the updates to the independents parameters. While it is desirable to use fractional error in the Jacobian matrix, it is important mathematically that the reference term remain constant while a given matrix is generated and used. If eq_Ref evaluates to the same value during the iteration process, then errorCon and errorIter will remain at the same value throughout the iteration process. If not, the two errors will start at the same value and then differ as the value of yRef changes. This represents expected behavior and is not an error.

If eq_Ref is left blank (its default setting), the dependent condition's reference value becomes the value of the right-hand side of the condition (as determined by eq_rhs and stored in y2). If the right-hand side of a dependent evaluates to zero, and eq_Ref is not set, yRef is set to 1, making the error term equal the absolute difference between left- and right-hand sides. A good reference value is important to robust and stable convergence. If the eq_rhs expression is a constant or evaluates to values that will always represent a good reference, then the
eq_Ref expression can be left blank. If however eq_rhs is zero or tends towards zero, then it makes a poor reference and an expression yielding a good reference value should be given in eq_Ref.

As indicated, some Dependent object attributes (such as tolerance) have default values set by attributes in the Solver object. When the value of such an attribute is explicitly set by the user in the Dependent object, that value is locked. This means that any subsequent change to the default value in the Solver object will not change the value explicitly set in the Dependent object. Chapter 16 explains how the value of a Dependent object attribute can be unlocked so that it reverts to the default value set in the Solver object.

The following table summarizes the Dependent object output attributes.
**Table 27. Commonly Used Dependent Output Attributes**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorCon</td>
<td>real</td>
<td>Normalized or “non-dimensional” difference between the left- and right-hand sides of the dependent condition divided by (y_{Ref}). (\text{errorCon} = \frac{(y_1 - y_2)}{y_{Ref}})</td>
</tr>
<tr>
<td>errorIter</td>
<td>real</td>
<td>Normalized or &quot;non-dimensional&quot; difference between the left- and right-hand sides of the dependent condition divided by the locked reference value. (\text{ErrorIter} = \frac{(y_1 - y_2)}{y_{RefLock}})</td>
</tr>
<tr>
<td>y1</td>
<td>real</td>
<td>Left-hand side of the dependent condition. It is the value of expression eq_lhs.</td>
</tr>
<tr>
<td>y2</td>
<td>real</td>
<td>Right-hand side of the dependent condition. It is the value of expression eq_rhs.</td>
</tr>
<tr>
<td>yRef</td>
<td>real</td>
<td>Reference value of the error term. It is the absolute value of expression eq_Ref, if defined and non-zero, or otherwise the current value of expression y2. If y2 is also zero, yRef = 1. yRef is updated every pass.</td>
</tr>
<tr>
<td>yRefLock</td>
<td>real</td>
<td>The value of yRef at the point the current Jacobian matrix was generated. When a new matrix is generated, this attribute is updated</td>
</tr>
</tbody>
</table>

Other Dependent object attributes are described in Chapter 16. Most of these options have to do with the use of constraints which are discussed in Section 6.8.

### 6.7 Solver Setup

As discussed in Section 4.8.1, function `autoSolverSetup()` searches the entire model for Independent and Dependent objects that have the attribute `autoSetup` set to TRUE, and adds them to a solver automatically. Function `autoSolverSetup()` was discussed earlier as a global function, but it actually is a member function of every assembly. It also accepts a single Boolean integer argument that defaults to TRUE. When the argument is omitted, or explicitly set to TRUE, `autoSolverSetup()` adds objects with `autoSetup = TRUE` to the solver at its current level in the model, then recursively calls `autoSolverSetup()` in every assembly below that level. If the argument is explicitly set to FALSE as follows:

```cpp
autoSolverSetup( FALSE );
```

then `autoSolverSetup()` is called on the current assembly only. In either case, `autoSolverSetup()` first removes all active Independent and Dependent objects from the solver before adding those marked by `autoSetup = TRUE`. When the argument is explicitly set to FALSE, a user-created Independent or Dependent object with `autoSetup = TRUE` is added to a solver by `autoSolverSetup()` just as a system-created object is. Likewise, a system-created Independent or Dependent object can be omitted from the automatic solver setup by explicitly setting its `autoSetup` attribute to FALSE. The names of Independent and Dependent objects created by standard NPSS elements and subelements are given in the NPSS Reference Sheets. They can be found for an existing model using the procedure given in Section 4.8.2.

Before discussing commonly used solver input attributes, a few more details about normal solver operation are appropriate. The solver recognizes three kinds of passes, or single executions of the model. The first pass evaluates the model state before any steps are taken to converge the model. A perturbation pass is a pass in which one independent variable has been perturbed from its initial state to determine the resulting effect on the model’s error terms. As many perturbation passes as there are independent variables are made to build the Jacobian matrix. An iteration pass is made to determine a new model state after the solver has determined values for all the independent variables that should be closer to convergence. After each iteration pass, error terms for each dependent condition are computed. Each Dependent object’s convergent error term is compared to that object’s tolerance. If all the active Dependent objects are converged, the solver terminates. Otherwise, the convergence ratio is examined. The convergence ratio is a measure of how much progress the iteration just completed has made toward convergence. The convergence ratio is the ratio of the square root of the sum of the squares of the
errorCon values for the active dependents from the current iteration to the same quantity for the same set of dependents on the previous iteration.

A convergence ratio greater than 1.0 suggests the model is diverging. If it is greater than divergenceLimit (described in the table below), a new Jacobian is formed immediately. An exception is made if the iteration is very close to convergence as measured by comparing the square root of the sum of the squares of the convergence errors to the square root of the sum of the squares of the tolerance. If the errors are within a given multiple of the tolerances (given by divergenceCutoffMultiplier), then the divergenceLimit is ignored. This avoids generating a new Jacobian in a region where even very slight reversals in the progressive diminution of the errors can result in a comparatively large fractional increase in the total magnitude of the errors.

A convergence ratio less than 1.0 suggests the model is converging, and the smaller the value, the faster the convergence. If the convergence ratio is greater than convergenceLimit (i.e. the most recent iteration was not very successful in reducing the model error), a counter is indexed. If a certain number of consecutive iterations (defined by maxConvergenceFailures) fail to produce a convergence ratio less than convergenceLimit, a new Jacobian is formed. If the convergence ratio is greater than broydenLimit, the Jacobian is updated using the Broyden method. If a certain number of Jacobians (maxJacobi) have been formed, but convergence has not been achieved, the solver terminates. Also, if a certain total number of iterations (maxIterations) have been performed but convergence has not been achieved, the solver terminates.

Some of the Solver object input attributes that are commonly customized are given in the following table.

Table 28. Commonly Used Solver Input Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>broydenLimit</td>
<td>real</td>
<td>Compared to convergenceRatio each iteration to determine if a Broyden update should be performed on the Jacobian matrix. If the calculated value of convergenceRatio is smaller than the value of this variable, the speed of convergence is judged to be sufficiently rapid enough to forgo a Broyden update.</td>
<td>0.0</td>
</tr>
<tr>
<td>convergenceLimit</td>
<td>real</td>
<td>Compared with convergenceRatio each iteration to determine if the system is converging rapidly enough to be considered successful. See maxConvergeFailures below. Nominal range is between zero and one.</td>
<td>1.0</td>
</tr>
<tr>
<td>divergenceLimit</td>
<td>real</td>
<td>Compared with convergenceRatio each iteration to determine if the system is diverging. If convergenceRatio is larger than this variable, a new Jacobian matrix is generated immediately. Nominal range is greater than one.</td>
<td>2.0</td>
</tr>
<tr>
<td>debugLevel</td>
<td>Option variable</td>
<td>Defines the amount of diagnostic output generated. (See Section 16.7 for details).</td>
<td>&quot;NONE&quot;</td>
</tr>
<tr>
<td>defaultDxLimit</td>
<td>real</td>
<td>Maximum allowable change for independent variables on any single iteration. Whether the maximum change specified is fractional or absolute is determined by defaultDxLimitType.</td>
<td>0.10</td>
</tr>
<tr>
<td>defaultDxLimitType</td>
<td>Option variable</td>
<td>Allowable values are &quot;FRACTIONAL&quot; and &quot;ABSOLUTE&quot;.</td>
<td>&quot;FRACTIONAL&quot;</td>
</tr>
</tbody>
</table>
### Attribute Name | Type | Description | Default value
--- | --- | --- | ---
**defaultPerturbation** | real | Amount by which independent variables are varied when forming a Jacobian. Whether the perturbation specified is fractional or absolute is determined by defaultPerturbationType. | .001
**defaultPerturbationType** | Option variable | Allowable values are "FRACTIONAL" and "ABSOLUTE". | "FRACTIONAL"
**defaultTolerance** | real | Compared with dependent condition error terms each iteration to determine convergence. Whether the tolerance specified is fractional or absolute is determined by defaultToleranceType. | 0.0001
**defaultToleranceType** | Option variable | Allowable values are "FRACTIONAL" and "ABSOLUTE". | "FRACTIONAL"
**maxConvergeFailures** | int | Number of consecutive times that convergenceRatio may exceed convergenceLimit before an new Jacobian matrix is generated. | 3
**maxIterations** | int | Number of iterations that may be attempted to converge any single point. This does not include model passes required to generate the Jacobian. | 50
**maxJacobians** | int | Number of Jacobians that may be generated from scratch for any single point. | 10
**solutionMode** | Option variable | Solver mode. A value of "STEADY_STATE" attempts to converge a steady-state point. A value of "TRANSIENT" attempts to converge each time step of a transient run (see Chapter 6). A value of "ONE_PASS" cause a single model pass with no iteration. | "STEADY_STATE"

With reference to the concepts discussed in Section 6.2:

- **defaultTolerance** and **defaultToleranceType** affect the error terms and their tolerances
- **defaultPerturbation** and **defaultPerturbationType** affect how the Jacobian is formed
- **broydenLimit**, **convergenceLimit**, **divergenceLimit**, **maxBroydens**, and **maxConvergenceFailures** affect how the Jacobian is updated and when a new Jacobian is generated
- **defaultDxLimit** and **defaultDxLimitType** affect the damping of each iteration
- **maxIterations** and **maxJacobians** affect when the solver terminates.

It should be noted that **defaultDxLimitType**, **defaultPerturbationType** and **defaultToleranceType** have been included as "commonly used solver input attributes" for completeness in understanding other attributes. It is strongly recommended that the user leave these default types as "FRACTIONAL" unless there is clear reason to do otherwise. These types can be altered for individual **Independent** and **Dependent** objects when required as described in sections 6.5 and 6.6.

The following table lists some **Solver** object output attributes that are often of interest in examining a Solver's performance or including in **DataViewer** output.
Table 29. Commonly Used Solver Output Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>converged</td>
<td>Boolean</td>
<td>Indicates if the model is converged. If the solver terminates and converged = 0, something went wrong.</td>
</tr>
<tr>
<td>convergenceRatio</td>
<td>real</td>
<td>Current model error divided by the model error of the previous pass. See the detailed discussion earlier in this section.</td>
</tr>
<tr>
<td>determinant</td>
<td>real</td>
<td>Determinant of the current Jacobian matrix.</td>
</tr>
<tr>
<td>errorConverged</td>
<td>int array</td>
<td>Each entry is a Boolean integer (0 = false, 1 = true) indicating the convergence state of each dependent condition. The order of entries is the same as given by solver.list( &quot;Dependent&quot;, 0 ).</td>
</tr>
<tr>
<td>matrixSize</td>
<td>int</td>
<td>Number of rows and columns in the Jacobian matrix, which equals the number of independent variables, and the number of dependent conditions.</td>
</tr>
<tr>
<td>iterationCounter</td>
<td>int</td>
<td>Number of iterations taken to attempt convergence. This number does not include Jacobian formation passes.</td>
</tr>
<tr>
<td>passType</td>
<td>string</td>
<td>Type of pass being taken: &quot;noPass,&quot; &quot;firstPass&quot;, &quot;perturbationPass&quot;, or &quot;iterationPass.&quot;</td>
</tr>
<tr>
<td>numBroydens</td>
<td>int</td>
<td>Total number of Broyden updates performed during the convergence process.</td>
</tr>
<tr>
<td>numJacobians</td>
<td>int</td>
<td>Total number of Jacobian matrices formed from scratch during the convergence process.</td>
</tr>
<tr>
<td>passCounter</td>
<td>int</td>
<td>Total number of model execution passes performed during the convergence process. This includes the initial pass and all Jacobian formation passes as well as all iteration passes.</td>
</tr>
<tr>
<td>perturbationCounter</td>
<td>int</td>
<td>Number of the perturbation pass during the generation of a Jacobian. Starts at one. The value is zero when the pass type is anything other than “perturbationPass”.</td>
</tr>
<tr>
<td>lastPerturbationPass</td>
<td>Boolean</td>
<td>A flag to indicate that the pass is the perturbation pass for the last Independent. This is important because at the end of this pass while the pass type is still “perturbationPass”, the new Jacobian will be used to calculate a new set of updates to the Independents. This set of updates will be missed if output of Independent updates is only done when the pass type is “iterationPass”</td>
</tr>
</tbody>
</table>

Additional Solver object attributes, intended for advanced users, are discussed in Section 16.1.4.

6.8 Constraint Handling

6.8.1 Constraint Definition and Usage

As mentioned in Section 6.2, the NPSS solver has the ability to solve constrained problems. These are problems in which the user specifies that one or more dependent conditions are to be satisfied only if certain limits are not violated. For example, one might desire an engine's thrust to equal some value, but only if the burner exit temperature is less than some limit. If the desired thrust cannot be obtained, the limiting temperature is to be achieved instead. The basic dependent conditions of a model are equalities. Constraints are inequalities: they specify the maximum or minimum value some variable or expression can have.

To constrain a solution, the user must accomplish three tasks:

- Define the constraints.
- Associate each constraint with an active dependent condition.
- Notify the solver of the presence of the constraints.

Constraints are specified as Dependent objects, just as ordinary dependent conditions are (see Sections 4.8.1 and 6.6). It is the second step noted above, associating one Dependent object with another rather than adding it...
directly to the solver, that causes a Dependent object to be handled as a constraint (inequality) rather than as a dependent condition (equality). Any Dependent object can be either a dependent condition or a constraint, and can be switched from one to the other from case to case. A given Dependent object cannot be both a dependent condition and a constraint at the same time, however.

For clarity, dependents functioning as constraints will be referred to as constraint dependents, or simply constraints. A dependent with constraint dependents associated with it will be referred to as a target dependent.

Any target dependent can be constrained by one or more constraint dependents, but a given constraint can be used to constrain only one active target dependent at a time. Only the target dependents are added to the Solver. The Solver checks each Dependent as it is being added to determine if constraint Dependent have been added to it, in which case the Solver extracts the constraint information from the target Dependent and adds it to a list of auxiliary Dependents.

When, during convergence, a constraint is reached, the Solver moves the constraint Dependent to the active list of Dependents being converged (turning the constraint from an inequality condition to an equality condition) and moves the target Dependent to the inactive Dependent list. Since this is the expected behavior of the program, no warning or error messages are generated. However, the constraintsActive flag in the Solver is set to TRUE to indicate that one or more constraint Dependents are in the active Dependent set. The target Dependents that have been replaced by a constraint will contain the name of that constraint in its activeConstraintName attribute. This attribute will be set back to any empty string when the target is no longer constrained and is moved back onto the active Dependent list.

Each Dependent object has several special attributes that control its use of other Dependent objects as constraints. These were not discussed in Section 6.6, but are summarized in the table that follows.

### Table 30. Dependent Input Attributes Used to Define Constraints

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraintNameList</td>
<td>string array</td>
<td>The names of other Dependent objects that constrain this object. The named Dependent objects must not themselves be dependent conditions added directly to the solver.</td>
<td>None</td>
</tr>
<tr>
<td>limitTypes</td>
<td>string array</td>
<td>Specifies whether each Dependent object listed in constraintNameList represents a maximum limit (&quot;MAX&quot;) or a minimum limit (&quot;MIN&quot;). &quot;MAX&quot; and &quot;MIN&quot; are the only meaningful values for the entries of this array.</td>
<td><em>MAX</em></td>
</tr>
<tr>
<td>constraintPriorities</td>
<td>int array</td>
<td>Priority of each constraint listed in constraintNameList. Values should be positive integers. Negative values are allowed but not encouraged. The larger the value, the higher the priority. These priorities are used only when one or more maximum limits cannot be satisfied at the same time as one or more minimum limits. The constraint with the highest priority is satisfied to the exclusion of others it is in conflict with. More than one constraint can have the same priority. By default, all constraints have priority 1.</td>
<td>1</td>
</tr>
<tr>
<td>constraintSlopes</td>
<td>int array</td>
<td>Slope multipliers for each Dependent object listed in constraintNameList. It is intended that each value be either 1 or -1. By default, all values are 1. The user need not alter this default unless a solution instability is encountered.</td>
<td>1</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Type</td>
<td>Description</td>
<td>Default Value</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>resolveMinMaxConflict</td>
<td>Option variable</td>
<td>Allowable values are &quot;MAX&quot; and &quot;MIN&quot;. If two constraints (one a maximum limit and one a minimum limit) cannot be satisfied at the same time, and both have the same priority, the value of this attribute determines which is satisfied. If not set, the solver default setting is used, which is &quot;MAX&quot;.</td>
<td>solver. ResolveMinMax Conflict</td>
</tr>
<tr>
<td>useConstraints</td>
<td>Boolean integer</td>
<td>If this attribute is set to FALSE in a target dependent containing constraint dependents, the target will not activate any of its constraints even if they exceed their limits. If set to FALSE in a given constraint dependent, that dependent will not become an active dependent regardless of exceeding its limit value. Changing the value does not effect whether a dependent is present in the solver configuration. Rows in the Jacobian are present for all constraint objects regardless of the value of this flag.</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

A few attributes of Solver objects that affect constraints can be mentioned.

### Table 31. Solver Attributes Affecting Constraints

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraintHandlingOpt</td>
<td>Boolean integer</td>
<td>Determines whether constraints in any Dependent object will affect the solution.</td>
<td>TRUE</td>
</tr>
<tr>
<td>resolveMinMaxConflict</td>
<td>Option variable</td>
<td>Determines the default behavior when a maximum limit conflicts with a minimum limit. If set to &quot;MAX&quot;, the maximum limit is chosen; if set to &quot;MIN&quot;, the minimum limit is chosen. This default behavior can be overridden in individual constrained dependents.</td>
<td>&quot;MAX&quot;</td>
</tr>
<tr>
<td>constraintsActive</td>
<td>Boolean integer</td>
<td>Output indicating whether a constraint affected the final solution. 0 means the solution would have been the same without constraints; 1 means the solution was constrained.</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

### 6.8.2 Adding Constraints Dependents to Target Dependents

An individual constraint dependent can be associated with a target dependent in one of two ways: through the addConstraint member function, or by directly setting the internal storage arrays. Each method is discussed below.

For the examples in this section, assume that burner fuel/air ratio will be added to the solver as an independent variable. Assume that a Dependent object named RunCondition will also be added to the solver as a target dependent condition. RunCondition.eq_lhs is total net thrust, stored in variable PERF.Fn. The desired value of net thrust is stored in the variable PERF.FnTarget.
Dependent RunCondition {
    eq_lhs = "PERF.Fn";
    eq_rhs = "PERF.FnTarget";
}

Assume the variation of burner outlet temperature (BOT) and high-pressure compressor corrected speed (CmpH.Nc) with net thrust (Fn) is as follows.

![Graph with BOT vs. Fn]

Suppose you want to limit BOT to 2660ºR. If the burner object is named "Brn", then the parameter to be limited is "Brn.Fl_O.Tt". The limit value is specified by a model specific variable "Brn.TtOutMax". The following defines a Dependent object to be used as a constraint on the RunCondition target dependent:

Dependent BOTlimit {
    eq_lhs = "Brn.Fl_O.Tt";
    eq_rhs = "Brn.TtOutMax";
}

The addConstraint() method of the target is used to add a single constraint dependent at a time. This method takes four arguments: the name of the constraint dependent, the limit type (either "MAX" or "MIN"), the priority, and the slope. The only attribute that must be specified is the constraintDependent name. If the second, third and fourth arguments are not given, then default values of "MAX", 1 and 1 are used respectively. If the name of the
constraint Dependent doesn't make the limit-type clear, it is a good idea to specify the limit-type, even if it is the default value. In this example we might have called the constraint Dependent "BOTmaxLimit" and made the type clear in the name.

    RunCondition.addConstraint("BOTlimit", "MAX");

It does not matter if the target Dependent has been added to the Solver. If the target has already been added to the Solver, the additional constraint Dependent is automatically added to the Solver constraint list at the same time it is being added to the list of constraints held by the target. The same is true in reverse when a constraint is removed. If the target Dependent has not been added to a Solver, the constraint Dependent is simply added or removed from the targets list of constraints.

As shown in the first graph above, the example model requires a BOT greater than 2660 to achieve a net thrust of 6200, as requested by RunCondition (see above). The solution will therefore be limited to a BOT of 2660, producing a net thrust of only about 6142. Since this is expected behavior, no warning message is issued. However, any time a constraint has been reached, the solver.constraintsActive flag is set to TRUE to indicate that one or more constraints are active.

A given target dependent can be constrained by more than one constraint dependent. For example, suppose a second constraint, a limit on high-pressure compressor corrected speed, is defined. The high-pressure compressor in our example is an object named CmpH, and its corrected speed is given by attribute Nc. Further assume the model definition contains a user-specified variable "CmpH.NcMax", which is set to the desired maximum corrected speed.

    Dependent CmpHNcLimit {
        eq_lhs = "CmpH.Nc";
        eq_rhs = "CmpH.NcMax";
    }

The new limit on corrected speed is a maximum limit, as is the limit on BOT. To add this constraint to RunCondition, there is no need to specify a priority relative to BOTlimit because priority values are only used to choose between conflicting maximum and minimum limits. The slope multiplier should always be set to 1 unless there is reason to do otherwise. Therefore, the new constraint can be added to RunCondition as follows:

    RunCondition.addConstraint("CmpHNcLimit", "MAX");

The plots given earlier show that CmpH.Nc reaches 9800 at a lower net thrust than 6200, and a lower BOT than 2660. When multiple maximum or minimum limits constrain a dependent, the most severe prevails, regardless of its priority value. A new solution constrained as above by both BOTlimit and CmpHNcLimit will run to the CmpH corrected speed limit of 9800 (and a thrust of about 6078 (less than the target) at a BOT of about 2645 (less than the limit).

Function addConstraint() can be used as many times as desired to add constraints one at a time to dependent conditions.

It is possible, though not recommended, to add constraints by directly setting the arrays in the target Dependent used to store the constraint information. For example, the BOTlimit and CmpHNcLimit could be added to the RunCondition target as follows

    RunCondition {
        constraintNameList = { "BOTlimit", "CmpHNcLimit" };
        limitType = { "MAX", "MAX" };
        constraintPriorities = { 1, 1 };
        constraintSlopes = { 1, 1 };
    }

The constraintPriorities and constraintSlopes arrays will default to 1 for each constraint if it is not explicitly set. Thus the lines above that set these arrays can be omitted.
6.8.3 Constraint Groups

The second way to add constraint dependents to a target involves using constraint groups. Sometimes groups of constraints go together. For example, one set of limits might correspond to standard engine ratings, and another set correspond to a one-engine-inoperative (OEI) condition on a twin-engine aircraft. NPSS allows related groups of constraints to be added together.

First, as always, the constraints should be defined as Dependent objects.

```plaintext
Dependent BOTlimitStd {
    eq_lhs = "Brn.Fl_O.Tt";
    eq_rhs = "2660";
}
Dependent BOTlimitOEI {
    eq_lhs = "Brn.Fl_O.Tt";
    eq_rhs = "2675";
}
Dependent CmpHNcLimitStd {
    eq_lhs = "CmpH.Nc";
    eq_rhs = "9800";
}
Dependent CmpHNcLimitOEI {
    eq_lhs = "CmpH.Nc";
    eq_rhs = "9850";
}
Dependent LeanBlowOut {
    eq_lhs = "Brn.Wfuel";
    eq_rhs = "100./3600.";
}
```

Next, objects of type ConstraintGroup should be created. Constraints that are to be handled as a group should be added to the same ConstraintGroup object exactly as they would have been added directly to a Dependent object. For example:

```plaintext
ConstraintGroup StdLims;
StdLims.addConstraint( "BOTlimitStd", "MAX", 2);
StdLims.addConstraint( "CmpHNcLimitStd", "MAX", 2);
StdLims.addConstraint( "LeanBlowOut", "MIN", 1);

ConstraintGroup OEIlims;
OEIlims.addConstraint( "BOTlimitOEI", "MAX", 2);
OEIlims.addConstraint( "CmpHNcLimitOEI", "MAX", 2);
OEIlims.addConstraint( "LeanBlowOut", "MIN", 1);
```

In the example, notice that a given constraint (in this case LeanBlowOut) can be member of more than one constraint group. But the limitation that a given constraint can not be used to limit two target Dependents simultaneously still applies. Both minimum and maximum constraints are present within a group, so, to be safe, a higher priority is assigned to one set. In this case, however, there is little possibility of conflicting constraints.

Objects of type ConstraintGroup have the same attributes used to define constraints as objects of type Dependent do. That is, ConstraintGroup objects have attributes constraintNameList, limitTypes, constraintPriorities, and constraintSlopes. Dependent objects can be associated with a ConstraintGroup object using these attributes directly rather than command addConstraint() if desired.

Individual constraints can be removed from constraint groups in the same way as they would be from Dependent objects. The function removeConstraint() can be used (it is a member function of ConstraintGroup objects as it is of Dependent objects).

Function containsConstraint() is a member function of ConstraintGroup objects. It takes a single string argument that should be the name of a Dependent object. If the named dependent is a constraint in the
group on which containsConstraint() was called, the function returns 1 (or "TRUE"). If not, it returns 0 (or "FALSE"). For example,

```plaintext
StdLims.containsConstraint( "LeanBlowOut" )
```

returns 1 since Dependent object LeanBlowOut is part of ConstraintGroup StdLims.

The ConstraintGroup object can be added to the target Dependent using either the setConstraintGroup or addConstraintGroup functions. These functions differ in that the setConstraintGroup function clears out any previously added constraints from the target before adding the constraints from the group, while the addConstraintGroup function simply augments the constraints already present in the target. Any duplicates between the constraints already present and those in the group are quietly ignored by the addConstraintGroup function. The setConstraintGroup function is used in the example below to insure that the target contains only the constraints in the group.

```plaintext
RunCondition.setConstraintGroup( "StdLims" );
PERF.FnTarget = 6250;
solver.forceNewJacobian = TRUE;
CASE++;
run();
```

A net thrust of 6250 requires a fuel flow well above the lean blowout limit, so the minimum constraint does not come into play. The maximum limits on BOT and CmpH.Nc would be violated, however, so the more severe of the two is held: the solution yields an CmpH corrected speed of 9800.

Constraints can be added individually to a Dependent object in addition to those added as part of a group. Individual constraints must be added after the group when using the setConstraintGroup method, however. Any constraints added to the target Dependent prior to adding the constraint group will be removed when the constraint group is added to the target Dependent. The following lines add an additional constraint to those in the StdLims group added above.

```plaintext
Dependent CmpHNmechLim { eq_lhs = "CmpH.Sh_O.Nmech"; eq_rhs = "9320"; }
RunCondition.setConstraintGroup( "StdLims" );
RunCondition.addConstraint( "CmpHNmechLim", "MAX", 2, 1 );
```

### 6.8.4 Removing Constraints

Constraints can be removed one at a time from Dependent objects using function removeConstraint(). The function takes a single string argument: the name of the constraint to be removed. For example:

```plaintext
RunCondition.removeConstraint( "CmpHNcLimit" );
```

All the constraints can be removed from a target with the removeAllConstraints() command. For example:

```plaintext
RunCondition.removeAllConstraints();
```

The system had no function to remove all the constraints brought in by a constraint group because a constraint may have come in through more than one ConstraintGroup. It is difficult to know the user's intent. If one group is removed, should the duplicate constraint be removed, or should it remain behind because it is a part of another group that is not being removed? If all the constraints from a group need to be removed, it is probably simpler and less error prone to clear all the constraints and add back in the desired ConstraintGroup(s) and individual constraints.

### 6.8.5 Temporarily Disabling Constraints

An unconstrained solution can be found without disturbing the established constraints. All constraint handling in the solver can be temporarily disabled using the constraintHandlingOpt attribute:
When constraintHandlingOpt is FALSE, the constraints remain on the auxiliary list of dependents and the solver is converged to the target dependents regardless of whether any of the constraints are being violated. Unconstraining a single target Dependent can be done by setting the target Dependent object’s useConstraints attribute to FALSE:

```cpp
RunCondition.useConstraints = FALSE;
CASE++;
Brn.FAR *= 1.001;
run();
```

This run will converge to a net thrust of 6250 as specified by the basic RunCondition dependent condition. Other target Dependents that have useConstraints equal TRUE are unaffected and will converge to a constraint Dependent if required.

To reactivate the constraints set up earlier, simply set the useConstraints flag on the target back to TRUE. No constraints were lost by setting useConstraints to FALSE, nor was any existing Jacobian matrix affected, so the Jacobian does not need to be regenerated. Therefore,

```cpp
RunCondition.useConstraints = TRUE;
run();
```

will again be limited by a BOT of 2675.

An individual constraint associated with a target dependent can also be disabled by setting the useConstraints attribute in the individual constraint to FALSE. For example the LeanBlowOut constraint can be disabled for a single point.

```cpp
LeanBlowOut.useConstraints = FALSE;
CASE++;
Brn.FAR *=1.001;
run();
```

The point will now converge to one of the other constraints or the target dependent.

Setting constraintHandling Option in the solver or useConstraints in individual targets or constraint Dependents does not effect whether the constraint(s) are present in the Solver configuration. Printing out the content of Solver.constraintNames will still list all of the constraints and there will be rows for all constraints in the Jacobian matrix regardless of the value of useConstraints. It only effects whether the constraint(s) are adhered to in the converged solution.

### 6.8.6 Resolving Constraint Slope Conflicts

In certain rare cases, the constraints associated with a given dependent may have incompatible slopes (changes in their error values with respect to changes in the solver independents). The algorithm used to invoke constraints requires that the slopes have the same sign (+ or -). If this is not the case, the constraint selection will tend to toggle back and forth between different constraints until the solver exceeds its iteration limits. Automatically detecting this mode of convergence failure is not simple, and its diagnosis is left to the user. The problem can be corrected by reversing the sign of the slopes of the offending constraints. When the sign of the slope is reversed, the limit-type must also be reversed (from "MIN" to "MAX" or vice versa). If this problem arises, the user should call invertConstraint(), which is a member function of both Dependent objects and ConstraintGroup objects. The syntax is as follows.

```cpp
Dependent_name.invertConstraint( "constraint_name" );
```
or

```cpp
ConstraintGroup_name.invertConstraint( "constraint_name" );
```

For example:
RunCondition.invertConstraint( "BOTlimit" );
StdLims.invertConstraint( "LeanBlowOut" );

If a constraint that is part of a `ConstraintGroup` object is inverted before that group is associated with a dependent, then the inversion remains when the group is associated (i.e., when `setConstraintGroup()` or `addConstraintGroup()` is called). If a constraint that is part of a `ConstraintGroup` object is inverted after that group is associated with a dependent, then the inversion is lost if the constraint or the group is removed from the dependent.

Another option to eliminate nonconvergent constraint toggling is to disable the use of projected errors for selecting constraints. This can be done on a case by case basis if desired and is accomplished by setting `solver.maxConstraintProjections` to zero. More information is given in Chapter 16.

### 6.8.7 Resolving Min/Max Constraint Conflicts

A min/max conflict is a situation where a maximum and minimum type constraint can not both be satisfied at the same time. In other words, it is when the maximum limit value is less than the minimum limit value. In these situations it is impossible to find a value that doesn't violate both constraints. A choice has to be made as to which of the conflicting constraints is matched.

There are two ways of specifying which of the conflicting constraints is selected. The primary method is by selecting the constraint with the highest numerical value of priority. For example, recall that in the stdLims constraint group defined previously there are three constraints; BOTlimitStd, NPCNcLimitStd, and LeanBlowOut. The first two were defined as MAX type constraints and given a priority level of 2, while the third was defined as being a MIN type constraint and given a priority of 1. It is extremely unlikely that these particular constraints would ever get into a min/max conflict, but if they did, then the given priorities would mean that one of the max constraints (which ever one was most in violation) would be selected. The min constraint would be allowed to be in violation.

If the conflicting minimum and maximum constraint have the same priority level, then the secondary resolution method is to use the value of the `resolveMinMaxConflict` flag. The default value is stored in the Solver instance of this variable. This can be overridden by setting the value in the target dependent. Calling the `revertToDefaults()` function on the target dependent allows the target to again use the default value stored in the Solver.
7 Transient Simulation

7.1 Introduction

Chapter 6 discussed how the NPSS solver iteratively controls passes through the model until a system of equations defining the physics of the model and the requirements of the user is satisfied. The solutions in view were steady-state solutions in which time did not play a part. Transient simulations, in which the time-varying behavior of a system is sought over some time interval, are performed by finding a series of solutions at discrete time steps spanning the interval desired. The solution at each time step is similar to a steady-state solution: it is represented by a system of equations that can be solved iteratively by the methods of Chapter 6. The transient problem is unique, however, in that some of the equations to be solved represent integrations.

Suppose it is desired to find the transient response of a gas turbine engine to a ramp in fuel flow.

![Graph showing transient response](image)

The change in HP spool speed to this event is defined by a first-order differential equation:

\[ T_{\text{net}} = I \alpha = I \frac{dN}{dt} \]

\[ \frac{dN}{dt} = \frac{T_{\text{net}}}{I} \]

\[ t_2 \int_{t_1}^{t_2} dN = \int_{t_1}^{t_2} \frac{T_{\text{net}}}{I} dt \]

\[ N_{t_2} - N_{t_1} = \int_{t_1}^{t_2} \frac{T_{\text{net}}}{I} dt \]

\( T_{\text{net}} \) is the net torque on the shaft, \( I \) is the rotational inertia of the spool, \( N \) is the rotational speed of the spool, \( t \) is time, and the time step under consideration is from time \( t_1 \) to time \( t_2 \). In the following discussion we will refer to \( N \) as the state variable and \( \frac{dN}{dt} \) as the state derivative, which for the shaft we know is a function of net torque, which is in turn a function of shaft speed. Suppose that \( N \) and \( T_{\text{net}} \) are known at time \( t_1 \), and it is desired to know \( N \) at \( t_2 \). It is generally true that the equations arising in transient simulations can be expressed by first-order equations similar to the example above.
Calculation of the transient depends on the way the integral on the right-hand-side of the final equation above is evaluated. Some methods use only the information known at the previous time \((t1)\) to determine conditions at the current time \((t2)\). These are known as \textit{explicit} methods. The only explicit integration method that NPSS supports at this time is the Euler method. The Euler method simply assumes that the integrand is constant over the interval:

\[
N_{t_2} = N_{t_1} + \frac{dN}{dt}
\bigg|_{t_1} (t_2 - t_1)
\]

\[
N_{t_2} = N_{t_1} + \frac{T_{\text{net}}}{I}
\bigg|_{t_1} (t_2 - t_1)
\]

Since explicit methods, such as Euler, only use information at the previous time to determine values at the current time, no iteration is required to perform the integration. Steady-state iteration may be required at a given time to determine the required quantities (in this case the net torque), but once that iteration is complete, the time integration can be performed directly. This may result in a significant performance improvement.

Explicit methods, however, must make some sort of assumption regarding the behavior of the integrand over the time interval. As illustrated above, a poor assumption can lead to inaccurate results. The cumulative error resulting from the assumption above can be minimized by taking very small time steps. This is a typical requirement for successful use of explicit integration methods.

A much more accurate integration would be had if the trapezoidal rule were used to approximate the integral.

\[
N_{t_2} = N_{t_1} + \frac{1}{2}
\left(\frac{dN}{dt}
\bigg|_{t_1} + \frac{dN}{dt}
\bigg|_{t_2}\right)(t_2 - t_1)
\]

\[
N_{t_2} = N_{t_1} + \frac{1}{2}
\left(\frac{T_{\text{net}}}{I}
\bigg|_{t_1} + \frac{T_{\text{net}}}{I}
\bigg|_{t_2}\right)(t_2 - t_1)
\]
This more accurate integral requires the value of the state derivative \( \frac{dN}{dt} \bigg|_{t_2} \), which is not known purely from the information at time \( t_1 \). However, given a value of \( N_{t_2} \), the value \( \frac{dN}{dt} \bigg|_{t_2} \) can be calculated. Therefore, a prediction of \( N_{t_2} \) must be made. An iteration is required to compare the predicted value of \( N_{t_2} \) to the value calculated by the Trapazoidal integration equation above. The solver accomplishes this by including the predicted \( N_{t_2} \) as an independent term to be varied and \([N_{t_2} \text{ (calculated)} - N_{t_2} \text{ (predicted)}]\) as the dependent error to be solved.

This method is referred to in general as a “Predictor/Corrector” method, where the solver updates the predicted value every iteration, runs the model to recalculate the derivative, and the Trapazoidal integration equation calculates a corrector value of \( N_{t_2} \). When the predicted value set by the solver is equal, within tolerance, of the corrector value, a converged value of \( N_{t_2} \) has been calculated, and time is ready to advance.

Methods that require such an iteration are called *implicit* methods. There are three implicit integration equations available in NPSS. They are the Trapazoidal equation shown above, 1\(^\text{st}\) Order Gear equation and 2\(^\text{nd}\) Order Gear equation. The form of the two Gear equations is:

**1\(^\text{st}\) Order Gear**

\[
N_{t_2} = N_{t_1} + \left. \frac{dN}{dt} \right|_{t_2} (t_2 - t_1)
\]

**2\(^\text{nd}\) Order Gear**

\[
N_{t_2} = N_{t_1} + \left( \left. \frac{1}{3} \frac{dN}{dt} \right|_{t_1} + \frac{2}{3} \left. \frac{dN}{dt} \right|_{t_2} \right) (t_2 - t_1)
\]

Note that 1\(^\text{st}\) Order Gear differs from Euler in that it assumes that the derivative over the interval is equal to the value at the end of the interval, while Euler assumes that it is equal to the value at the beginning. The 2\(^\text{nd}\) Order Gear takes the weighting of the derivative at beginning and end of the time interval from equal, as it is in the Trapazoidal, to one giving more weight to the derivative at the end of the interval. The 1\(^\text{st}\) Order Gear equation is used by default in NPSS.

It is the job of the *TransientExecutive* object to use the NPSS solver to perform time integrations while also solving the steady-state equations in the model. Setting up the TransientExecutive to perform the integrations automatically defined by standard NPSS components is discussed in Section 7.2. Constructing special time integrations is discussed in Section 7.6.
Just as NPSS automatically creates a solver in the top-level assembly (see Section 4.8), it also automatically creates a TransientExecutive object. The top-level solver is automatically named `solver`; the top-level TransientExecutive is automatically named `transient`. Other solvers can be created in assemblies under the top-level assembly (see Section 4.8.5), but this is not the case with the TransientExecutive. The top-level TransientExecutive object controls the time step and time integration activities for the entire model, thus only one such object can exist. Users, therefore, never need to explicitly instantiate an object of type `TransientExecutive`. They need only address the attributes and use the member functions belonging to the global TransientExecutive object named `transient`.

The variation of spool speed with time from the fuel ramp example above might be as follows:

![Graph of spool speed](image1)

The choice of time step size in a transient simulation is important. A time step size of 0.1 seconds, for example, is adequate to resolve the variation in spool speed for the case shown above. However, one of the important parameters during an acceleration is minimum surge margin. The following plot shows the variation of surge margin with time when a time step of 0.1 seconds is used, compared with a time step of 0.02 seconds. The steady-state surge margin is also shown for reference.

![Graph of surge margin](image2)
If a time step of 0.1 were used, one would conclude that the surge margin never drops below 7%. In reality, however, it approaches 2%.

In the example presented above, even though the fuel flow increases at a constant rate over 3 seconds, it is evident that the most rapid acceleration takes place during the first 0.6 seconds after the start of the fuel flow ramp. A time step adequate for the majority of the transient is too large for this initial portion, and yet if a small enough time step is chosen to resolve the initial acceleration, many points are wasted on the remainder of the transient. NPSS includes provisions to allow the time step to vary throughout the course of a transient. This variation can either be specified by the user, or can be automatically determined by the system. Automatic time step variation, called adaptive time stepping, reduces the time step as necessary to prevent excessive changes in independent variables between successive points, and increases the time step when the independent variables are changing slowly. Variable time stepping is discussed in Section 7.3.

A final consideration in transient simulation is when to terminate the simulation. In the example above, since the precise timing of the item driving the transient (the fuel flow ramp) is known, the duration of the transient is reasonably clear. This is not always the case, however. For example, instead of specifying fuel flows, one might use the model to derive the fuel flows corresponding to a certain minimum allowable surge margin. In this case the duration of the acceleration is unknown. The termination criterion might be quiescence, or the condition at which a steady-state has been reached, within a specified tolerance. As another example, the user might wish to terminate a simulation as soon as an unacceptable condition is reached, such as a negative surge margin. In NPSS, the user must always specify a maximum duration of a transient simulation, but has the option of specifying other criteria that can terminate the simulation sooner. How to do this is discussed in Section 7.4.

In general, the user should consider the following items when constructing a transient simulation.

1. Component inputs necessary for transient running. A steady-state solution is possible with zero inertia, for example, but a transient simulation is not. The user should review each component using the NPSS Reference Sheets to insure that attributes affecting time-dependent behavior are set appropriately.
2. Output
3. Initial conditions
4. Setup of the events driving the transient
5. Integration method
6. Setup of transient model
7. Time step
8. Termination criteria

These steps are summarized for a basic transient run in the following section. As mentioned periodically above, more details are given concerning options available for several of the steps in later sections.

### 7.2 Running a Basic Transient Simulation

#### 7.2.1 Basic Setup

The integrators necessary to correctly model the time-dependent behavior of the system are automatically created when each component is instantiated, and are added to the solver when autoSolverSetup() is called. (Integrators are discussed in more detail in Section 7.6.) Thus, the basics necessary for transient running as well as steady-state running are established by calling autoSolverSetup() from the top-level assembly. As mentioned in the preceding section, the first special concern of the user beyond the usual steady-state issues is to set the component attributes (such as inertias) necessary for transient running.

#### 7.2.2 Output

After setting the transient-related component attributes, the next item often addressed in setting up a transient simulation is the output desired. The most commonly used form of output for transients is the CaseRowViewer,
with a separate line (or row) for each discrete time, and a column for each variable of interest (global variable
_time_ usually being chosen for the first column). Section 4.9.2.3 discusses the use of CaseViewers. If a
CaseViewer is added to the _postExecutionSequence_ array of the top-level assembly’s Executive (in this case
“solver”), it will be updated with information from each time step. Following is a simple example.

```cpp
OutFileStream caseRow {
    filename = "demo.sampleRuns";
}

CaseRowViewer CaseView {
    outputStreamHandle = "caseRow";
}

CaseView.variableList = { "time:??.??",
                          "Brn.Wfuel:??.??????=Wf",
                          "CmpH.Sh_O.Nmech:????..??=Nmech",
                          "ShH.trqNet:????..??=trqNet",
                          "PERF.Fn:?????.?=Fn",
                          "CmpH.SMN:???.??=SM" }

solver.postExecutionSequence.append("CaseView");
```

If output is desired on less than every time step, instead of placing the CaseViewer in _postExecutionSequence_,
the user should place there a function that calls the viewer’s _update_() member function when desired. Another
option is to write a special _update_() function that overrides the standard one in the CaseViewer. For information
on such tasks, see the _Developer Guide_.

### 7.2.3 Initial Conditions

Normally a transient is initiated from steady-state conditions. (See Chapter 17 for information and a example of
initializing transients at non-steady conditions.) Global variable _time_ holds the current simulation time in
seconds, and is updated automatically during a transient simulation. When NPSS is first launched, _time_ is
initialized to zero, but it is not reset to zero at the end of each transient run. It is the user's responsibility, then, to
set the _time_ to the desired value before running a transient. If a steady-state point is run to establish initial
conditions, this point is not repeated when the transient begins. It is recommended, therefore, that the output
desired for the transient run be set up first, the _time_ variable initialized, and then the initial steady-state point run.
This will print the steady-state point as though it were the first point of the transient, and will show the initial time
as specified by the user. The following is an example, which is assumed to follow the CaseViewer setup lines
given earlier (Section 7.2.2).

```cpp
RunCondition.eq_lhs = "Brn.Wfuel";
RunCondition.eq_rhs = "Wf1";
solver.forceNewJacobian = TRUE;
time = 0.;
CASE++;
run();
```

In the example it is assumed that two steady-state points have been run previously, one at a low power setting
which yielded a burner fuel flow that was stored in global variable _Wf1_, and the other at a high power setting
whose fuel flow has been stored in global variable _Wf2_. The initial condition for the transient, which will be
considered _time_ = 0., corresponds to a burner fuel flow of _Wf1_.

### 7.2.4 Transient Executive

To simulate a transient, the model must be set up so that some specified time-varying event drives the transient
response of the rest of the system (i.e. a forcing function). This event could be a change in environmental
conditions simulating an aircraft maneuver, fuel flow changes resulting from simulated operator input, system
configuration changes simulating a mechanical failure such as a shaft break, etc. The fuel ramp example given above could be implemented as follows.

```c
real Ramp( real time, real tstart, real duration, real Y1, real Y2 ) {
  real Y, slope;
  if ( duration < 0 ) {
    duration = 0;
  }
  if ( duration ) {
    slope = ( Y2 - Y1 ) / duration;
  }
  if ( time <= tstart ) {
    Y = Y1;
  } else if ( time > tstart && time < tstart+duration ) {
    Y = Y1 + ( slope * (time-tstart) );
  } else {
    Y = Y2;
  }
  return Y;
}

RunCondition.eq_rhs = "Ramp( time, 0.2, 3.0, Wf1, Wf2 )";
```

Assume an Independent object is active in the solver that makes burner fuel/air ratio an independent variable. Dependent object RunCondition is also active in the solver to allow the user a convenient means to specify some desired operating condition for steady-state solutions. It was used in the example above to establish the initial condition. Function Ramp() is a general ramp function that returns argument Y1 for times less than tstart, Y2 for times greater than a specified interval (duration) after tstart, and a linear interpolation between the two for intermediate times. RunCondition is adapted for transient use by making its right-hand-side a time-varying fuel flow returned by Ramp(). Effectively, the model fuel/air ratio is varied by the solver so that the fuel flow matches the schedule returned by Ramp().

### 7.2.5 Integration Method

If it is known that an integration method other than the default is desired, an appropriate time to change methods is after the solver has been set up and before switching to transient mode. The integration method can be set as follows:

```c
Transient.integrationType = "TRAPEZOIDAL";
```

Some integration methods produce a different solver and TransientExecutive setup than others. For the present example, the default integration method (GEAR_1ST_ORDER) will be accepted. Other integration methods are:

- GEAR_2ND_ORDER
- TRAPEZOIDAL
- EULER

These are discussed further in Chapter 17.

### 7.2.6 Setup of Transient Mode

The mode of the solver is determined by its solutionMode attribute, which is an Option variable. By default this attribute equals STEADY_STATE. Up to this point, nearly all examples have involved steady-state running, so it was unnecessary to explicitly set the attribute. After the solver has been set up for the transient in steady-state mode, it can be switched to transient mode as follows:
setOption( "solutionMode", "TRANSIENT" );

Function `setOption()` should be called from the top-level assembly (global scope) since more objects than just the solver have a `solutionMode` attribute that must be set to `TRANSIENT` for a successful simulation. This should normally be done after all necessary independents and dependents have been set up in the solver. When the solver and TransientExecutive are switched to transient mode, the TransientExecutive retrieves configuration information from the solver, and may make some modifications to dependent conditions in the solver. The easiest way to insure a proper setup is to simply configure the solver in steady-state mode, then switch to transient mode.

If it is necessary to make changes to the solver configuration while in transient mode, take the following steps:

1. Clear the TransientExecutive setup by issuing the command `transient.clear()`. This reestablishes the solver setup that existed before the switch to transient mode. Function `transient.clear()` is automatically called whenever `solutionMode` is switched from `TRANSIENT` to something else.
2. Modify the solver setup as desired, following the usual rules for solver setup changes (see Section 4.8.3 for example).
3. Reestablish the correct solver and TransientExecutive configurations for transient running by issuing the command `transient.setup()`.

This procedure should also be followed if an integration method is changed to or from type `EULER`. The Euler integration method is an explicit method and requires a different configuration than the other methods, which are implicit.

After the model has been switched to transient mode, internal storage areas used for variable histories can be initialized using the conditions established by the initial point.

```
initializeHistory();
```

The `initializeHistory()` function is a global function and should be called from global scope. It should only be called after the system has been switched to transient mode. It need be called only if the initial conditions for the transient to be run differ from the final conditions of the most recent transient.

### 7.2.7 Time Step

The time step to be used can be addressed next. The simplest, and default, time stepping method uses a constant time step throughout the duration of the simulation. The value of this time step is set by the `baseTimeStep` attribute of the TransientExecutive as follows. (Variable time stepping is covered in Section 7.3).

```
transient.baseTimeStep = 0.10;
```

NPSS allows for certain objects called `time-discrete objects` that can directly control the time step from outside the TransientExecutive. Thus it is possible in models with such objects for the time step to vary even when constant time stepping has been selected in the TransientExecutive. The creation of time-discrete objects is an advanced topic touched upon in Chapter 17 and dealt with in more detail in the `Developer Guide`. There are no time-discrete objects among the standard NPSS elements and subelements.

### 7.2.8 Termination Criteria

Finally, the user must establish termination criteria for the transient. At a minimum, the user must specify a value of `time` at which the simulation will terminate. This is done using attribute `stopTime` belonging to the TransientExecutive:

```
transient.stopTime     = 3.60;
```
Note that since \texttt{time} is not necessarily zero at the start of a transient, \texttt{stopTime} is not necessarily the number of seconds simulated. The duration of the transient is \texttt{stopTime} less the value of variable \texttt{time} when the transient is initiated.

Additionally, the user can set a maximum number of iterations that can be run on a transient point. If the max limit is hit, the solver will assume convergence and move on to the next point. This is done using the attribute \texttt{maxIterations} belonging to the \texttt{TransientExecutive}.

\begin{verbatim}
    transient.maxIterations = 5;
\end{verbatim}

The user can also limit the maximum number of iterations that can be run on a transient point by setting the Frame Rate (specified in milliseconds). If the max limit is hit, the solver will assume convergence and move on to the next point. This is done using the attribute \texttt{frameRate} belonging to the \texttt{TransientExecutive}.

\begin{verbatim}
    transient.frameRate = 50;  // Frame Rate is 50 milliseconds
\end{verbatim}

Establishing other criteria for termination is discussed in Section 7.4.

### 7.2.9 Running the Transient

Once these steps have been taken, the transient is run with the usual \texttt{run()} command issued from global scope:

\begin{verbatim}
CASE++;
run();
CaseView.display();
\end{verbatim}

All time steps converged as part of the transient will bear the same \texttt{CASE} number.

Transient simulations can be interrupted and restarted easily. The following code produces the same results as a single transient run from time = 0 to time = 3.6.

\begin{verbatim}
setOption( "solutionMode", "STEADY_STATE" );
RunCondition.eq_rhs = "Wf1";
solver.forceNewJacobian = TRUE;
time = 0.;
CASE++;
run();

RunCondition.eq_rhs = "Ramp( time, 0.2, 3.0, Wf1, Wf2 )";
setOption( "solutionMode", "TRANSIENT" );
initializeHistory();
transient.baseTimeStep = 0.10;
transient.stopTime     = 0.60;
CASE++;
run();

transient.stopTime     = 3.60;
run();
CaseView.display();
\end{verbatim}

Transients driven by a different time-varying event can be set up using the \texttt{transient.clear()} and \texttt{transient.setup()} commands discussed in Section 7.2.6, as in the following example.

\begin{verbatim}
real Fn1 = PERF.Fn;
real Fn2 = Fn1 * 0.80;
transient.clear();
RunCondition.eq_lhs = "PERF.Fn";
RunCondition.eq_rhs = "Ramp( time, 0.2, 2.0, Fn1, Fn2 )";
solver.forceNewJacobian = TRUE;
transient.setup();
time = 0.;
transient.stopTime = 2.40;
\end{verbatim}
7.3 Variable Time-Step Computation

By default, the time-increment between points in any transient simulation is constant, and is equal to the _baseTimeStep_ attribute of the TransientExecutive object. As illustrated in Section 7.1, it is desirable in some cases to vary the time-step during a simulation. During periods in which the model states change slowly, a relatively large time step can be used. During periods in which one or more model states change rapidly, a smaller time step must be used to accurately model the event. This can be accomplished either automatically, or as specified by the user.

7.3.1 Adaptive Time Stepping

When using adaptive time stepping, the user should still specify _transient.baseTimeStep_ to be used at the beginning of the transient, as in constant time step running. The following attributes provide the most commonly used controls over adaptive time stepping:

- _dxTransLimit_
- _dxTransLimitType_
- _maxTimeStep_
- _minTimeStep_

Attribute _transient.dxTransLimit_ specifies the maximum allowable change in an independent variable from one time step to the next. Attribute _dxTransLimitType_ specifies whether _dxTransLimit_ is _FRACTIONAL_ or _ABSOLUTE_. By default it is _FRACTIONAL_. When the independent variables are changing slowly, large time steps, up to the limit specified by _maxTimeStep_, will be taken. As necessary, small time steps, down to the limit specified by _minTimeStep_, will be taken to insure that no independent variable changes more than _dxTransLimit_ during the step.

Adaptive time stepping is activated by setting attribute _timeStepMethod_ equal to _ADAPTIVE_ as in the following example.

```plaintext
SetOption( "solutionMode", "TRANSIENT" );
i nitializeHistory();
transient {
    timeStepMethod = "ADAPTIVE";
    baseTimeStep   = 0.10;
    dxTransLimit   = 0.05;
    maxTimeStep    = 0.20;
    minTimeStep    = 0.01;
    stopTime       = 3.60;
}
CASE++;
run();
CaseView.display();
```

The points run by this method for the example presented in Section 7.1 are illustrated below.
The times steps range from the minimum specified (0.01) to the maximum (0.20). The number of time steps taken is 69. A constant time step of 0.01 over the same interval would require 360 steps.

Adaptive time stepping is best suited for models that are not highly discontinuous with respect to time. If the user anticipates that the system being simulated will exhibit periods of very slow change followed immediately by periods of very abrupt change, care must be exercised to insure that the abrupt change is not skipped over. Setting an appropriately restrictive value for `maxTimeStep` can insure this.

Although in some cases adaptive time stepping can shorten simulation time by greatly reducing the number of time steps taken, it can increase execution time in other cases. This is because the method takes additional passes through the model to estimate the rate of change of the independent variables. In the example above, 459 model passes were required for 69 time steps, for an average of 6.65 passes per time step. The same problem run with a constant time step of 0.01 requires 1053 model passes for 360 time steps for an average of 2.93 passes per time step – less than half that required for adaptive time stepping. In this case adaptive time stepping reduced the number of time steps by about 80%, but the number of model passes by only 56%. Adaptive time stepping, therefore, must dramatically reduce the number of time steps in order to result in a net execution time savings. It has the advantage, however, of clustering data points where they are usually most needed.

The model can be restored to constant time steps by setting `timeStepMethod` to "CONSTANT_DT".

The prediction passes made by the adaptive time stepping method take place before calls to *time-discrete objects*. These are components that either directly manage the time step, or perform special operations at each time step. They can be useful in simulating controls. If a time-discrete object assumes that the model data available to it is from the previous time step, its logic can be corrupted by the predictor passes of the adaptive time stepping method. Care should be taken, therefore, when using time-discrete objects together with adaptive time stepping. See the *Developer Guide* for more information.

### 7.3.2 User-Defined Time-Step Computations

If `timeStepMethod` is set to `USER_DEFINED`, then an expression can be placed in attribute `userTimeStepFunction` that will be evaluated after each time step to determine the next time step. Normally this is a function call that contains the logic required to properly set the time step through the transient. For example:
real varTimeStep1() {
    real dt;
    if      ( time < 0.195 ) { dt = 0.10; }
    else if ( time < 0.595 ) { dt = 0.01; }
    else if ( time < 0.995 ) { dt = 0.02; }
    else                     { dt = 0.20; }
    return dt;
}

RunCondition.eq_rhs = "Ramp( time, 0.2, 3.0, Wf1, Wf2 )";
setOption( "solutionMode", "TRANSIENT" );
initializeHistory();
transient {
    timeStepMethod = "USER_DEFINED";
    userTimeStepFunction = "varTimeStep1()";
    baseTimeStep = 0.10;
    stopTime = 3.60;
} 
CASE++;
run();
CaseView.display();

Similar results may be had by setting timeStepMethod to CONSTANT_DT and directly varying the baseTimeStep after each solver convergence:

void varTimeStep2() {
    real dt;
    if      ( time < 0.195 ) { dt = 0.10; }
    else if ( time < 0.595 ) { dt = 0.01; }
    else if ( time < 0.995 ) { dt = 0.02; }
    else                     { dt = 0.20; }
    transient.baseTimeStep = dt;
}

RunCondition.eq_rhs = "Ramp( time, 0.2, 3.0, Wf1, Wf2 )";
setOption( "solutionMode", "TRANSIENT" );
initializeHistory();
transient {
    timeStepMethod = "CONSTANT_DT";
    baseTimeStep = 0.10;
    stopTime = 3.60;
} 
solver.postExecutionSequence.insertAt( 0, "varTimeStep2" );
CASE++;
run();
CaseView.display();

7.4 Controlling Termination of a Transient Run

As mentioned earlier, the user must specify a value for the TransientExecutive attribute stopTime. In any event, the transient simulation will terminate when time becomes greater than stopTime. The user may also, however, supply an expression for attribute terminateCondition. This expression, which may be a function call, must return an integer value, interpreted as a Boolean. If the expression evaluates to 0, and time is less than or equal to stopTime, the transient simulation continues. If the expression evaluates to nonzero, the simulation terminates. This is useful when the exact duration of the transient is unknown.

Suppose, for example, the fuel ramp example from Section 7.1 were modified to run until 7% surge margin is obtained, then accelerate at constant 7% surge margin until the final fuel flow is obtained, then hold that fuel flow until the shaft speed is essentially steady. The fuel flow schedule corresponding to such an acceleration is desired, as well as the time required for the acceleration. To accomplish this, functions runtoSM() and runtoWF() are
written to modify the solver to run either to 7% surge margin, or to constant fuel flow. Another function, 
runSwitch(), is placed in solver.postExecutionSequence to control the solver switching. Finally, function 
steady() is written to determine when the shaft speed is constant enough to consider the acceleration over, and a 
call to this function made via terminateCondition. The complete example is as follows.

```cpp
void runtoSM() {
    transient.clear();
    RunCondition.eq_lhs = "CmpH.SMN";
    RunCondition.eq_rhs = "7.0";
    solver.forceNewJacobian = TRUE;
    transient.setup();
    initializeHistory();
}

void runtoWF() {
    transient.clear();
    RunCondition.eq_lhs = "Brn.Wfuel";
    RunCondition.eq_rhs = "Wf2";
    solver.forceNewJacobian = TRUE;
    transient.setup();
    initializeHistory();
}

void runSwitch() {
    if ( RunCondition.eq_lhs != "CmpH.SMN" && CmpH.SMN < 7.0 ) {
        runtoSM();
    }
    else if ( RunCondition.eq_lhs == "CmpH.SMN" && Brn.Wfuel > Wf2 ) {
        runtoWF();
    }
}

real lastVal;
int steady( real tstart, string var, real tol ) {
    if ( time <= tstart ) {
        lastVal = var->value;
        return 0;
    }
    real val = var->value;
    int stop = 0;
    if ( lastVal != 0 && abs( (val-lastVal)/lastVal ) < tol ) {
        stop = 1;
    }
    lastVal = val;
    return stop;
}

RunCondition.eq_rhs = "Ramp( time, 0.2, 3.0, Wf1, Wf2 )";
solver.executionSequence.append( "runSwitch" );
setOption( "solutionMode", "TRANSIENT" );
initializeHistory();
transient {
    timeStepMethod = "CONSTANT_DT";
    baseTimeStep   = 0.02;
    terminateCondition = "time > 0.3 && steady( 0., "ShH.Nmech")", 1e-4 )";
    stopTime       = 4.00;
}
CASE++;
run();
CaseView.display();
```
The resulting mechanical speed variation is shown below.

Although `stopTime` was set to 4 seconds, the transient is over in less than a second, and much time is saved by its terminating then.

In the example, a user-written function named `steady()` was used in connection with `terminateCondition` to determine when the simulation should stop. A built-in member function of all `TransientExecutive` objects named `quiescence()` is also available. Function `transient.quiescence()` returns 1 or `TRUE` when the fractional changes of all independent variables are less than attribute `transient.quiescenceTolerance`.

The simulation can be made to stop after each time step by setting `terminateCondition` simply to "1". The transient can be advanced another time step by issuing the `run()` command. Such a technique can be useful for debugging transient simulations in interactive mode (Section 2.3).

### 7.5 TransientExecutive Attributes

This section summarizes the attributes of `TransientExecutive` objects discussed in the preceding sections, and discusses some additional controls available to the user. Full details on the `TransientExecutive` and related object types is found in Chapter 17.

The variables in the table below are attributes of the `TransientExecutive` object, i.e. they are referenced by `transient.attribute_name`. Bear in mind that two variables important to transient simulations reside in the `top-level assembly` rather than in object `transient: time` and `timeStep`. Global variable `time` can be set by the user before a transient run is launched, but is controlled by the system during the transient. Global variable `timeStep` is always an output variable, and reflects the time stepping logic of the `TransientExecutive` object.

The following attributes are commonly set by users in order to control transient simulations.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseTimeStep</td>
<td>real</td>
<td>Time step to be used unless overridden by other logic such as adaptive time stepping, user-defined time stepping, or time-discrete objects. Units are seconds.</td>
<td>0.05</td>
</tr>
<tr>
<td>dxTransLimit</td>
<td>real</td>
<td>Maximum allowable change in an independent variable from one time step to the next under adaptive time stepping. Whether the specified change is fractional or absolute is determined by <code>dxTransLimitType</code>. Used only when <code>timeStepMethod = &quot;ADAPTIVE&quot;</code>.</td>
<td>0.10</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Type</td>
<td>Description</td>
<td>Default Value</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>dxTransLimitType</td>
<td>Option variable</td>
<td>Allowable values are &quot;FRACTIONAL&quot; and &quot;ABSOLUTE&quot;.</td>
<td>&quot;FRACTIONAL&quot;</td>
</tr>
<tr>
<td>integrationType</td>
<td>Option variable</td>
<td>Default integration method for all integrations. This can be overridden in individual integrators. Allowable values are &quot;GEAR_1\textsuperscript{st} ORDER&quot;, &quot;GEAR_2\textsuperscript{nd} ORDER&quot;, &quot;TRAPEZOIDAL&quot;, which are implicit methods, and &quot;EULER&quot; which is an explicit method.</td>
<td>&quot;GEAR_1\textsuperscript{st} ORDER&quot;</td>
</tr>
<tr>
<td>maxTimeStep</td>
<td>real</td>
<td>Maximum allowable adaptive time step. Used only when timeStepMethod = &quot;ADAPTIVE&quot;.</td>
<td>1.0</td>
</tr>
<tr>
<td>minTimeStep</td>
<td>real</td>
<td>Minimum allowable adaptive time step. Used only when timeStepMethod = &quot;ADAPTIVE&quot;.</td>
<td>0.0001</td>
</tr>
<tr>
<td>quiescenceTolerance</td>
<td>real</td>
<td>Used by member function \texttt{quiescence()}. If the fractional change of all state variables over a time step is less than \texttt{quiescenceTolerance}, \texttt{quiescence()} returns 1 or \texttt{TRUE}. Otherwise it returns 0 or \texttt{FALSE}. Function \texttt{quiescence()} can be used in attribute terminateCondition as part of a termination test.</td>
<td>0.01</td>
</tr>
<tr>
<td>solutionMode</td>
<td>Option variable</td>
<td>Mode of the TransientExecutive object. Allowable values are &quot;ONE_PASS&quot;, &quot;STEADY_STATE&quot;, and &quot;TRANSIENT&quot;.</td>
<td>&quot;STEADY_STATE&quot;</td>
</tr>
<tr>
<td>stopTime</td>
<td>real</td>
<td>Maximum value of ( time ) after which the transient simulation will be terminated.</td>
<td>0.0</td>
</tr>
<tr>
<td>terminateCondition</td>
<td>string</td>
<td>Expression that should evaluate to an integer, interpreted as a Boolean (0 = &quot;false&quot;, nonzero = &quot;true&quot;). The simulation terminates when the expression evaluates to &quot;true&quot;. The expression can include calls to other functions.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>timeStepMethod</td>
<td>Option variable</td>
<td>Time step method. Allowable values are &quot;CONSTANT_DT&quot;, &quot;ADAPTIVE&quot;, and &quot;USER_DEFINED&quot;.</td>
<td>&quot;CONSTANT_DT&quot;</td>
</tr>
<tr>
<td>userTimeStepFunction</td>
<td>string</td>
<td>Expression that should evaluate to a real value giving the time step to be used. The expression can include calls to other functions. Used only when timeStepMethod = &quot;USER_DEFINED&quot;.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>maxIterations</td>
<td>int</td>
<td>Maximum number of iterations that can be run on a transient point. If the max limit is hit, the solver will assume convergence and move on to the next point.</td>
<td>0</td>
</tr>
<tr>
<td>frameRate</td>
<td>int</td>
<td>Maximum number of milliseconds allowed for iterations that can be run on a transient point. If the max limit is hit, the solver will assume convergence and move on to the next point.</td>
<td>0</td>
</tr>
</tbody>
</table>
### 7.6 Integrators

Integrators are a subclass of the Dependent class. They function identically to any other Dependents during steady-state operation, but during transient operation they utilize additional attributes and methods to carry out extra responsibilities. In fact, some of the default NPSS components, such as the Shaft Element, create Integrators as part of their class definition. An integrator has all the attributes of a Dependent. It also has an attribute, `stateName`, that references a state variable in the model, such as metal temperature, and an attribute, `derivativeName`, that references the time derivative of this state. The `state` and `derivative` attributes hold the values of the model parameters pointed to by the `stateName` and `derivativeName` attributes.

The Integrator class calculates two different error terms depending on the solution mode. For steady-state mode, the error term in the Integrator is $error = (y_1 - y_2)$ [or $(y_1 - y_2)/y_{Ref}$], the same as any other Dependent. For transient mode, the error term changes to $error = state - stateDemand$ [or $(state - stateDemand)/y_{Ref}$], where `stateDemand` is the result of the integrator equation.

#### 7.6.1 Implicit Integration

There are four integration equations available that calculate the value of the state at the end of the time step using the state value at the beginning of the time step, the time step size, and the state derivative values from the beginning of the time step, the end of the time step, or both. They are 1\textsuperscript{st} order Gear, 2\textsuperscript{nd} order Gear, Trapazoidal or Euler. The `integrationType` attribute in the transient executive sets the default type for all Integrators. However, this default can be overwritten for each Integrator, so each Integrator can use a different integration equation. These equations, save Euler, use the value of the state derivative from the end of the time step. For example the Trapazoidal equation is of the form:

$$x_{i+dt} = x_i + \frac{1}{2} \left( \frac{dx}{dt}_{i+dt} + \frac{dx}{dt}_{i} \right) \cdot dt$$
The state derivative at the end of the time step, however, is a function of the state value at the end of the time step. This results in a dependent condition that must be iterated to solve. A guess or prediction of the value of the state at the end of the time step must be made before the derivative can be calculated. The value of the state used in the model is decoupled from the results of the integration equation by capturing the results of the integration equation in a parameter called \textit{stateDemand}. The above equation for the Trapazoidal equation then becomes:

\[
x_{\text{demand \_ end}} = x_i + \frac{1}{2} \left( \frac{dx}{dt \_ \text{end}} + \frac{dx}{dt \_ i} \right) \ast dt
\]

The difference between the value of the state variable set before each iteration pass and the value of \textit{stateDemand} calculated after the iteration pass form an error condition, called the corrector error, that must be driven to zero. The solver varies the state variable, or state substitution variable from which the state can be calculated, to close the corrector error term in exactly the same process used to solve steady-state dependencies. And in fact, the solver is simultaneously solving the corrector error for Integrators and regular Dependent errors for non-dynamic aspects of the model, such as mass and energy continuity.

### 7.6.2 Explicit Integration

The above discussion applies only to the implicit methods, 1\textsuperscript{st} order Gear, 2\textsuperscript{nd} order Gear, and Trapazoidal. The Euler equation is an explicit integration form which depends only on the derivative at the beginning of the time step, which is known without iteration, in order to calculate the end state value. Therefore, any Integrator set to Euler is removed from the list of Integrators that require iteration. The state value in the model is set directly to the \textit{stateDemand} value only once at the start of each time step, and it remains at that value for any subsequent iterations required to converge any existing error terms, including other Integrators integrated with implicit methods.

To set an Integrator to integrate using Euler, there must be an Independent object in the solver setup that points directly to the same model parameter referenced by the \textit{stateName} parameter in the Integrator. The reason is that for transient integration the Independent and Integrator are paired in a way that they are not for steady-state. For transient the Independent must vary the state variable directly or indirectly by varying a parameter that the model is set up to use for calculating the state variable (a so-called state substitution variable). If the partner Independent of an Integrator points to a state substitution variable, that means the state variable is calculated by modeling code, which is incompatible with Euler integration, which calculates the state variable value directly.

### 7.6.3 Switching Individual Integrators to Steady-State

Individual Integrators can be switched from transient mode back to steady-state mode, while leaving the simulation as a whole in transient mode. The result is that the Integrator moves back to calculating the steady-state error term. When each time step is converged, therefore, the Integrator is at its steady-state condition. This is done when a given state very nearly reaches its steady-state value at the end of each time step. In this situation, the impact on the dynamic behavior of the model of the state associated with the Integrator is nearly zero. Setting an Integrator to steady-state reduces the number of states being tracked through time.

The number of states being integrated is referred to as the order of the system. Switching an Integrator to steady-state reduces the order of the system by one. As such, it is a very effective way to perform reduced order modeling, which seeks to determine the minimum set of states that must be integrated to provide an accurate time simulation of the system being modeled.

Integrators are not normally created outside the definition of an Element or Subelement class. This is because the Integrator must work with Elements and Subelements that contain state and state derivative variables, and in turn, these Elements and Subelements must have an Integrator to properly integrate the state in order for it to represent the transient behavior of its portion of the model.
### Table 34. Integrator Input Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>derivativeName</td>
<td>string</td>
<td>Name of a variable holding the current value of the time derivative of the variable named by stateName.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>desiredDerivativeValue</td>
<td>Real</td>
<td>When the solutionMode is set to &quot;SET_DERIVATIVE,&quot; this attribute holds the derivative value at which the model is to be at convergence.</td>
<td>0.0</td>
</tr>
<tr>
<td>eq_RefTransient</td>
<td>string</td>
<td>Expression evaluated only during a transient run that yields a value of yRef. If an expression is not given, then the expression in eq_Ref is used during transient as well as steady-state operation. And if eq_Ref is not given, then eq_rhs is used to determine yRef</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>integrationType</td>
<td>Option variable</td>
<td>Integration method for this integrator. Different integrators in the same system can use different integration methods. Allowable values are &quot;GEAR_1ST_ORDER&quot;, &quot;GEAR_2ND_ORDER&quot;, &quot;TRAPEZOIDAL&quot;, which are implicit methods, and &quot;EULER&quot; which is an explicit method. If unset, the default set in the TransientExecutive is used. It defaults to &quot;GEAR_1ST_ORDER&quot;.</td>
<td>TransientIntegrationType</td>
</tr>
<tr>
<td>solutionMode</td>
<td>Option variable</td>
<td>&quot;ONE_PASS&quot;, &quot;STEADY_STATE&quot;, &quot;TRANSIENT&quot;, &quot;SET_DERIVATIVE.&quot;</td>
<td>&quot;STEADY_STATE&quot;</td>
</tr>
<tr>
<td>stateName</td>
<td>string</td>
<td>Name of the variable whose time-varying value is to be determined by integrating the derivative specified by derivativeName.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>timeConstantExpr</td>
<td>string</td>
<td>A string containing an expression that when evaluated will yield a single real number representing the time constant value of the state.</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

As indicated in Table 34, some Integrator object attributes such as integrationType have default values set by attributes in the TransientExecutive. When the value of such an attribute is explicitly set by the user in the Integrator object, that value is locked. This means that any subsequent change to the default value in the TransientExecutive will not change the value explicitly set in the integrator. Chapter 17 explains how the value of an Integrator object attribute can be unlocked so that it reverts to the default value set in the TransientExecutive. Chapter 17 also covers other attributes and topics of interest to advanced users and includes extensive examples.

### Table 35. Integrator Output Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>derivative</td>
<td>real</td>
<td>Current derivative of the state variable, as given by derivativeName.</td>
</tr>
<tr>
<td>state</td>
<td>real</td>
<td>Current value of the state variable, as given by stateName.</td>
</tr>
<tr>
<td>stateDemand</td>
<td>real</td>
<td>Calculated value from the Integrator equation, including the Euler equation.</td>
</tr>
<tr>
<td>timeConstant</td>
<td>real</td>
<td>Value of the time constant of the state. If a timeConstantExpr is given, this attribute holds the results of its evaluation. If not, then the time constant is calculated by the following formula: timeConstant = fabs(state) *</td>
</tr>
</tbody>
</table>
8 Linear Model Generator

8.1 Introduction

The Linear Model Generator (LMG) is an optional object created at run-time that generates a linear representation of the full non-linear model about a converged point of the non-linear model. A linear model relates changes in selected state derivative and model output variables to changes to the corresponding state and selected input variables. The linear model thus provides a means of estimating the model state derivative and output values for a given set of state and input values by means of simple and very fast matrix multiplications rather than complete convergence of the non-linear thermodynamic model. The linear models also represent the characteristic response of the full engine model and are thus used in the development in engine control algorithms.

The linear model, by definition, will exactly agree with the non-linear model at the point where it was generated. However if the linear model is used to estimate the output and state derivative values at input and state variable values away from the base values about which the model was generated, the results will not exactly agree with the full non-linear model. The region where the linear model produces results that are sufficiently close to the non-linear model depends on the non-linearity of the full model and the amount of difference that the user finds acceptable.

Coverage for a broader range of operating conditions can be obtained by creating a number of linear models at operating conditions spaced around the operating range of interest and then interpolating between them to represent performance at any operating condition. The number and spacing of these points is dependent on non-linearity of the full model and the allowed deviation between the linear and non-linear models. Collectively, this set of linear models is called a piece-wise continuous linear model. The Linear Model Generator object will generate only one linear model at a time, and it does not retain any information about previous linear models that may have been generated during the current run. To obtain the individual linear models required for a piece-wise continuous linear model, simply repeat the process of converging the non-linear model to provide a base point, modifying the solver setup, generating a linear model, and printing out the linear model matrices and baseline variable values at each condition required.

A linear model is generated about a converged point of the non-linear model by determining the sensitivity of each derivative and output variable to small changes in the state and input variable. This is done by perturbing each state and input variable in turn, determining the changes in the state derivative and output variables in response to these perturbations, and then dividing those changes by the change in the state or input value. The resulting sensitivity terms are gathered into four matrices. Perturbations to state variables can be indirect by using state substitute variables.

The first matrix, commonly referred to as the $A$ matrix, contains the sensitivities of each of the selected state derivatives to changes in the states. The second, or $B$ matrix, contains the sensitivities of the selected output variables to changes in the state variables. The $C$ matrix contains the sensitivities of the state derivatives to changes in the input parameters. And the $D$ matrix contains the sensitivities of the output variables to changes in the input variables. Of these four matrices, only the $A$ matrix is required to be square since the number of state and state derivative terms must be equal. The number of input and output variables selected for inclusion in the linear model is arbitrary. Thus the other three matrices can be rectangular. The baseline variable values for each variable is also given as part of the linear model output.

8.2 Basic Setup

A Linear Model Generator must be manually added to a model. It is not automatically created at model start-up. At a minimum, the input, output, state and state derivative variable names must be specified. All other input variables have default values. There are two ways to specify the variables to be used, and both can be intermixed.

- First, the variables can be added one at a time by using the addInput, addOutput or addState functions as appropriate. The addState function takes the name of both a state and state derivative variable because the number of each must always be equal and because the state and state derivative variables must correspond to each other. When state substitution is being used, the addState function must be used to add the state, state
The `addInput` and `addState` functions have three optional arguments. They are the perturbation size, the perturbation type and a bias value. Arguments cannot be skipped. To specify the perturbation type, a value for perturbation size must be supplied. Similarly, if a bias value is to be given, values for perturbation size and type must be supplied, even if they are simply restating the default value. Once the variable or variables have been verified, they are added to the `inputVars`, `outputVars`, `stateVars`, or `derivativeVars` string arrays. If state substitution is being used, the three `addState` arguments described in this section are not optional. This is because the state substitute variable name is input following these three arguments.

- The second method is to bypass the add functions and assign arrays of variable names directly to these string arrays. Even though they are added as a group, all the variables are confirmed to be valid variable names just as is done in the add functions. It should be noted that when variables are added by this method any previous content in the array being assigned is deleted. In addition, if the array being set is the `inputVars` or `stateVars`, all the associated perturbation size and type and bias arrays are emptied. Functions to set the perturbation size, perturbation type and bias are available to change these parameters for variables previously added by either method. As previously stated, the `addState` function must be used to input state information for cases where state substitution is being used.

### 8.2.1 Creation and Setup of the Linear Model Generator

The following example defines a linear model of a turbojet engine. Both methods for specifying variable names are used. Since the array method was used to specify the input variable, the desired perturbation size could not be given when the variables were added. Therefore, the `setPerturb` function is used to set the perturbation size. Separate functions are used to set the perturbation type and bias.

```cpp
LinearModelGenerator TestLinearModel {

    // *** REQUIRED INPUT ***

    // assign arrays of variable names directly to the linear model attribute
    inputVars = {"BURN.fuelFlow", "NOZ.AthColdDes"} 
    outputVars = {"CPHI.Fl_O.Pt", "CPHI.Fl_O.Tt", "CPHI.Fl_O.W",  
                  "BURN.Fl_O.Pt", "BURN.Fl_O.Tt", "BURN.Fl_O.W",  
                  "NOZ.Fl_I.Pt", "NOZ.Fl_I.Tt", "NOZ.Fl_I.W",  
                  "NOZ.Fl_O.Pt", "NOZ.Fl_O.Tt", "NOZ.Fl_O.W"}

    // use the addState function to set the state and state derivative
    // variables.
    addState("SHHP.Nmech", "SHHP.dNqdt", 0.05, "FRACTIONAL", 0.0); 
    addState("CPHI.S_Qblade.Tmat", "CPHI.S_Qblade.dTmatqdt",  
             0.05, "FRACTIONAL");  
    addState("TBHI.S_Qblade.Tmat", "TBHI.S_Qblade.dTmatqdt", 0.05);

    // *** OPTIONAL INPUT ***

    // Change the perturbation amount from the default of 0.5% to
    // 3% for the inputs.
    setPerturb("BURN.fuelFlow", 0.03); 
    setPerturb("NOZ.AthColdDes", 0.03);

    reportFileName = "cerr";
    repeatabilityCheck = TRUE;
    printDiagnostics = TRUE;
    defaultPerturbation = 0.02;
}
```
8.2.2 Setup of the Solver During Linear Model Generation

The Solver and the Linear Model Generator cannot both be trying to control the same set of input and state variables. Therefore, when a state or input variable is added to the Linear Model, the Solver setup must be checked to see if that variable is also under the control of the Solver. If so, then the Independent that is trying to control the variable must be removed from the Solver while the linear model is being generated.

Since the Solver must have an equal number of Independents and Dependents, if Independents must be removed from the Solver, an equal number of Dependents must be also be removed. When an Independent controlling a state is removed from the Solver, it is obvious that the Dependent that should be removed is the Integrator Dependent associated with that state. In the case of Independents that control the same model variable as a Linear Model input variable, the Dependent that should be removed is not so obvious.

The following illustrates the input required to make the Solver setup compatible with the Linear Model Generator configuration given above. In this example none of the Solver Independents pointed to Linear Model Input variables, so only State Independent and Integrator Dependents need to be removed.

```plaintext
solver
{
    // remove state independents and integrators from the Solver manually
    removeIndependent("CPHI.S_Qblade.ind_Tmat");
    removeIndependent("TBHI.S_Qblade.ind_Tmat");
    removeIndependent("SHHP.ind_Nmech");
    removeDependent("CPHI.S_Qblade.integ_Tmat");
    removeDependent("TBHI.S_Qblade.integ_Tmat");
    removeDependent("SHHP.integrate_Nmech");
}
```

To return to normal execution of the model following linear model generation, all removed Independents and Dependents must then be added back to the Solver. When generating a number of linear models around different operation conditions, any removed Independents and Dependents must be returned to the Solver in order for the base point for the next linear model to be converged.

8.2.3 Modifying the Linear Model Setup

Individual variables can be removed from the Linear Model using the `removeInput`, `removeOutput`, and `removeState` functions. Each of these functions returns an integer TRUE/FALSE depending on whether the named variable was found in the appropriate list of linear model variables. A return of FALSE is the only indication that the variable was not found. No error or warning message is issued.

If a state variable is removed, the corresponding derivative variable must also be removed to keep the number of states and derivatives equal. This requirement is enforced by having a single `removeState` function that takes both the name of the state variable and the corresponding state derivative variable. This function will return FALSE if either variable or both variables are not found.

The following illustrates the removal process using the `removeState` function.

```plaintext
// return value not captured
TestLinearModel.removeState ("CPHI.S_Qblade.Tmat");

// return value saved in a new variable
int removeReturn = TestLinearModel.removeState ("CPHI.S_Qblade.Tmat",
    "CPHI.S_Qblade.dTmatqdt");

// test return value immediately without saving to a variable
if(TestLinearModel.removeInput ("BURN.fuelFlow")) {
    // variable removed successfully
```
8.2.4 Generating a Linear Model

Once the Linear Model Generator object has been setup a linear model is generated and automatically output by simply calling the `generate()` function on the Linear Model Generator object from user input.

```cpp
TestLinearModel.generate();
```

Multiple linear models with different combinations of state, input and output variables can be generated for the same base point by changing the setup of the Linear Model Generator and calling the `generate` function. It is not necessary to rerun the base point if only the Linear Model Generator setup is changed.

8.2.5 Printing/Saving a Linear Model

A Linear Model Generator object has internal storage for one linear model at a time. When a second linear model is generated about another converged point, the previous linear model data is deleted. Thus the linear model data must be printed or saved in some way between calls to the `generate()` function. All the necessary calculated linear model parameters are available to the user. Therefore, model input can directly retrieve the desired parameters. For example, the following code saves the A matrix from two different linear models to user-declared array variables A1 and A2.

```cpp
Ambient.Alt = 0.0;
Ambient.MN = 0.2;
Burner.Wf = 2500 / 3600 "lbm/sec";
run();
TestLinearModel.generate();

real A1[][] = TestLinearModel.A;
Burner.Wf = 2000 / 3600 "lbm/sec";
run();
TestLinearModel.generate();

real A2[][] = TestLinearModel.A;
```

This capability can be used to directly output the values to a desired stream. (See Section 4.9.1 for more information on streams.) The following code sends the values to standard out.

```cpp
cout << "Matrix values \n\n";
cout << "A matrix \n " << TestLinearModel.A << endl;
cout << "B matrix \n " << TestLinearModel.B << endl;
cout << "C matrix \n " << TestLinearModel.C << endl;
cout << "D matrix \n " << TestLinearModel.D << endl;
cout << "\n\n Baseline Values \n\n";
cout << "State = " << TestLinearModel.stateBase << endl;
cout << "Derivative = " << TestLinearModel.derivativeBase << endl;
cout << "Input = " << TestLinearModel.inputBase << endl;
cout << "Output = " << TestLinearModel.outputBase << endl;
cout << "Passed Linearity Test = " << TestLinearModel.passedLinearTest << endl;
cout << "Passed Repeatability Test = " << TestLinearModel.passedRepeatTest << endl;
```

A formatted report containing all the calculated linear model variables is automatically created and sent to the location indicated by the `reportFileName` variable. The default location is `cout`. This report contains the A, B, C, and D matrices with each column and row labeled with the variable to which it corresponds. The report also
contains the baseline values of all the input, output, state and state derivative variables. And finally, the report indicates if the linear model passed the linearity check and the repeatability check (if requested).

The built-in report format can be replaced by a user-defined format. This is done by implementing the userReport function of the Linear Model Generator. The userReport function is similar in concept to the prepass, preexecute and postexecute functions available on Elements. Like these functions it takes no arguments, returns void, and has direct access to the data. When defined, the userReport function is automatically called by the execute function immediately after the linear model has been generated. A blank implementation of the userReport function effectively turns off the automatic report creation. The following is a sample implementation.

```cpp
OutFileStream LMGreport {
    filename = "linearModels.out";
}
TestLinearModel {
    void userReport() {
        LMGreport << "\n\nLinear Model\n\n";
        LMGreport << "A matrix \n " << A << endl;
        LMGreport << "B matrix \n " << B << endl;
        LMGreport << "C matrix \n " << C << endl;
        LMGreport << "D matrix \n " << D << endl;
        LMGreport << "\n\nBaseline Values \n\n";
        LMGreport << "State = " << TestLinearModel.stateBase << endl;
        LMGreport << "Derivative = " << TestLinearModel.derivativeBase << endl;
        LMGreport << "Passed Linearity Test = " << TestLinearModel.passedLinearTest << endl;
        LMGreport << "Passed Repeatability Test = " << TestLinearModel.passedRepeatTest << endl;
    }
}
```

### 8.3 Functions

**void addInput(input_name, perturbation_size, perturbation_type, bias)**

Adds the named variable as an input to the Linear Model Generator with the given perturbation size, type and bias. Valid values of perturbation type are "ABSOLUTE" and "FRACTIONAL". The last three arguments are optional. If they are not given, then the default values will be used.

**void addOutput(output_name)**

Adds the named variable as an output to the Linear Model Generator.

**void addState(state_name, derivative_name, perturbation_size, perturbation_type, bias, state_substitute_name)**

Adds the named state and derivative variables to the Linear Model Generator with the given perturbation size, type and bias applied to the state. If state substitution is to be used, the state substitute variable name must be added as the last argument. Valid values of perturbation type are "ABSOLUTE" and "FRACTIONAL". The perturbation size, type and bias arguments are optional when state substitution is not being used. If not given then the default values will be used. Examples of addState usage are provided below.

```cpp
// No state substitution; default perturbation size, type and bias.
addState("volume.density", "volume.dRHOdt");

// No state substitution; non-default bias value
// (the default values for perturbation size and type must be input).
addState("volume.density", "volume.dRHOdt", 0.001, "FRACTIONAL", 0.2);
```
// State substitution being used; perturbation size, type and bias
// values must be input.
addState("volume.density", "volume.dRHOdt", 0.001, "FRACTIONAL", 0.0,
"volume.Pt");

void clear()
Removes all the input, output, state and state derivative variables from the Linear Model Generator as well as any
perturbation size, type and bias values added. This function also removes all calculated values.

void execute()
An implementation of the general execute function. Checks the return value of the generateCondition expression,
if given, and confirms that this Linear Model Generator is not being called as part of the generation process of
another Linear Model Generator located higher in the model hierarchy. This is the function that is automatically
called if the Linear Model Generator object is placed in one solver sequencer lists.

void generate()
Causes a linear model to be generated around the state of the model at the time it is called. It should be called only
after a converged model point if the resulting linear model is to have any meaning. It can be called after as many
different converged points as desired.

real getBias (variable_name)
Returns the bias value for the requested variable. This function will check both the input and state lists. If a
specific bias value has not been set for the requested variable, then the default value is returned.

real getPerturb (variable_name)
Returns the perturbation size for the requested variable. This function will check both the input and state lists. If a
specific perturbation size has not been set for the requested variable, then the default value is returned.

string getPerturbType (variable_name)
Returns the perturbation type for the requested variable. This function will check both the input and state lists. If a
specific perturbation type has not been set for the requested variable, then the default value is returned.

int removeDerivative (derivative_name)
Removes the named derivative variable. Returns integer TRUE if the variable was found and successfully remove,
or FALSE if not.

int removeInput (input_name)
Removes the named input variable. Returns integer TRUE if the variable was found and successfully remove, or
FALSE if not.

int removeOutput (output_name)
Removes the named output variable. Returns integer TRUE if the variable was found and successfully remove, or
FALSE if not.

int removeState (state_name)
Removes the named state variable. Returns integer TRUE if the variable was found and successfully remove, or
FALSE if not.

void revertToDefaults()
Removes any perturbation, perturbation type or bias assigned to any specific input or state variable. All variable
will use the default values associated with the particular Linear Model Generator.

void setBias(variable_name, bias_value)
Assigns a specific bias value to the given input or state variable.
void setPerturb(variable_name, perturb_value)
Assigns a specific perturbation value to the given input or state variable.

void setPerturbType(variable_name, perturb_type_value)
Assigns a specific perturbation type value to the given input or state variable. Valid values of perturbation type are "ABSOLUTE" and "FRACTIONAL".
### 8.4 User Input Attributes

#### Table 36. Linear Model Generator User Input Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>calcMethod</td>
<td>Option variable that determines the differencing technique used to calculate the linear model matrices. Valid values are &quot;NEGATIVE&quot;, &quot;CENTRAL&quot;, &quot;POSITIVE&quot;.</td>
<td>&quot;CENTRAL&quot;</td>
</tr>
<tr>
<td>defaultPerturbation</td>
<td>default perturbation size.</td>
<td>0.005</td>
</tr>
<tr>
<td>defaultPerturbationType</td>
<td>default perturbation type.</td>
<td>&quot;FRACTIONAL&quot;</td>
</tr>
<tr>
<td>derivativeVars</td>
<td>String array that contains the user supplied derivative variable names. Contains those names passed directly to this variable as an array of strings as well as those added through the addState function. When set directly, all previously added derivativeVars are removed.</td>
<td>{}</td>
</tr>
<tr>
<td>diagnosticFileName</td>
<td>File name to which diagnostic output is written when requested by the user through the printDiagnostics flag.</td>
<td>&quot;cerr&quot;</td>
</tr>
<tr>
<td>generateCondition</td>
<td>expression that when it evaluates to TRUE allows a linear model to be generated when the execute function is called.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>inputVars</td>
<td>String array that holds the user supplied input variables names. Contains those names passed directly to this variable as an array of strings as well as those added through the addInput function. When set directly, all previously added inputVars are removed.</td>
<td>{}</td>
</tr>
<tr>
<td>linearityTolerance</td>
<td>The limit on the largest fractional difference between the same term in the positive and negative matrices. If the difference between any term exceeds this tolerance value, then the linear model as a whole fails the linearity test.</td>
<td>0.20</td>
</tr>
<tr>
<td>minNorm</td>
<td>The smallest value that the denominator is allowed to be when calculating the fractional variation between the positive and negative perturbation matrices. If a value is not explicitly set by the user, then a value is calculated as a function of the perturbation size. If the calculated value is less than 0.001 of the perturbation size, then the default value of 0.001 is used.</td>
<td>0.001</td>
</tr>
<tr>
<td>outputVars</td>
<td>String array that holds the user supplied input variables. Contains those names passed directly to this variable as an array of strings as well as those added through the addInput function. When set directly, all previously added outputVars are removed.</td>
<td>{}</td>
</tr>
<tr>
<td>printDiagnostics</td>
<td>Integer flag that controls printing of supplemental diagnostic output. Diagnostics are sent to the file specified in diagnosticFileName.</td>
<td>FALSE</td>
</tr>
<tr>
<td>reportFileName</td>
<td>File name to which the linear model is written.</td>
<td>&quot;cout&quot;</td>
</tr>
<tr>
<td>repeatabilityCheck</td>
<td>Integer flag that indicates whether a repeatability check should be performed or not. If FALSE, then the second set of positive perturbation matrices are not generated.</td>
<td>FALSE</td>
</tr>
<tr>
<td>repeatTolerance</td>
<td>The maximum amount any term in the second positive perturbation may differ from the same term in the initial positive perturbation matrices as a fraction of the average</td>
<td>0.001</td>
</tr>
</tbody>
</table>
### 8.5 Output (Calculated) Attributes

Table 37. Linear Model Generator Output Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A 2-D real array containing the &quot;A&quot; matrix of the linear model. The size is stateVars x derivativeVars, and is thus always square. The contents will be either the negative or positive perturbation matrix or an average of the two depending on the value of calcMethod.</td>
</tr>
<tr>
<td>B</td>
<td>A 2-D real array containing the &quot;B&quot; matrix of the linear model. The size is inputVars x derivativesVars. The contents will be either the negative or positive perturbation matrix or an average of the two depending on the value of calcMethod.</td>
</tr>
<tr>
<td>C</td>
<td>A 2-D real array containing the &quot;C&quot; matrix of the linear model. The size is stateVars x outputVars. The contents will be either the negative or positive perturbation matrix or an average of the two depending on the value of calcMethod.</td>
</tr>
<tr>
<td>D</td>
<td>A 2-D real array containing the &quot;D&quot; matrix of the linear model. The size is inputVars x outputVars. The contents will be either the negative or positive perturbation matrix or an average of the two depending on the value of calcMethod.</td>
</tr>
<tr>
<td>Anegative</td>
<td>A 2-D real array containing the &quot;A&quot; resulting from negative perturbations.</td>
</tr>
<tr>
<td>Bnegative</td>
<td>A 2-D real array containing the &quot;B&quot; resulting from negative perturbations.</td>
</tr>
<tr>
<td>Cnegative</td>
<td>A 2-D real array containing the &quot;C&quot; resulting from negative perturbations.</td>
</tr>
<tr>
<td>Dnegative</td>
<td>A 2-D real array containing the &quot;D&quot; resulting from negative perturbations.</td>
</tr>
<tr>
<td>Apositive</td>
<td>A 2-D real array containing the &quot;A&quot; resulting from positive perturbations.</td>
</tr>
<tr>
<td>Bpositive</td>
<td>A 2-D real array containing the &quot;B&quot; resulting from positive perturbations.</td>
</tr>
<tr>
<td>Cpositive</td>
<td>A 2-D real array containing the &quot;C&quot; resulting from positive perturbations.</td>
</tr>
<tr>
<td>Dpositive</td>
<td>A 2-D real array containing the &quot;D&quot; resulting from positive perturbations.</td>
</tr>
<tr>
<td>Arepeat</td>
<td>A 2-D real array containing the &quot;A&quot; resulting from repeating the positive perturbations after the negative matrices are generated. Only created if the repeatabilityCheck flag is TRUE.</td>
</tr>
<tr>
<td>Brepeat</td>
<td>A 2-D real array containing the &quot;B&quot; resulting from repeating the positive perturbations after the negative matrices are generated. Only created if the repeatabilityCheck flag is TRUE.</td>
</tr>
<tr>
<td>Crepeat</td>
<td>A 2-D real array containing the &quot;C&quot; resulting from repeating the positive perturbations after the negative matrices are generated. Only created if the repeatabilityCheck flag is TRUE.</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>perturbations</td>
<td>after the negative matrices are generated. Only created if the</td>
</tr>
<tr>
<td>Drepeat</td>
<td>repeatabilityCheck flag is TRUE.</td>
</tr>
<tr>
<td>derivativeBase</td>
<td>Real array containing the baseline value of the derivative variables.</td>
</tr>
<tr>
<td>InputBase</td>
<td>Real array containing the baseline value of the input variables.</td>
</tr>
<tr>
<td>InputPerturbVec</td>
<td>A real array of the perturbation sizes for each input variable. A zero value</td>
</tr>
<tr>
<td>InputPerturbTypeVec</td>
<td>indicates that the value of defaultPerturbation will be used for the</td>
</tr>
<tr>
<td>InputBiasVec</td>
<td>corresponding input variable.</td>
</tr>
<tr>
<td>minNormSet</td>
<td>A flag that indicates if a minNorm value has been set through user input</td>
</tr>
<tr>
<td>maxNonlinearVal</td>
<td>The maximum fractional difference between the terms in the positive and</td>
</tr>
<tr>
<td>maxNonrepeatingVal</td>
<td>negative perturbation matrices.</td>
</tr>
<tr>
<td>nonLinearLocation</td>
<td>A string containing the matrix name and indices where the</td>
</tr>
<tr>
<td>nonRepeatingLocation</td>
<td>maxNonlinearVal occurred.</td>
</tr>
<tr>
<td>outputBase</td>
<td>Real array containing the baseline value of the output variables</td>
</tr>
<tr>
<td>passedLinearTest</td>
<td>Integer flag indicating if the linear model passed the linearity test.</td>
</tr>
<tr>
<td>PassedRepeatTest</td>
<td>Integer flag indicating if the linear model passed the repeatability test.</td>
</tr>
<tr>
<td>stateBase</td>
<td>Real array containing the baseline value of the derivative variables</td>
</tr>
</tbody>
</table>
9 Hierarchical Data Format (HDF) Guide

(Hierarchical Data Format version 1.6.0)
For additional information, please go to the following Website: http://hdf.ncsa.uiuc.edu/HDF5/

9.1 Overview

NPSS implements the HDF file format for binary data storage. This is a platform-independent binary file format, with data stored in a hierarchical structure. The initial implementation of HDF in NPSS provides the capability to create a file and store standard two-dimensional real arrays within the file. Each "logical" dataset will actually consist of three physical HDF datasets within the file. A two-dimensional real array will hold the data values, presumably the value of variables over time. Two one-dimensional datasets, of datatype variable-length string, will be created to contain the variable names and the variable descriptions. The correlation between the names, descriptions, and data values will be by index into the arrays. With each write of the dataset, the variables will be evaluated and the values stored in the dataset in the HDF file. With each read, one column of data is returned and the location in the dataset is marked. To retrieve the entire dataset, multiple reads must be done until the entire dataset is read. Each read returns a 1-dimensional real array, the size of which is determined by the number of variables in the dataset. Each dataset can be tested for end of data via a user-accessible function EOD(). Note that access to the name and description datasets is through the getVarNames() and getVarDescriptions() user-accessible functions which return a variable-length string array containing the information for all variables in the dataset. A pictorial view of this data follows, as an example:

<table>
<thead>
<tr>
<th>VarName</th>
<th>Time</th>
<th>DataValues</th>
<th>Variable Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100.01</td>
<td>100.02</td>
<td>100.03 100.04</td>
</tr>
<tr>
<td>Var1</td>
<td>34.89</td>
<td>72.7</td>
<td>1005.6 2</td>
</tr>
<tr>
<td>Var2</td>
<td>35.43</td>
<td>73.4</td>
<td>999.2 5</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
</tr>
</tbody>
</table>

Note that to the user, the creation of the names and description datasets is transparent. The user will only need to think about creating one dataset. The NPSS code will handle creating the associated datasets.

9.2 Syntax

Example 1: Writing and reading a simple file.
The following is an example of creating an HDF file and dataset, writing to the dataset, and reading from it. It includes examples of using variable aliases and wildcards, and the accessor functions for the variable name and descriptions datasets. The filename is defaulted to the HDFFile object + .h5 (myHDF.h5 in this example). The default mode is 0, which means replace any existing files by this name.

HDFFile myHDF; //create HDFFile object called 'myHDF'
// declare variables, assign their values and description attributes
real one { value = 96.0; description = "I am a 1"; }
real two { value = 21.4; description = "I am a 2"; }
real three { value = 39.23; description = "I am a 3"; }
real four  { value = 74.62; description = "I am a 4"; }
real five  { value = 55.55; description = "I am a 5"; }
real six   { value = 60.21; description = "I am a 6"; }

setAlias("three", "thr3"); // test aliasing

// add above variables to the string array 'vars' to be passed to
// 'addDataSet()'; note the use of wildcard in "f*"; note that "three"
string vars[] = {"one", "two", "thr3", "f*", "six"};

myHDF.addDataSet("my data", vars);
myHDF.write(""); // perform a write to ALL datasets in "myHDF"

myHDF.write("""); // perform a write to ALL datasets in "myHDF"

//retrieve data just written to "my data" in "myHDF.h5"
real vals[] = myHDF.read("my data");

for (i = 0; i < names.entries(); i++) {
    cout << names[i] << "  " << vals[i] << "  " << descriptions[i] << endl;
}

The previous input file creates the following output:

Dataset "my data":
one     96      I am a 1
two     21.4    I am a 2
thr3    39.23   I am a 3
six     60.21   I am a 4
four    74.62   I am a 5
five    55.55   I am a 6

For the logical dataset my data, three physical datasets are actually created. My data will contain the real values of the variables. My data_Variables (a dataset that is hidden to the user) will contain the variable names. My data_Descriptions (also hidden to the user) will contain the variable descriptions. The user need not know about these additional physical datasets, as the NPSS code handles creating and accessing them. Access to the information in these datasets is through the user-accessible functions getVarNames() and getVarDescriptions(). Each file can have a virtually infinite number of datasets, but the dataset names must be unique.

Note that the addDataSet function supports NPSS wildcards and aliasing in the list of variables to be added to the system. Note that variable three is known as thr3 (its alias name) by the HDF dataset. Be careful when using wildcards as you might pick up native NPSS model variables that you are not expecting.

Example 2: Opening the existing file for reading and appending.
This example uses the file created in example 1. A new HDFFile object is instantiated and the existing file is opened with mode = 1 (for append). The existing file information is printed out. The variables (which are expected to already exist in the current model) are updated and written to the file. The dataset is rewound so that the entire dataset can be read from the beginning. A user-function reads each record and prints it out, checking for the end-of-data condition to avoid reading past the end of the dataset. The values for the entire dataset are then printed again, showing the original data plus the appended data.

//first define a function to read all data from a dataset
void retrieveData(string ds) {
    int i = 0;
    while (!myHDF2.atEOD(ds)) {
        cout << names[i] << "  " << vals[i] << "  " << descriptions[i] << endl;
    }
}
cout << "\t Values for time = " << i++ << ":\n" << myHDF2.read(ds) << endl;
} cout << "End of data reached for '" << ds << "'!\n\n";
}

//declare variables that were saved
real one;
real two;
real thr3;
real four;
real five;
real six;

HDFFile myHDF2; //create HDFFile object called "myHDF2"
myHDF2.openHDFFile("myHDF.h5", 1); //open existing file "myHDF.h5" for append

// Query the dataset first
//retrieve list of dataset names from "myHDF.h5"
string datasets[] = myHDF2.getDSnames();
int i, j;

for (i = 0; i < datasets.entries(); i++) {
    //loop through all existing datasets (only one in this example)
    cout << "\n Dataset '" << datasets[i] << "' in file '" << myHDF2.getFilename() << "':\n";
    //retrieve variable names from "my data"
    string names[] = myHDF2.getVarNames("my data");
    cout << "\t Variable Names:\n" << names << endl;
    //print out variable descriptions from variables in "my data"
    cout << "\t Variable Descriptions:\n" << myHDF2.getVarDescriptions(datasets[i]) << endl;
    //print out current dataset
    retrieveData(datasets[i]);
    //now append to dataset;
    real base = 100;
    for (j= 0; j < names.entries(); j++) {
        names[j]->value = base+j;
        cout << "Setting " << names[j] << " to " << base+j << endl;
    }
    //now write new values
    cout << "\n\nUpdating data values and writing out to file...\n";
    myHDF2.write(datasets[i]);
    cout << "Rewinding dataset to read from beginning of file...\n";
    myHDF2.rewindDataSet(datasets[i]);
    cout << "Updated values for dataset '" << datasets[i] << "':\n";
    retrieveData(datasets[i]);
}

The previous input file creates the following output:

**EXAMPLE 2 RESULTS:**

Dataset 'my data' in file 'myHDF.h5':
Variable Names:
{"one",
"two",
"thr3",
"six",
"four",
"five" }

Variable Descriptions:
{"I am a 1",
"I am a 2",
"I am a 3",
"I am a 6",
"I am a 4",
"I am a 5" }

Values for time = 0:
{ 96, 21.4, 39.23, 60.21, 74.62, 55.55 }
End of data reached for 'my data'!

Setting one to 100
Setting two to 101
Setting thr3 to 102
Setting six to 103
Setting four to 104
Setting five to 105

Updating data values and writing out to file...
Rewinding dataset to read from beginning of file...
Updated values for dataset 'my data':
  Values for time = 0:
  { 96, 21.4, 39.23, 60.21, 74.62, 55.55 }
  Values for time = 1:
  { 100, 101, 102, 103, 104, 105 }
End of data reached for 'my data'!

9.3 User-accessible Functions

HDFFile hdfObject;
Creates an empty HDFFile object. No physical file will be created until either the addDataSet() or
openHDFFile() functions are called. Note this is NOT a generic reader for HDF files; the structure imposed by
NPSS is expected and is there to relieve the user from having to implement the details.

void hdfObject.openHDFFile(string filename, int mode);
Attempts to open a physical file called filename as an HDF5 file. Mode indicates how the user would like to handle
existing files. Mode = 0 tells the system to delete any existing file by this name and create a new one. Mode = 1
tells the system to read in an existing file and append to any existing data. If no file exists with this name, one will
be created. The user can then add new datasets to this file or add to the existing datasets. This call is unnecessary if
you use the default parameter values, which are dsname = HDFFile object name +.h5 and mode = 0. If this
function is not called, the first time addDataSet() is called, the default values for these parameters are invoked
and the new file is created and the dataset added.

string hdfObject.getFilename();
Returns the name of the physical file currently associated with this HDFFile object. Only one physical file can be
associated with an HDFFile object for the life of the object.
  Example:
    cout << “The HDF file name is “ << myHDF.getFilename() << endl;
void hdfObject.addDataSet(string dsname, string[] varnames);
Creates a dataset in the file called dsname which will contain the real data values for the variables in the varNames list. This dsname MUST be unique in the file or an error message will be produced. The variables contained in varnames will be checked for correct datatype, and if found to be of an invalid type or nonexistent, will NOT be added to the dataset and a warning message will be issued.

void hdfObject.write(string dsname);
Causes the current real data values of all variables in dsname to be written to the HDFFile filename. If a null string ("") is supplied, ALL datasets in the file will be written. A write will reset the end-of-data flag so that additional reads can be performed on the new data.

Real[] hdfObject.read(string dsname);
Returns a 1-D real array consisting of one write of all variables to the named dataset. The first read will read the first set of values written; consecutive reads step through file, i.e., if there are 10 variables in the dataset, a read will return a 10-element real array. If a read is done past the end of data, an error message will be returned. The isEOD() function allows the user to test this condition before performing a read.

int hdfObject.atEOD(string dsname);
Returns 0 if the dataset contains more data to be read; 1 if the end of the dataset has been reached. To reset the read pointers to read from the beginning of the dataset, use rewindDataSet(). This function operates on a single dataset at a time.

void hdfObject.rewindDataSet(string dsname);
Resets the read pointers to allow reading from the beginning of the dataset. Can be called at any point in reading the data. A null string parameter value ("") will reset ALL datasets. Alternatively, a single dsname can be provided to operate on a single dataset.

string[] hdfObject.getDSnames();
Returns a list of the names of all datasets in the file.

string[] hdfObject.getVarNames(string dsname);
Returns a list of the variable names in dsname. If alias names were provided for any variable, those names will be returned.

string[] hdfObject.getVarDescriptions(string dsname);
Returns a list of the variable descriptions for the variables in dsnames.
Part 2: Reference
10 Assembly Attributes

An assembly is a group of elements encapsulated and managed for users. An assembly has the following attributes associated with it.

Table 38. Assembly Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>presolverSequence</td>
<td>A list of the names of the elements and/or functions that are executed prior to the solver iteration loop. This is an alias to the Assembly’s current Executive’s preExecutionSequence.</td>
<td>{   }</td>
</tr>
<tr>
<td>postsolverSequence</td>
<td>A list of the names of the elements and/or functions that are executed after the solver iteration loop. This is an alias to the Assembly’s current Executive’s postExecutionSequence.</td>
<td>{   }</td>
</tr>
<tr>
<td>solverSequence</td>
<td>A list of the names of the elements and/or functions that are executed in the solver iteration loop. This is an alias to the Assembly’s current Executive’s executionSequence.</td>
<td>{   }</td>
</tr>
<tr>
<td>steadyStateExecutiveName</td>
<td>A string that holds the name of the Executive that should be set as the current Executive of this Assembly if the solutionMode OptionVariable is set to “STEADY_STATE” or “ONE_PASS.”</td>
<td>“solver”</td>
</tr>
<tr>
<td>transientExecutiveName</td>
<td>A string that holds the name of the Executive that should be set as the current Executive of this Assembly if the solutionMode OptionVariable is set to “TRANSIENT.”</td>
<td>“transient”</td>
</tr>
<tr>
<td>switchDes</td>
<td>Design mode switch indicator – equals either DESIGN or OFFDESIGN</td>
<td>DESIGN</td>
</tr>
</tbody>
</table>
11 Units

Units track the quantity of a real value. They keep the values consistent no matter which system is used to enter them. Rather than being based on a giant lookup table, units are built on primitive base measures, which can be constructed into complex units.

The following is an example showing how automatic conversion takes place.

```c
real x = 10;
x.units = "mile";
cout << "x = " << x << " " << x.units << endl;
x = 10 mile
```

Since the unit of the variable `x` is mile, an assignment in feet will be converted to mile:

```c
x = 6000 "ft";
cout << "x = " << x << " " << x.units << endl;
x = 1.13636 mile
```

In the units functions that follow, "left" and "right" are aliases for English and Metric. "Left" and "right" are used because they are more generic. There could be other unit systems; the units code was designed to allow a nearly complete redefinition of unit systems. When adding a new unit, use the `addUnitRelation(...)` mentioned in the following section. To change from English to Metric units, for every variable used specify its unit string with its equivalent metric unit.

11.1 Units Functions

The following functions are used to display, set and manipulate the units:

- `displayConvTables()`
  - This function call outputs a string stream that contains the base units table, the unit relations table, and the names given to the left and right unit systems. If the user extends or resets the unit tables as described as follows, this function call will output the current tables.

- `resetConvTable(string leftSystem, string rightSystem)`
  - If users want to have their own set of units, they have to destroy the old conversion table. To do that, they need to use this function. Pass in the two strings that are the names of the new conversion unit systems. For example:
    ```c
    resetConvTable("US", "my_system");
    ```
  - The purpose of the strings is to label the left and right unit systems in the new table. By default these strings are "US" and "SI". This is needed if the variable `unitSystem` is set in the Data Viewer definition statements.

- `addUnitBase (string leftUnit, string rightUnit, real conversionFactor, string type)`
  - Unit bases are the primitives, where the actual conversions take place. All other units are complex units and will be broken down into unit bases. A conversion factor is needed to translate between two primitive base units. The “Q_” in the function arguments such as `Q_LENGTH` stands for “Quantity”. For example:
    ```c
    addUnitBase("ft","m", .3048, Q_LENGTH);
    addUnitBase("sec","sec", 1, Q_TIME);
    ```

There are only 9 primitive base units:

```c
addUnitBase("ft", "m", 0.3048, Q_LENGTH);
addUnitBase("sec", "sec", 1, Q_TIME);
```
addUnitBase("A", "A", 1, Q_CURRENT);
addUnitBase("SQRT_R", "SQRT_K", 0.745355992499930, Q_TEMP/2);
addUnitBase("slug", "kg", 14.5939, Q_MASS);
addUnitBase("mol", "mol", 1, Q_SUBSTANCE);
addUnitBase("rad", "rad", 1, Q_ANGLE);
addUnitBase("SQRT_lbf", "Q_SQRT_N", 2.1090807476244, Q_SQRT_F);
addUnitBase("cd", "cd", 1, Q_LIGHT);

NOTE: Q_TEMP is divided by two because type Units::TEMP is the two factors squared.

NOTE: The addUnitBase(...) method should seldom be used, since this deals with unit primitives.

Here both feet ("ft") and meters ("m") are added as length units, with a conversion factor of .3048 between left and right units. Sec, for second, is unit system neutral but still must be defined. Argument 4 uses Q_LENGTH, Q_TIME, Q_SQRT_F, Q_MASS, Q_ANGLE, Q_CURRENT, Q_LIGHT and Q_SUBSTANCE to differentiate between the different types of units of measure. Q_SQRT_T is for square root temperature as in SQRT_R. Two base unit entries are put in by default. They are "none" of type NONE and "+" of type WITHOFFSET. The type "none" is needed for variables that have no units. The type "+" is used so temperature conversions can be made with offsets when they are not mixed with other units. Compound unit entries, such as "1000*m", are not permitted in this function call.

After you have entered all the primitives and their conversions, use the next function to enter the unit relations.

addUnitRelation(string leftUnitName, string leftUnitRelation, string correspondingRightUnitName, string correspondingRightUnitRelation)

This function may be used to add a new unit. Some examples are:

addUnitRelation("mile", "5280.0*ft", "km", "1000*m");
addUnitRelation("ft2", "ft*ft", "m2", "m*m");

Note that ordering is important when entering these statements. NPSS must be given the definition of each item in the 2nd and 4th arguments before hand. For example, this statement

addUnitRelation("knot", "nmile/hr", "km/hr", "km/hr");

will cause errors unless the following are entered first:

addUnitRelation("nmile", "6076.115*ft", "km", "1000*m");
addUnitRelation("hr", "3600*sec", "hr", "3600*sec");

The default unit relations can also be expanded using the above call. Add a Unit Relation when you want to use a new, previously undefined unit of measure. All Unit Relations must be expressed in unit bases or previously defined relations. If, for example, you wish to use yards and furlongs, you must create Unit Relations as follows:

addUnitRelation("yd", "3*ft", "m", "m");
addUnitRelation("furlong", "500*yd", "km", "1000*m");

Note that the 3rd and 4th arguments are the closest units of measure in the rightSystem; if there is a numerical constant, it needs to be on the left side of the multiplication sign. The 2nd and 4th argument's types are checked to make sure they are the same type of quantity (i.e., length, velocity, etc.). Above, "furlong" and "kilometer" are simply aliases for the actual relations. Checks are done to make sure that the relations contain primitives from their proper systems.

A plus sign ("+") is used as a unit to tag other units. By default it means "with offset" and is used with the temperature relation:

addUnitRelation("F", "R*+", "C", "K*+");

This allows stand-alone temperature conversions with offset to be properly handled.

Other unit offset relations can be defined by specifying a plus(+) or minus(-) offset, followed by an offset number. For example, the unit relation of “psig” to “psia” can be specified as psig = psia-14.7 as shown below:
// create new unit "psig" and relate it to "psia"
addUnitRelation("psig", "psia-14.7", "Pa", "Pa");

// output should be 14.7 - 14.7 = 0
real P1 { units="psig"; }
P1 = 14.7 "psia";
cout << "\n 14.7 psia converts to " << P1 << " " << P1.units << endl;

In this case, the output would be:

14.7 psia converts to 0 psig

addUnitPref (string leftUnit, string preferredRightUnit)
The user has the option to specify preferred units for various quantities in the output conversions as a result of using the Data Viewer UnitSystem call. For example:

    addUnitPref("ft", "km");

This function is used to specify whether a user prefers feet to be converted to kilometers instead of meters, for instance. Unit Prefs are kept in a table, and all UnitPrefs are destroyed with a call to resetConvTable.

real getUnitAdder(string inUnit, string outUnit)
Returns the adder required to convert from inUnit to outUnit.

real getUnitsFactor(string inUnit, string outUnit)
Returns the factor required to convert from inUnit to outUnit. An error is returned if either the inUnit or outUnit is "F", "C", "K" or "K". The error message indicates that "dF", "dC", "dR", or "dK" is acceptable.

For example:

    cout << toStr(getUnitsFactor("BTU","kJ"), 8);

Returns:

    1.0550553

Whereas:

    cout << toStr(getUnitsFactor("dR","C"), 8);

Returns:

    ERROR(91041001) in MemberFunction 'getUnitsFactor': In Unit Conversion: Unit 'PleaseUse-dC' is invalid

Because "C" should have been entered as "dC".

void convertUnits(string inputValue, string convertToUnits)
Accepts inputValue that has a value and units, calculates and returns a new value based on the conversion units input in parameter convertToUnits.

Examples:

    real inp { value=1.0; units="ft"; }
    real C_FTtoM { value=convertUnits("inp","m"); IOstatus="const"; }
    real C_FTtoMiles = convertUnits("inp","mile");

Would set:

    C_FTtoM    = 0.3048 meters
    C_FTtoMiles = 0.000189394 miles
real getUnitScaler(string inUnit, string outUnit)
Returns the scaler required to convert from inUnit to outUnit.

11.2 Valid Unit Strings
The NPSS accepts the following unit strings. Numerous other combinations are also accepted, but mixing unit primitives from left and right systems in the same unit string will generate a warning. Nonsensical units may be assigned to variables for documentation purposes only, but the variable's unitsRule attribute must first be set to "DISPLAY". Attempted conversion with nonsensical units will generate an error.

<table>
<thead>
<tr>
<th>Unit String</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;1/sec&quot;</td>
<td>per second</td>
</tr>
<tr>
<td>&quot;1/rpm&quot;</td>
<td>per revolutions per minute</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>Ampere</td>
</tr>
<tr>
<td>&quot;atm&quot;</td>
<td>atmosphere</td>
</tr>
<tr>
<td>&quot;bar&quot;</td>
<td>bar</td>
</tr>
<tr>
<td>&quot;Btu&quot;</td>
<td>Btu</td>
</tr>
<tr>
<td>&quot;Btu/hr&quot;</td>
<td>Btu per hour</td>
</tr>
<tr>
<td>&quot;Btu/(hr<em>ft</em>F)&quot;</td>
<td>Btu per hour per foot per degrees Fahrenheit</td>
</tr>
<tr>
<td>&quot;Btu/(hr*ft2)&quot;</td>
<td>Btu per hour per foot squared</td>
</tr>
<tr>
<td>&quot;Btu/(hr*ft3)&quot;</td>
<td>Btu per hour per foot cubed</td>
</tr>
<tr>
<td>&quot;Btu/lbm&quot;</td>
<td>Btu per pounds mass</td>
</tr>
<tr>
<td>&quot;Btu/(lbfm*sec)&quot;</td>
<td>Btu per pound mass per second</td>
</tr>
<tr>
<td>&quot;Btu/(lbfm*R)&quot;</td>
<td>Btu per pounds mass per degree Rankine</td>
</tr>
<tr>
<td>&quot;Btu/(lbfm<em>R</em>R)&quot;</td>
<td>Btu per pounds mass per degree Rankine squared</td>
</tr>
<tr>
<td>&quot;Btu/R&quot;</td>
<td>Btu per degrees Rankine</td>
</tr>
<tr>
<td>&quot;Btu/(R*in)&quot;</td>
<td>Btu per Rankine per inch</td>
</tr>
<tr>
<td>&quot;Btu/sec&quot;</td>
<td>Btu per second</td>
</tr>
<tr>
<td>&quot;Btu/(sec<em>ft</em>R)&quot;</td>
<td>Btu per second per foot per Rankine</td>
</tr>
<tr>
<td>&quot;Btu/(sec*ft2)&quot;</td>
<td>Btu per second feet Rankine</td>
</tr>
<tr>
<td>&quot;Btu/(sec*ft2)&quot;</td>
<td>Btu per second per square foot</td>
</tr>
<tr>
<td>&quot;Btu/(sec<em>in</em>R)&quot;</td>
<td>Btu per second inches Rankine</td>
</tr>
<tr>
<td>&quot;Btu/(sec<em>in2</em>R)&quot;</td>
<td>Btu per second inches squared Rankine</td>
</tr>
<tr>
<td>&quot;Btu/(sec*R)&quot;</td>
<td>Btu per second per Rankine</td>
</tr>
<tr>
<td>&quot;Btu<em>sec2/(lbfm</em>ft2)&quot;</td>
<td>Btu seconds squared per pound mass foot squared</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>degrees Celsius</td>
</tr>
<tr>
<td>&quot;cd&quot;</td>
<td>candela</td>
</tr>
<tr>
<td>&quot;cm&quot;</td>
<td>centimeter</td>
</tr>
<tr>
<td>&quot;cm2&quot;</td>
<td>square centimeter</td>
</tr>
<tr>
<td>&quot;cm3&quot;</td>
<td>cubic centimeter</td>
</tr>
<tr>
<td>&quot;cm4&quot;</td>
<td>centimeters to the fourth</td>
</tr>
<tr>
<td>&quot;cm5&quot;</td>
<td>centimeters to the fifth</td>
</tr>
<tr>
<td>&quot;Coul&quot;</td>
<td>coulomb</td>
</tr>
<tr>
<td>&quot;daN&quot;</td>
<td>dekanewton</td>
</tr>
<tr>
<td>&quot;day&quot;</td>
<td>day</td>
</tr>
<tr>
<td>&quot;deg&quot;</td>
<td>degrees</td>
</tr>
<tr>
<td>&quot;dC&quot;</td>
<td>Delta degrees Celsius (no offset on conversion)</td>
</tr>
<tr>
<td>&quot;dF&quot;</td>
<td>Delta degrees Fahrenheit (no offset on conversion)</td>
</tr>
<tr>
<td>&quot;dK&quot;</td>
<td>Delta degrees Kelvin (no offset on conversion)</td>
</tr>
<tr>
<td>&quot;dR&quot;</td>
<td>Delta degrees Rankin (no offset on conversion)</td>
</tr>
<tr>
<td>&quot;F&quot;</td>
<td>degrees Fahrenheit</td>
</tr>
<tr>
<td>Unit String</td>
<td>Definition</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>&quot;FD&quot;</td>
<td>farad</td>
</tr>
<tr>
<td>&quot;ft&quot;</td>
<td>feet</td>
</tr>
<tr>
<td>&quot;ft*lbf&quot;</td>
<td>foot-pounds force</td>
</tr>
<tr>
<td>&quot;ft*lbf/Btu&quot;</td>
<td>foot-pounds force per BTU</td>
</tr>
<tr>
<td>&quot;ft<em>lbf/(R</em>lbm)&quot;</td>
<td>foot-pounds force per degrees Rankine per pounds mass</td>
</tr>
<tr>
<td>&quot;ft<em>lbf/(R</em>lbmol)&quot;</td>
<td>foot-pounds force per degrees Rankine per pound mole</td>
</tr>
<tr>
<td>&quot;ft*lbf/rad&quot;</td>
<td>foot-pounds force per radian</td>
</tr>
<tr>
<td>&quot;ft<em>lbf/(rad</em>sec)&quot;</td>
<td>foot-pounds force per radians second</td>
</tr>
<tr>
<td>&quot;ft*lbf/sec&quot;</td>
<td>foot-pounds force per second</td>
</tr>
<tr>
<td>&quot;ft<em>lbf</em>in3/(lbm<em>rad</em>rad)&quot;</td>
<td>foot-pounds force cubic inch per pound mass per radian squared</td>
</tr>
<tr>
<td>&quot;ft<em>lbf/(lbm</em>sec2)&quot;</td>
<td>foot-pounds mass per foot-pounds force per second squared</td>
</tr>
<tr>
<td>&quot;ft/sec&quot;</td>
<td>feet per second</td>
</tr>
<tr>
<td>&quot;ft/sec2&quot;</td>
<td>feet per second squared</td>
</tr>
<tr>
<td>&quot;ft2&quot;</td>
<td>square feet</td>
</tr>
<tr>
<td>&quot;ft2*lbm&quot;</td>
<td>square feet pound mass</td>
</tr>
<tr>
<td>&quot;ft2/sec2&quot;</td>
<td>square feet per second squared</td>
</tr>
<tr>
<td>&quot;ft2/(sec2*R)&quot;</td>
<td>square feet per second squared per degrees Rankine</td>
</tr>
<tr>
<td>&quot;ft3&quot;</td>
<td>cubic foot</td>
</tr>
<tr>
<td>&quot;ft3/lbm&quot;</td>
<td>cubic foot per pounds mass</td>
</tr>
<tr>
<td>&quot;g/sec&quot;</td>
<td>grams per second</td>
</tr>
<tr>
<td>&quot;gal&quot;</td>
<td>gallon</td>
</tr>
<tr>
<td>&quot;gal<em>in/(min</em>SQR2_lbf)&quot;</td>
<td>gallon inch per minute per square root pound force</td>
</tr>
<tr>
<td>&quot;grains&quot;</td>
<td>grains</td>
</tr>
<tr>
<td>&quot;H&quot;</td>
<td>henry</td>
</tr>
<tr>
<td>&quot;hbar&quot;</td>
<td>hectobar</td>
</tr>
<tr>
<td>&quot;hp&quot;</td>
<td>horsepower</td>
</tr>
<tr>
<td>&quot;hp/ft2&quot;</td>
<td>horsepower per square foot</td>
</tr>
<tr>
<td>&quot;hp*h&quot;</td>
<td>horsepower hour</td>
</tr>
<tr>
<td>&quot;hr&quot;</td>
<td>hours</td>
</tr>
<tr>
<td>&quot;Hz&quot;</td>
<td>hertz</td>
</tr>
<tr>
<td>&quot;in&quot;</td>
<td>inches</td>
</tr>
<tr>
<td>&quot;in2&quot;</td>
<td>square inches</td>
</tr>
<tr>
<td>&quot;in3&quot;</td>
<td>cubic inch</td>
</tr>
<tr>
<td>&quot;in2*in2&quot;</td>
<td>inches to the fourth</td>
</tr>
<tr>
<td>&quot;in2<em>in2</em>sec2/(lbm*ft)&quot;</td>
<td>inches to the fourth seconds squared per pound mass foot</td>
</tr>
<tr>
<td>&quot;in2*in3&quot;</td>
<td>inches to the fifth</td>
</tr>
<tr>
<td>&quot;in4&quot;</td>
<td>inches to the fourth</td>
</tr>
<tr>
<td>&quot;in4<em>sec2/(lbm</em>ft)&quot;</td>
<td>inches to the fourth squared per pound mass per foot</td>
</tr>
<tr>
<td>&quot;in5&quot;</td>
<td>inches to the fifth</td>
</tr>
<tr>
<td>&quot;in1<em>lb</em>sec2/lbm&quot;</td>
<td>inch pound force second squared per pound mass</td>
</tr>
<tr>
<td>&quot;inH2O&quot;</td>
<td>inch of water</td>
</tr>
<tr>
<td>&quot;inHg&quot;</td>
<td>inch of mercury</td>
</tr>
<tr>
<td>&quot;in/sec&quot;</td>
<td>inch per second</td>
</tr>
<tr>
<td>&quot;in3*sec/rad&quot;</td>
<td>cubic inch second per radian</td>
</tr>
<tr>
<td>&quot;J&quot;</td>
<td>Joule</td>
</tr>
<tr>
<td>&quot;J/kg&quot;</td>
<td>Joules per kilogram</td>
</tr>
<tr>
<td>&quot;J/(kg*K)&quot;</td>
<td>Joules per kilogram per degrees Kelvin</td>
</tr>
<tr>
<td>&quot;J/sec&quot;</td>
<td>Joule per second</td>
</tr>
<tr>
<td>&quot;k&quot;</td>
<td>degrees Kelvin</td>
</tr>
<tr>
<td>&quot;kg&quot;</td>
<td>kilograms</td>
</tr>
<tr>
<td>Unit String</td>
<td>Definition</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>&quot;kg/J&quot;</td>
<td>kilograms per Joule</td>
</tr>
<tr>
<td>&quot;kg*m/sec2&quot;</td>
<td>kilogram-meters per second squared</td>
</tr>
<tr>
<td>&quot;kg*m2&quot;</td>
<td>kilogram-square meters</td>
</tr>
<tr>
<td>&quot;kg/m3&quot;</td>
<td>kilograms per cubic meter</td>
</tr>
<tr>
<td>&quot;kg/(N*sec)&quot;</td>
<td>kilograms per Newton per second</td>
</tr>
<tr>
<td>&quot;kg/sec&quot;</td>
<td>kilogram per second</td>
</tr>
<tr>
<td>&quot;kA&quot;</td>
<td>kilo ampere</td>
</tr>
<tr>
<td>&quot;kCoul&quot;</td>
<td>kilo coulomb</td>
</tr>
<tr>
<td>&quot;kH&quot;</td>
<td>kilo henry</td>
</tr>
<tr>
<td>&quot;kOhm&quot;</td>
<td>kilo ohm</td>
</tr>
<tr>
<td>&quot;kS&quot;</td>
<td>kilo siemens</td>
</tr>
<tr>
<td>&quot;kV&quot;</td>
<td>kilo volt</td>
</tr>
<tr>
<td>&quot;kW&quot;</td>
<td>kilo watts</td>
</tr>
<tr>
<td>&quot;kWB&quot;</td>
<td>kilo weber</td>
</tr>
<tr>
<td>&quot;kJ/kg&quot;</td>
<td>kilojoule per kilogram</td>
</tr>
<tr>
<td>&quot;kJ/(kg*K)&quot;</td>
<td>kilojoules per kilogram per Kelvin</td>
</tr>
<tr>
<td>&quot;km&quot;</td>
<td>kilometer</td>
</tr>
<tr>
<td>&quot;kN&quot;</td>
<td>kilonewton</td>
</tr>
<tr>
<td>&quot;knot&quot;</td>
<td>nautical miles per hour</td>
</tr>
<tr>
<td>&quot;kPa&quot;</td>
<td>kilopascals</td>
</tr>
<tr>
<td>&quot;kW&quot;</td>
<td>kilowatts</td>
</tr>
<tr>
<td>&quot;lbf&quot;</td>
<td>pounds force</td>
</tr>
<tr>
<td>&quot;lbf*ft2&quot;</td>
<td>pound force per square foot</td>
</tr>
<tr>
<td>&quot;lbf<em>in</em>sec2/lbm&quot;</td>
<td>pound force inch second squared per pound mass</td>
</tr>
<tr>
<td>&quot;lbf*in2&quot;</td>
<td>pound force square inch</td>
</tr>
<tr>
<td>&quot;lbf<em>in4</em>sec2/lbm&quot;</td>
<td>pound force inches to the fourth second squared per pound mass</td>
</tr>
<tr>
<td>&quot;lbf<em>sec/(lbm</em>in2)&quot;</td>
<td>pound force second per pound mass per inch</td>
</tr>
<tr>
<td>&quot;lbf<em>sec2/(lbm</em>in5)&quot;</td>
<td>pound force second squared per pound mass per inches to the fifth</td>
</tr>
<tr>
<td>&quot;lbf<em>sec2/(lbm</em>in2)&quot;</td>
<td>pound force second squared per pound mass per square inch</td>
</tr>
<tr>
<td>&quot;lbm&quot;</td>
<td>pounds mass</td>
</tr>
<tr>
<td>&quot;lbm/(ft*sec)&quot;</td>
<td>pound mass per foot per second</td>
</tr>
<tr>
<td>&quot;lbm/(ft*sec2)&quot;</td>
<td>pounds mass per foot per second squared</td>
</tr>
<tr>
<td>&quot;lbm/ft3&quot;</td>
<td>pound mass per cubic foot</td>
</tr>
<tr>
<td>&quot;lbm/ft3*sec&quot;</td>
<td>pound mass per cubic foot per second</td>
</tr>
<tr>
<td>&quot;lbm/lbm&quot;</td>
<td>pounds mass per pounds mass</td>
</tr>
<tr>
<td>&quot;lbm/hr&quot;</td>
<td>pound mass per hour</td>
</tr>
<tr>
<td>&quot;lbm/(hr*hp)&quot;</td>
<td>pounds mass per hour per horsepower</td>
</tr>
<tr>
<td>&quot;lbm/(hr*lbf)&quot;</td>
<td>pounds mass per hour per pounds force</td>
</tr>
<tr>
<td>&quot;lbm/(in*sec)&quot;</td>
<td>pounds mass per inch per second</td>
</tr>
<tr>
<td>&quot;lbm/in2&quot;</td>
<td>pounds mass per inch squared</td>
</tr>
<tr>
<td>&quot;lbm/in3&quot;</td>
<td>pounds mass per cubic inch</td>
</tr>
<tr>
<td>&quot;lbm/(in3*sec)&quot;</td>
<td>pounds mass per cubic inch per second</td>
</tr>
<tr>
<td>&quot;lbm/mol&quot;</td>
<td>pounds mass per mole</td>
</tr>
<tr>
<td>&quot;lbm/sec&quot;</td>
<td>pounds mass per second</td>
</tr>
<tr>
<td>&quot;lbm/(sec*ft2)&quot;</td>
<td>pounds mass per second per feet squared</td>
</tr>
<tr>
<td>&quot;lbm/(sec*in2)&quot;</td>
<td>pounds mass per second per inches squared</td>
</tr>
<tr>
<td>&quot;lbm/sec2&quot;</td>
<td>pounds mass per second squared</td>
</tr>
<tr>
<td>&quot;lbm*in2&quot;</td>
<td>pounds mass-square inches</td>
</tr>
<tr>
<td>&quot;lbm*lbm/(sec2)&quot;</td>
<td>pound mass squared per second squared</td>
</tr>
<tr>
<td>&quot;lbm<em>lbf</em>in2/(lbf*sec2)&quot;</td>
<td>pound mass squared square inch per pound force per second squared</td>
</tr>
<tr>
<td>&quot;lbm*R/sec&quot;</td>
<td>pound mass Rankine per second</td>
</tr>
<tr>
<td>Unit String</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>&quot;1bm*SQRT_R/psia&quot;</td>
<td>pounds mass square root degrees Rankine per pounds per square inch</td>
</tr>
<tr>
<td>&quot;1bmol&quot;</td>
<td>pound mole</td>
</tr>
<tr>
<td>&quot;liter&quot;</td>
<td>liters</td>
</tr>
<tr>
<td>&quot;m&quot;</td>
<td>meters</td>
</tr>
<tr>
<td>&quot;mA&quot;</td>
<td>milli ampere</td>
</tr>
<tr>
<td>&quot;mCoul&quot;</td>
<td>milli coulomb</td>
</tr>
<tr>
<td>&quot;mFD&quot;</td>
<td>milli farad</td>
</tr>
<tr>
<td>&quot;mH&quot;</td>
<td>milli henry</td>
</tr>
<tr>
<td>&quot;mOhm&quot;</td>
<td>milli ohm</td>
</tr>
<tr>
<td>&quot;mS&quot;</td>
<td>milli siemens</td>
</tr>
<tr>
<td>&quot;mV&quot;</td>
<td>milli volt</td>
</tr>
<tr>
<td>&quot;mW&quot;</td>
<td>milli watts</td>
</tr>
<tr>
<td>&quot;mWb&quot;</td>
<td>milli weber</td>
</tr>
<tr>
<td>&quot;mbar&quot;</td>
<td>milli bar</td>
</tr>
<tr>
<td>&quot;msec&quot;</td>
<td>milli second</td>
</tr>
<tr>
<td>&quot;MA&quot;</td>
<td>mega ampere</td>
</tr>
<tr>
<td>&quot;MCoul&quot;</td>
<td>mega coulomb</td>
</tr>
<tr>
<td>&quot;MH&quot;</td>
<td>mega henry</td>
</tr>
<tr>
<td>&quot;MOhm&quot;</td>
<td>mega ohm</td>
</tr>
<tr>
<td>&quot;MS&quot;</td>
<td>mega siemens</td>
</tr>
<tr>
<td>&quot;MV&quot;</td>
<td>mega volt</td>
</tr>
<tr>
<td>&quot;MW&quot;</td>
<td>mega watts</td>
</tr>
<tr>
<td>&quot;MWb&quot;</td>
<td>mega weber</td>
</tr>
<tr>
<td>&quot;mm4&quot;</td>
<td>millimeters to the fourth</td>
</tr>
<tr>
<td>&quot;mm5&quot;</td>
<td>millimeters to the fifth</td>
</tr>
<tr>
<td>&quot;m/sec&quot;</td>
<td>meters per second</td>
</tr>
<tr>
<td>&quot;m/sec2&quot;</td>
<td>meters per second squared</td>
</tr>
<tr>
<td>&quot;m2&quot;</td>
<td>square meters</td>
</tr>
<tr>
<td>&quot;m3&quot;</td>
<td>cubic meters</td>
</tr>
<tr>
<td>&quot;mg/J&quot;</td>
<td>milligrams per joule</td>
</tr>
<tr>
<td>&quot;mg/(N*sec)&quot;</td>
<td>milligrams per newton per second</td>
</tr>
<tr>
<td>&quot;mil&quot;</td>
<td>1/1000 inch (0.001*in)</td>
</tr>
<tr>
<td>&quot;mile&quot;</td>
<td>mile (5280 ft)</td>
</tr>
<tr>
<td>&quot;min&quot;</td>
<td>minute</td>
</tr>
<tr>
<td>&quot;mm&quot;</td>
<td>millimeters</td>
</tr>
<tr>
<td>&quot;mm2&quot;</td>
<td>square millimeters</td>
</tr>
<tr>
<td>&quot;mm3&quot;</td>
<td>cubic millimeters</td>
</tr>
<tr>
<td>&quot;mol&quot;</td>
<td>mole</td>
</tr>
<tr>
<td>&quot;MPa&quot;</td>
<td>megapascals</td>
</tr>
<tr>
<td>&quot;N&quot;</td>
<td>Newton</td>
</tr>
<tr>
<td>&quot;nA&quot;</td>
<td>nano amperes</td>
</tr>
<tr>
<td>&quot;nCoul&quot;</td>
<td>nano coulomb</td>
</tr>
<tr>
<td>&quot;nFD&quot;</td>
<td>nano farad</td>
</tr>
<tr>
<td>&quot;nH&quot;</td>
<td>nano henry</td>
</tr>
<tr>
<td>&quot;nOhm&quot;</td>
<td>nano ohm</td>
</tr>
<tr>
<td>&quot;nS&quot;</td>
<td>nano siemens</td>
</tr>
<tr>
<td>&quot;nsec&quot;</td>
<td>nano second</td>
</tr>
<tr>
<td>&quot;nV&quot;</td>
<td>nano volt</td>
</tr>
<tr>
<td>&quot;nW&quot;</td>
<td>nano watts</td>
</tr>
<tr>
<td>&quot;nWb&quot;</td>
<td>nano weber</td>
</tr>
<tr>
<td>&quot;N*m&quot;</td>
<td>Newton-meters</td>
</tr>
<tr>
<td>Unit String</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>&quot;N/m²&quot;</td>
<td>Newtons per meter squared</td>
</tr>
<tr>
<td>&quot;none&quot;</td>
<td>no unit</td>
</tr>
<tr>
<td>&quot;nmi&quot;</td>
<td>nautical mile</td>
</tr>
<tr>
<td>&quot;ohm&quot;</td>
<td>ohm</td>
</tr>
<tr>
<td>&quot;Pa&quot;</td>
<td>Pascal</td>
</tr>
<tr>
<td>&quot;pCoul&quot;</td>
<td>pico coulomb</td>
</tr>
<tr>
<td>&quot;pFD&quot;</td>
<td>pico farad</td>
</tr>
<tr>
<td>&quot;pWb&quot;</td>
<td>pico weber</td>
</tr>
<tr>
<td>&quot;pH&quot;</td>
<td>pico henry</td>
</tr>
<tr>
<td>&quot;pS&quot;</td>
<td>pico siemens</td>
</tr>
<tr>
<td>&quot;psec&quot;</td>
<td>pico second</td>
</tr>
<tr>
<td>&quot;psia&quot;</td>
<td>pounds per square inch</td>
</tr>
<tr>
<td>&quot;psia/sec&quot;</td>
<td>pounds per square inch absolute per second</td>
</tr>
<tr>
<td>&quot;°R&quot;</td>
<td>degrees Rankine</td>
</tr>
<tr>
<td>&quot;°R/sec&quot;</td>
<td>degrees Rankine per second</td>
</tr>
<tr>
<td>&quot;rad&quot;</td>
<td>radians</td>
</tr>
<tr>
<td>&quot;rad/sec&quot;</td>
<td>radians per second</td>
</tr>
<tr>
<td>&quot;rev&quot;</td>
<td>revolution</td>
</tr>
<tr>
<td>&quot;rev/sec&quot;</td>
<td>revolutions per second</td>
</tr>
<tr>
<td>&quot;rpm&quot;</td>
<td>revolutions per minute</td>
</tr>
<tr>
<td>&quot;rpm/sec&quot;</td>
<td>revolutions per minute absolute per second</td>
</tr>
<tr>
<td>&quot;rpm/SQRT_R&quot;</td>
<td>revolutions per minute square root degrees Rankine</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>siemens</td>
</tr>
<tr>
<td>&quot;sec&quot;</td>
<td>second</td>
</tr>
<tr>
<td>&quot;sec²&quot;</td>
<td>second squared</td>
</tr>
<tr>
<td>&quot;sec³&quot;</td>
<td>second cubed</td>
</tr>
<tr>
<td>&quot;slug&quot;</td>
<td>slug</td>
</tr>
<tr>
<td>&quot;slug*ft²&quot;</td>
<td>slug-square feet</td>
</tr>
<tr>
<td>&quot;slug*ft³&quot;</td>
<td>slug per foot cubed</td>
</tr>
<tr>
<td>&quot;SQRT_lbf&quot;</td>
<td>square root pound force</td>
</tr>
<tr>
<td>&quot;SQRT_N&quot;</td>
<td>square root Newton</td>
</tr>
<tr>
<td>&quot;SQRT_R&quot;</td>
<td>square root degrees Rankine</td>
</tr>
<tr>
<td>&quot;μA&quot;</td>
<td>micro ampere</td>
</tr>
<tr>
<td>&quot;μCoul&quot;</td>
<td>micro coulomb</td>
</tr>
<tr>
<td>&quot;μFD&quot;</td>
<td>micro farad</td>
</tr>
<tr>
<td>&quot;μH&quot;</td>
<td>micro henry</td>
</tr>
<tr>
<td>&quot;μOhm&quot;</td>
<td>micro ohm</td>
</tr>
<tr>
<td>&quot;μS&quot;</td>
<td>micro siemens</td>
</tr>
<tr>
<td>&quot;μsec&quot;</td>
<td>micro second</td>
</tr>
<tr>
<td>&quot;μV&quot;</td>
<td>micro volt</td>
</tr>
<tr>
<td>&quot;μW&quot;</td>
<td>micro watts</td>
</tr>
<tr>
<td>&quot;μWb&quot;</td>
<td>micro weber</td>
</tr>
<tr>
<td>&quot;V&quot;</td>
<td>volt</td>
</tr>
<tr>
<td>&quot;W&quot;</td>
<td>Watts</td>
</tr>
<tr>
<td>&quot;W/(m*K)&quot;</td>
<td>Watts per meter per degree Kelvin</td>
</tr>
<tr>
<td>&quot;W/m²&quot;</td>
<td>Watts per meter squared</td>
</tr>
<tr>
<td>&quot;W/(m²*K)&quot;</td>
<td>Watts per meter squared per degree Kelvin</td>
</tr>
<tr>
<td>&quot;W/m³&quot;</td>
<td>Watts per meter cubed</td>
</tr>
<tr>
<td>&quot;Wb&quot;</td>
<td>weber</td>
</tr>
<tr>
<td>&quot;yd&quot;</td>
<td>yard</td>
</tr>
</tbody>
</table>
12 Function Summary

12.1 User-Accessible Member Functions

In the following sections, user-accessible member functions are categorized according to the object types that they operate on.

12.1.1 Functions for All NPSS Objects (Elements, Subelements, Assemblies, DataViewers, Solver, Ports, Stations, Sockets) and Variables

void create (string baseType, string type, string instanceName)

Creates an object of the given baseType and type with the given instance name. If a function: creates an object using the create() function, the newly created object resides in the function's parent. Therefore, the "parent." token must be used in front of the object name when referencing it from inside the function. For example:

```cpp
void test () {
    setThermoPackage("GasTbl");
    create("Element", "Duct", "d1"); // Creates a Duct element named "d1"

    cout << exists("d1") << endl; // d1 does NOT exist inside this function.
    cout << parent.exists("d1") << endl; // d1 DOES exist in this function's parent
}
```

The output is as follows.

0
1
1

After invoking create(), instanceName:VCinit() should be called to complete the initialization of the new object.

any evalExpr(string expression)

Evaluates expression and returns the result.

```cpp
cout << evalExpr("2 + 4") << endl; // 6
```

int exists(string name)

Returns 1 (TRUE) if the object exists in the model, 0 (FALSE if) it does not.

string getDataType()

Returns the object's data type such as: int, real string, void, real[], real[][], real[][][], int[], int[][], string[], any, Matrix, no_value.

```cpp
real x;
cout << x.getDataType() << endl; // real
```

string getName()

Returns the object's local name.

int getPassType()

Returns integers 0, 1, or 2. Global integer constants NO_PASS = -1, FIRST_PASS = 0, PERTURBATION_PASS = 1, and ITERATION_PASS = 2 have been defined to allow test of the return value of this function to be more readable. This function is only available for Elements, Subelements, and Assemblies. The return values refer to
whether the current Model pass is the first pass of a convergence attempt, a matrix generation pass where an
independent has been perturbed, or a pass where the solver is varying the Independent parameters in order to
converge the system, or in the case of noPass, the solver is no longer in control. An example of the usage of this
function is in the preexecute function of an element to determine if the current pass is the first pass of a new
convergence attempt so that a guess of the independent parameters can be made.

```plaintext
string getPathName()
Returns the object's full pathname.
```

```plaintext
int hasInterface (string typeName)
Returns 1(TRUE) if the object is of the type specified by typeName. It also returns 1 (TRUE) if the object is
inherited from the given type. For example, if object myDuct is a Duct object, which is inherited from Element,
then both of the following function calls would return true:

myDuct.hasInterface("Duct");       // returns 1(TRUE) since myDuct is a Duct.
myDuct.hasInterface("Element");    // returns 1 (TRUE) since myDuct inherits
                                    // from Element.
myDuct.hasInterface("Subelement"); // returns 0 (FALSE) because a Duct is not
                                    // a Subelement.
```

```plaintext
int hashCollisions()
Returns the number of collisions in this VariableContainer’s instance hash table. The instance hash table is
used for finding objects within the VariableContainer. Excessive collisions (object names hashing to the same
value) can impair performance in some situations.
```

```plaintext
real[] hashStats()
Returns an array of statistics for this VariableContainer’s instance and class hash tables.
```

```plaintext
string[] hashStatNames()
Returns an array of names for the statistics returned by hashStats().
```

```plaintext
int hashVal(string name)
Returns the hash value associated with name.
```

```plaintext
void hide(int flag)
This function is intended to insulate the user from parts of a model that they don't need to be aware of. Objects
marked as hidden are invisible to the list() command. Also, list() will not search inside of a hidden object
unless called directly on that object. Consequently, marking a parent object as hidden effectively hides all of that
object's children, even though they are not explicitly marked as hidden. hide() can also affect the behavior of an
object if that object is converted to C++. Valid options to hide() are as follows:

flag value :  result
0        :  object is NOT hidden. This is the default.
1  :  object is hidden. It will not be returned from any list() command unless showHidden is TRUE.
2        :  object is hidden AND will not be user-accessible at all in the C++ version.
```

showHidden is a global variable that can be used to make all hidden variables visible to the list() command. If
its value is TRUE, all variables will behave as if they are not hidden. If its value is FALSE, objects marked as
hidden will not be returned from list().

```plaintext
showHidden=FALSE; // this is the default

To hide an object:
myObj.hide(1);

To reveal an object:
```
myObj.hide(0);

To make an object hidden AND prevent it from being user-accessible at all in the C++ version of the given component:

myObj.hide(2);

To make all objects visible to the list() command:

showHidden=TRUE;

Note: a hide value of 2 should only be used with variables or functions, not with other kinds of objects, e.g., Subelements. Functions and variables are internally accessible after a component is converted to C++ regardless of whether or not they are user-accessible, but other types of objects are only accessible internally if they are user-accessible.

int hidden()
Returns 0 for visible objects, 1 for hidden objects, and 2 for objects that are hidden and will not be user-accessible when converted to C++.

string isA()
Returns the type of the object.

    cout << myDuct.isA() << endl; // duct

string[] list (string type, int recurse, string expression)
Returns a list of sorted objects, each of which must match the given type and return TRUE for the given expression. The third argument, a conditional expression, is optional but must be a string that when parsed and evaluated produces a TRUE or a FALSE. Include only the first and second arguments to list all objects matching a given type. To list all Elements, for example: list("Element", TRUE). To narrow the list, use the conditional filtering functionality by inserting an expression as the third argument. To list all variables with an IStatus set to "input", for example: list("Variable", TRUE, "IStatus == "input"""). (Note: In the previous example, to include the string "input" inside the expression, one must "escape" the quotation marks with a backward slash (\") so that the quotation marks are not mistaken by the parser as the quotation marks which end the third argument.) Specific element and subelement types may also be listed, for example, Duct, Burner, Compressor, etc.

string[] listInterfaces()
Returns a string array. For each name that appears in the string array, hasInterface(name) on the given object will return 1 (TRUE). (Note: Use this function to list socket types.)

void reconfigureHash(real targetACL, int addAtHead, int recurse)
Reconfigures the instance and class hash tables for the specified target average chain length. A chain is a list of object names which hash to the same value. Long chains can affect performance in some situations. If addAtHead is TRUE, then new colliding object names are added at the beginning of the chain rather than being added at the end (the default). If recurse is TRUE, then all child VariableContainers are similarly configured as well.

int setOption(string name, any value)
This function will set the variable specified by "name" in this object and every object contained within this object. The named variable will be set only where it exists, i.e., if some child objects do not contain the named variable, they will simply be skipped. As long as there is at least one instance of the named variable somewhere in the hierarchy at or below the level where the function is called, the return value of setOption will be 1. A return value of 0 indicates that no variables matching the given name were found. Note that the specified variable can be of any type, i.e., it does not have to be an Option variable.

void showHash()
Displays the instance hash table on cout.
void showHashStats([string prefix [, string stream]])
Displays hash table statistics. Statistics lines are prefixed by prefix (default null), and written to the named
stream (default cout).

int varNameIsActiveIndep(string name)
Returns 1 (TRUE) if the specified variable is controlled by an Independent object that is part of the current Solver
setup. If the variable is pointed to by an Independent object, but that Independent is not part of the set of
Independents being used by the Solver when this function is called, then the function will return 0 (FALSE). The
variable path name must be the full path name of the variable. This function is only available for Elements,
Subelements, and Assemblies. An example usage is to determine which input variables for which it is acceptable
to provide guess values. Guesses are only appropriate for variables that are controlled by an active Independent,
otherwise a user input value could be overwritten by the guess value.

int verify()
Returns 1 (TRUE) if the model is valid, 0 (FALSE) if it is not. Also displays warning messages describing what is
wrong.

void saveA(real saveTime, [string filename="NPSSRestart.dat"], [int precision=12],
[string[] ExtraVariables=NULL])
The saveA function allows the user to save NPSS solver and system variables in ASCII format, including solver
states, state derivatives, independents and any user variables passed to the function. This is a dynamic process
where the function will determine what variables exist in memory and capture their type and value. This function
may be used to save the state of the model after a steady-state run or at the end of a transient run.

The first argument is the specified saveTime. It is the point in time when the data was saved, i.e. the model is run
to a particular stop time and the data is saved at that point. The value for saveTime need not be the same as the
actual global variable time.

The second argument, filename, is the name of the saved data file. This argument is optional and the default value
is "NPSSRestart.dat".

The third argument, precision, allows the user to specify the precision of the saved data. This argument is optional
and the default precision is 12.

The fourth argument is an optional variable data array to be stored along with the detected NPSS variables. This
argument intended to be used for storing variables that did not exist in memory when the run() function was
called. The format of the array is (name1, value1, name2, value2, etc.), and all entries must be in quotes.

Examples:
To save NPSS variables at runtime = 0.75 seconds using the saveA default values for filename and precision.

        transient.stopTime = 0.75; // set stop time at 0.75 sec
        run(); // run the transient, stopping at 0.75 sec
        saveA(0.75); // save ASCII file at 0.75 sec

To save NPSS variables at runtime = 0.75 seconds adding 4 additional variables (4 name/value pairs).

        transient.stopTime = 0.75; // set stop time at 0.75 sec
        run(); // run the transient, stopping at 0.75 sec
        // declare 4 new variables (after the run has finished)
        real r1 = 0.1;
        real r2 = 0.2;
        int i1 = 2;
        int i2 = 10;
// store 4 variable names and values in string array saveV[]
string saveV[8];
saveV[0] = "r1";
saveV[1] = "0.1";
saveV[2] = "r2";
saveV[3] = "0.2";
saveV[4] = "i1";
saveV[5] = "2";
saveV[6] = "i2";
saveV[7] = "10";

// Save ASCII file at 0.75 sec, specifying filename, precision,
// and extra variables
saveA(0.75,"NPSSRestart.dat",12,saveV);

A example of the data typically saved is shown below:

The transient behavior for a spring model with second order damping includes custom element classes of
the type: Spring, MassObj and Damper. With declared elements: Spring mySpring, Damper myDamper and
MassObj myMass. The first value saved is always the stop time followed by the
variables within the declared model elements and then any extra variables declared, in this case r1, r2,
i1 and i2.

time = 0.75;
myDamper.autoAddToSolvSeq = 1;
myDamper.b = 10;
myDamper.Fdrag = -7;
myMass.accel = -0.655726347142;
myMass.autoAddToSolvSeq = 1;
myMass.basePosition = 5;
myMass.Finput = 789.55;
myMass.Fnet = -0.655726347142;
myMass.mass = 1;
myMass.position = 4.95862489179;
myMass.positionIndep.a = 0;
myMass.positionIndep.x = 4.95862489179;
myMass.positionIndep.xFirst = 4.95903472076;
myMass.positionIndep.xFirstErr = 0.00826497216279;
myMass.positionIndep.xLimitCheck = 0;
myMass.positionIndep.xLimitExceeded = 0;
myMass.positionIndep.xModel = 4.95862489179;
myMass.positionInteg.conflictingMin = "";
myMass.positionInteg.derivativeName.trigger = 0;
myMass.positionInteg.derivativeName = "speed";
myMass.positionInteg.derivativeName.trigger = 1;
myMass.positionInteg.description = "";
myMass.positionInteg.eq_lhs.trigger = 0;
myMass.positionInteg.eq_lhs = "basePosition";
myMass.positionInteg.eq_lhs.trigger = 1;
myMass.positionInteg.eq_Ref.trigger = 0;
myMass.positionInteg.eq_Ref = "1.0"
myMass.positionInteg.eq_Ref.trigger = 1;
myMass.positionInteg.eq_RefTransient.trigger = 0;
myMass.positionInteg.eq_RefTransient = "";
myMass.positionInteg.eq_RefTransient.trigger = 1;
myMass.positionInteg.eq_rhs.trigger = 0;
myMass.positionInteg.eq_rhs = "position";
myMass.positionInteg.eq_rhs.trigger = 1;
myMass.positionInteg.iDescription = "";
myMass.positionInteg.integErrorForm.trigger = 0;
myMass.positionInteg.integErrorForm = "INTEGRAL";
myMass.positionInteg.integErrorForm.trigger = 1;
myMass.positionInteg.integrationType = "GEAR_1ST_ORDER";
myMass.positionInteg.integrationType.trigger = 0;
myMass.positionInteg.iResolveMinMaxConflict = "MIN";
myMass.positionInteg.resolveMinMaxConflict.trigger = 0;
myMass.positionInteg.resolveMinMaxConflict = "MAX";
myMass.positionInteg.resolveMinMaxConflict.trigger = 1;
myMass.positionInteg.resolveYRefLock = "PER_PARENT";
myMass.positionInteg.solutionMode.trigger = 0;
myMass.positionInteg.solutionMode = "TRANSIENT";
myMass.positionInteg.solutionMode.trigger = 1;
myMass.positionInteg.stateName.trigger = 0;
myMass.positionInteg.stateName = "position";
myMass.positionInteg.stateName.trigger = 1;
myMass.positionInteg.timeConstantExpr.trigger = 0;
myMass.positionInteg.timeConstantExpr = "";
myMass.positionInteg.timeConstantExpr.trigger = 1;
myMass.positionInteg.toleranceType = "FRACTIONAL";
myMass.positionInteg.toleranceType.trigger = 0;
myMass.speed = 0.714639133581;
myMass.speedIndep.a = 0;
myMass.speedIndep.autoSetup = 1;
myMass.speedIndep.dxLimit = 0.1;
myMass.speedIndep.lockDxLimit = 0;
myMass.speedIndep.lockDxLimitType = 0;
myMass.speedIndep.lockPerturbation = 0;
myMass.speedIndep.lockPerturbationType = 0;
myMass.speedIndep.lockXTolerance = 0;
myMass.speedIndep.lockXToleranceType = 0;
myMass.speedIndep.perturbation = 0.001;
myMass.speedIndep.s = 1;
myMass.speedIndep.x = 0.714639133581;
myMass.speedIndep.xFirst = 0.781046670163;
myMass.speedIndep.xFirstErr = 9.2924573343;
myMass.speedIndep.xFirstErrorRef = 0.442716910548;
myMass.speedIndep.xLimitCheck = 0;
myMass.speedIndep.xLimitExceeded = 0;
myMass.speedIndep.xLimitReport = 0;
myMass.speedIndep.xModel = 0.714639133581;
232  12  Function Summary

myMass.speedIndep.xModelFirst = 0.781046670163;
myMass.speedIndep.xModelFirstErr = 9.2924573343;
myMass.speedIndep.xPrevious = 0.714639133581;
myMass.speedIndep.xRef = 15;
myMass.speedIndep.xTolerance = 0.0001;
myMass.speedInteg.conflictingMax = "";
myMass.speedInteg.conflictingMin = "";
myMass.speedInteg.derivativeName.trigger = 0;
myMass.speedInteg.derivativeName = "accel";
myMass.speedInteg.description = "";
myMass.speedInteg.eq_lhs.trigger = 0;
myMass.speedInteg.eq_lhs = "myMass.Finput";
myMass.speedInteg.eq_lhs.trigger = 1;
myMass.speedInteg.eq_Ref.trigger = 0;
myMass.speedInteg.eq_Ref = "1.0";
myMass.speedInteg.eq_Ref.trigger = 1;
myMass.speedInteg.eq_RefTransient.trigger = 0;
myMass.speedInteg.eq_RefTransient = "";
myMass.speedInteg.eq_RefTransient.trigger = 1;
myMass.speedInteg.eq_rhs.trigger = 0;
myMass.speedInteg.eq_rhs = "myMass.position*mySpring.k - myDamper.Fdrag";
myMass.speedInteg.eq_rhs.trigger = 1;
myMass.speedInteg.iDescription = "";
myMass.speedInteg.integErrorForm.trigger = 0;
myMass.speedInteg.integErrorForm = "INTEGRAL";
myMass.speedInteg.integErrorForm.trigger = 1;
myMass.speedInteg.integrationType.trigger = 0;
myMass.speedInteg.integrationType = "GEAR_1ST_ORDER";
myMass.speedInteg.iResolveMinMaxConflict = "MIN";
myMass.speedInteg.resolveMinMaxConflict.trigger = 0;
myMass.speedInteg.resolveMinMaxConflict = "MAX";
myMass.speedInteg.resolveMinMaxConflict.trigger = 1;
myMass.speedInteg.solutionMode.trigger = 0;
myMass.speedInteg.solutionMode = "TRANSIENT";
myMass.speedInteg.solutionMode.trigger = 1;
myMass.speedInteg.stateName.trigger = 0;
myMass.speedInteg.stateName = "speed";
myMass.speedInteg.stateName.trigger = 1;
myMass.speedInteg.timeConstantExpr.trigger = 0;
myMass.speedInteg.timeConstantExpr = "";
myMass.speedInteg.timeConstantExpr.trigger = 1;
myMass.speedInteg.toleranceType.trigger = 0;
myMass.speedInteg.toleranceType = "FRACTIONAL";
myMass.speedInteg.toleranceType.trigger = 1;
mySpring.autoAddToSolvSeq = 1;
mySpring.displace = 0;
mySpring.Frestore = 6;
mySpring.k = 157;
postData.autoAddToSolvSeq = 1;
testResults.amp = 5;
testResults.autoAddToSolvSeq = 1;
testResults.dampingRatio = 0.400279333787;
testResults.dampTerm = 0.916393177049;
testResults.idealPos = 5.04547408562;
testResults.massPos = 4.95862489179;
testResults.omega = 12.5662245722;
testResults.outputError = -0.017213287068;
yInput.autoAddToSolvSeq = 1;
yInput.outputVal = 5;
void restartA([string filename=“NPSSRestart.dat”])

The `restartA` function allows the user to recall the values of NPSS variables saved in ASCII format into system memory. The `filename` argument is optional and the default name is "NPSSRestart.dat". If restoring a transient point, the transient start time will be automatically set to the time saved in the restart file.

Example of using the default filename of “NPSSRestart.dat”. The model is then run from the state of the restart file at time = 0.75 seconds to a new stop time of 2.5 seconds. The model will automatically be put into transient mode because saved point was a transient run.

```cpp
// Start from the restart file point (transient mode at 0.75 sec) and run out
// to 2.5 sec. The restart time will automatically be 0.75 sec.

// load saved model point using default filename
restartA(); // could also have loaded file using #include "NPSSRestart.dat"

transient.stopTime = 2.5; // set stop time at 2.5 sec
initializeHistory();

run(); // run the transient, stopping at 2.5 sec
```
12.1.2 Array Functions

void append(type value)
If type is the type of an array member, this function appends value to the end of the array. If type is the type of the array, this function appends each top-level member of value to the end of the array.

int contains(type value)
Returns TRUE if there is a member of the array equal to value.

int entries()
Returns the number of entries in the array.

void getMember(int i1, [int i2], [int i3])
Accepts a minimum of 1 to a maximum of 3 int values as arguments for 1D, 2D, and 3D arrays. Returns the value of the element indexed by the argument(s).

int index(type value)
Returns the first array index at which value was found, or -1.

void insertAt(int index, type value)
Inserts value at the location specified by the index and moves all the values that follow the inserted value down. No value is deleted and the order of the values following the inserted one does not change.

void remove(type value)
Removes the first member found equal to value from the array.

void resize(int newSize)
Resizes the current array to the new size specified. To release allocated memory, you must call resize(0). Merely resizing to a smaller size does not release any memory.

void setMember(int i1, [int i2], [int i3], type val)
Accepts a minimum of 1 to a maximum of 3 int values as arguments for 1D, 2D, and 3D arrays as well as an element value argument. Type is the type of array member. It sets the element indexed by the argument(s) to the value specified by val.

For string arrays only:

int containsIgnoreCase(string value)
Is like contains(), but ignores case distinctions.

void insertByName(string anchor, int offset, string strToInsert)
Inserts the string strToInsert offset places from the the string anchor. A zero offset inserts the string just ahead of the anchor string. A negative offset inserts further towards to start of the array. No value is deleted and the order of the values following the inserted one does not change.

void replaceByName(string strToReplace, string strReplaceWith)
Replaces the string strToReplace with the string strReplaceWith. The number of values in the array does not change.

12.1.3 Assembly Functions

string getExecutive()
Returns the name of an Assembly’s current Executive.
void linkPorts (string port1, string port2, string stationName)
Connects port1 to port2 and labels the station with stationName.

void linkPortsDesc (string port1, string port2, string stationName, string iDescription)
Connects port1 to port2 and labels the station with stationName. Adds iDescription to the linked ports and stationName. Assigns a default station name if the stationName argument is an empty string ("").

void postKillPoint(int sequence, int index)
This function may be overwritten by the user to specify desired behavior after an NCPKillPointException has been generated. The sequence specifies in which solver sequence the exception occurred. The the index specifies in which element the exception occurred. (Please refer to Section 4.14.9 [errHandler Functions] for more detailed information).

void promotePort (string port, string newName)
Tells NPSS that an element's port (string) is to be considered a port on the specified assembly (string).

void setExecutive (string executiveName)
If executiveName is the name of an executive in the current Assembly, then it is set to be the Assembly's current Executive.

void unpromotePort (string port)
Undoes the work of promotePort by taking the name of a port that has been promoted to the assembly level (string), unlinking that port from any connections, and un-registering it from its current Assembly.

void setupExecutive()
Sets up the current Executive in an Assembly to prepare it to be run.

12.1.4 BleedInPort/BleedOutPort Functions
These functions are also used for InterStageBleedInPort and InterStageBleedOutPort.

void setBleedStation()
Sets the bleed station conditions based on current cycle conditions and bleed inputs. For optimal performance, call this function inside the verify() function.

string getStation()
Returns the station inside the bleed port.

void updateBleed ()
Calculates current station conditions; must be called after setBleedStation() has been called at least once. For optimal performance, call this function should be called inside the calculate() function.

12.1.5 DataViewer Functions

void display()
Causes the named DataViewer to print or display its data. display() is now automatically called when a DataViewer has buffered a full page of information. This cuts memory use that was taking place with extraordinarily large buffered data. display() may be called explicitly by the user at the desired time without side effects to the automatic feature.

If the user fails to specify the pageLength of a CaseRowViewer, the page length is assumed to be infinite, but the DataViewer will display automatically once a default amount of data has been buffered. This automatic display of CaseRowViewer will not insert spaces and headers if showHeaders is set to FALSE. This way, the automatic display will save memory without ruining plotter-friendly data.
If the user fails to specify the page width of a CaseColumnViewer, the DataView will assume an infinitely wide page, as would be consistent with generating plotter-friendly data. Because of the way CaseColumn data is displayed, however, it cannot be displayed automatically and still remain plotter-friendly. In the case of infinite pageWidth of a CaseColumnViewer, memory use and performance will suffer, since there is no choice but to buffer all data at once. If a pageWidth is provided for CaseColumnViewer, then it will display automatically and achieve higher performance, but will likely not be plotter-friendly.

display() is also automatically called upon destruction of the DataView. If an explicit user call to reset() a DataView is made, it is assumed that the user wants to destroy the information stored in the DataView buffer, and that information is not automatically display(ed. An explicit display() would have to be called to clear out the buffer before a reset().

void update()
Updates the named DataView. Updating a DataView causes it to retrieve data from the model in its current state. Some types of DataViewers, for example, the VarDumpViewer, will also print the data at this point, while others, such as the CaseColumnViewer, will store the data and display it if it has accumulated enough buffered data to force an automatic display().

### 12.1.6 Dependent Functions

void revertToDefaults()
Unlocks all "lockable" attributes in this dependent. This function is also available for objects derived from a Dependent, such as the Integrator.

### 12.1.7 Element, Assembly, and Subelement Functions

void autoSolverSetup([int recurse])
This assembly function tells the solver to retrieve the default independents and dependents from the model. Parameter recurse is optional and defaults to TRUE (non-zero). A value of FALSE (0) forces the function to set up ONLY the solver in the assembly the function is called on.

string getTextRep(int saveLevel)
Returns a string containing an ASCII text representation of the current assembly, element, or subelement. Parameter saveLevel can be set to either 1 or 0. A saveLevel of 1 gives a full save, delving into each object's children and the variables contained within. A saveLevel of 0 saves a more skeletal state representation.

void run()
Runs the given element or assembly. At the top level, runs the entire model.

void addInterface (string socketType)
Allows the Subelement to be inserted into a Socket with type socketType, even if the Subelement is not of type socketType and is not derived from a socketType class.

string[] listSocketTypes ()
Returns a string array containing all the socketTypes that the Subelement can be plugged into. Note: listSocketTypes() is deprecated. Please use listInterfaces() instead.

void tree()
Displays a hierarchical view of the model presented in execution sequence. The elements are listed in the order defined in the solver sequence array. Any elements not named there will be listed next.

### 12.1.8 errHandler Functions

Please see Section 4.14.9 for the errHandler functions. It is a subsection of 4.14, Error Handling and Messages.
### 12.1.9 External Component Attributes and Functions

These attributes and functions are available with all external components, and are used with CORBA builds.

**int proxyTraceLevel**
Debug tracing level for proxy (client) side of CORBA transactions, default 0.

**int proxySynchronous**
Perform calculate(), parseFile(), parseString(), sendFile(), recvFile(), and send() operations synchronously, default TRUE.

**int proxyMaxRetries**
Max CORBA::TRANSIENT retries within proxyRetryWindow, default 10.

**int proxyRetryWindow**
Window within which no more than proxyMaxRetries are attempted (seconds), default 600.

**void parseFile(string remoteName)**
Parses the file remoteName, which must exist on the external component's server. What it means to parse a file is component-dependent.

**void parseString(string text)**
Parses the given string. What it means to parse a string is component-dependent.

**void revertToDefaults()**
Revert to a default or initial state. Exact definition is component-dependent.

**real getFile(string remoteName)**
**real getFile(string remoteName, string localName)**
**real getFile(string remoteName, string localName, int buffSize)**
Copies a remote file to the local system (i.e., the host running NPSS). Simple binary copy, no attempt at translation from the remote system's representation is made. The single argument form uses remoteName as the name of the local copy. The default for buffSize is 128KB. Returns the data transfer rate (bytes/sec).

**real putFile(string localName)**
**real putFile(string localName, string remoteName)**
**real putFile(string localName, string remoteName, int buffSize)**
Copies a local file to the remote system (i.e., the host running the server associated with the external component). Simple binary copy, no attempt at translation to the remote system's representation is made. The single argument form uses localName as the name of the remote copy. The default for buffSize is 128kB. Returns the data transfer rate (bytes/sec).

**real sendFile(string localName, string dest)**
**real sendFile(string localName, string dest, string destName)**
Send a file from the server associated with the external component to another component, dest. In this context, localName refers to a file on the server associated with the external component, not a file local to the host running NPSS. The two-argument form uses localName as the name of the destination copy. The destination component dest may be an interpreted, internal, or external component. If dest is an interpreted or internal component, it is checked for a directory attribute. If this attribute exists, it is prepended to destName, resulting in a destination path of directory/destName. Returns the data transfer rate (bytes/sec).
real recvFile(string src, string srcName)
real recvFile(string src, string srcName, string localName)
Receive a file from the server associated with another component, src. In this context, localName refers to a file on the server associated with the external component, not a file local to the host running NPSS. The two-argument form uses srcName as the name of the destination copy. The source component src may be an interpreted, internal, or external component. If src is an interpreted or internal component, it is checked for a directory attribute. If this attribute exists, it is prepended to srcName, resulting in a source path of directory/srcName. Returns the data transfer rate (bytes/sec).

void removeFile(string remoteName)
Removes the file from the remote system.

void send(string localVar, string remote)
void send(string localVar, string remote, string remoteVar)
void send(string localVars[], string remote)
void send(string localVars[], string remote, string remoteVars[])
Sends one or more variables from the server associated with the external component to another external component, remote. In this context, localVar refers to a variable on the server associated with the external component, not a variable in the NPSS model. The two-argument forms use localVar(s) as the name(s) on remote.

void shutdown()
Shuts down remote server. Any attempts to access a shut down server will fail.

int testReady()
Returns TRUE if the component is ready to process a new operation.

void wait()
Waits for any previous (presumably asynchronously invoked) operation to complete. Typically this function is not needed as all operations implicitly wait for the previous operation to complete before invoking the next.

12.1.10 FlowStation Functions

Note: All FlowStation functions can be called from the FluidOutputPort side of a link. However, only get functions can be called from the FluidInputPort side of a link. Any other FlowStation functions called on a linked FluidInputPort will generate an error message. FluidInputPorts that are not linked have access to all FlowStation functions.

Note: Any FlowStation functions that take a stationName as a string argument must NOT contain the element name as part of the string. Otherwise, NPSS will not be able to resolve the station name. For example,

DUCT.Fl_O.copyFlow("Fl_I")  //right way

DUCT.Fl_O.copyFlow("DUCT.Fl_I")  //wrong way

Note: Not all FlowStations use all of these functions.

void add(string stationName)
Adds the secondary stream stationName to this flow station. Keeps the total pressure the same and uses an energy balance to calculate the new temperature.

void flowStationName.burn(string fuelStationName, real efficiency)
Adds a fuel stream (fuelStationName) to the specified flow station (FlowStationName) and burns unburned fuel to the given efficiency.

void copyFlow(string stationName)
Copies flow total conditions from the specified stationName to this flow station.
void copyFlowRemote(string stationName)
Copies flow total conditions from the specified stationName outside the local scope to this flow station.

void copyFlowStatic(string stationName)
Copies total conditions and static conditions from the specified stationName to this flow station.

void copyFlowStaticRemote(string stationName)
Copies total conditions and static conditions from the specified stationName outside the local scope to this flow station.

real getExergy(real Tref, real Pref)
Temporarily set the state to the input reference temperature (Tref) and reference pressure (Pref) and return exergy defined as:

\[ \text{exergy} = h - h_{ref} - Tref \times (s - s_{ref}) \]

Where 'ref' refers to the state conditions at the the input reference temperature and pressure.
Exergy is equal to the value of h (enthalpy) before setting the input reference temperature and pressure, minus h after setting the reference temperature and pressure, minus the input reference temperature times the difference in the value of s (entropy) before and after setting the reference temperature and pressure.
The state is set back to the original temperature and pressure values before getExergy() returns.

real getIdealEnthalpy(real Sref)
Temporarily sets the state to a condition defined by the input entropy Sref and current pressure and then returns the enthalpy for that state. State is returned to its previous value when the function returns.

real getIdealWork(real Pref)
Temporarily sets the state to a condition defined by the input pressure Pref and current entropy and then returns the enthalpy difference between the two states. State is returned to its previous value when the function returns.

string getStationType()
Returns the flow station type as assigned by the setThermoPackage command

virtual int haveStatic()
Returns a flag to let the user know if statics are available.

void setDryAir()
Sets the consistency of the flow to dry air. This may be called for the following thermo packages: AllFuel and FPT. The following parameters are set:
Common to all thermo packages:
The Water to Air Ratio is set to zero for the flowstation through a call to setWAR(0.0)

Additional actions specific to thermo package allFuel:
The following FlowStation parameters are set:
Fraction of flow that is inlet flow = 1.0
Fraction of flow that is unburned fuel = 0.0
Fraction of flow that is burned fuel = 0.0
Fuel to Air Ratio = 0.0
Water to Air Ratio = 0.0
Status of static values = NO

setTotalTP(Tt, Pt) is called which sets the total conditions based on the total temperature and total pressure.
Additional actions specific to thermo package FPTFlowStation

Status of static values = NO
The Water to Air Ratio of the FlowStation is set to 0.0.
setTotalTP(Tt, Pt) is called which sets the total conditions based on the total temperature and total pressure.

```c
void setStatic_u__rho(real u, real rho)
```
Set total conditions based on static input internal energy and density.

The only thermo package FlowStation that uses this set function is allFuel

The difference between the input energy and the current static energy is used in a total enthalpy solver to determine the total enthalpy, ht. The difference between the input density and the current static density is used in a total pressure solver to determine the total pressure, Pt.

```c
void setTotal_hP(real h, real P)
```
Sets total conditions based on the input enthalpy and pressure.

Common to all thermo packages:

The following FlowStation parameters are set:
Sets the total enthalpy to the input value.
Sets the total pressure to the input value.
Sets the total temperature through a call to get_T_hP(real h, real P) using the input values of h and P.

Additional actions specific to thermo package allFuel:

Sets the following FlowStation parameters:
Sets the total pressure to the input value.
Sets the total enthalpy to the input value.
oar = har = 0.0
air = fracAir
plog = log(Pt)
h = ht
FAR = FAR*air
war = WAR*air

After the following function call, additional parameters are set:

```
thofhc_(&h, &plog, &t, &s, &cs, &r, (char*)_jphase.data(), allFuelFlowStation::_tfuel, &FAR, &war, &air, &har, &oar)
```
Tt = t
Rt = r
gamt = cs/(cs – r)
entropy = s
Cp = cs
Rhot = Pt*144/(Rt*Tt*C_BTUtoFT_LBF)
Status of static values = NO

Additional actions specific to thermo package FPT:

Sets the following FlowStation parameters:
Sets the total pressure to the input value.
Sets the total enthalpy to the input value.
Sets the total temperature through a call to get_T_ht().
Sets the flowstation’s entropy through a call to get_s(this).
Status of static values = NO
Additional actions specific to thermo package JanafFlowStation

Sets the following FlowStation parameters:
- Sets the total pressure to the input value.
- Sets the total enthalpy to the input value.
- Status of static values = NO

If the Fuel to Air Ratio > 0 the total temperature is determined through a Tt solver based on the input enthalpy. Otherwise, the total temperature is calculated from the input enthalpy, heat of formation, and the total fluid composition.

```cpp
void setTotal_hS(real h, real S)
```

Sets total conditions based on the input enthalpy and entropy.

Common to all thermo packages:

- Sets the following parameters:
- Sets the total enthalpy to the input value.
- Status of static values = NO
- A total pressure solver is used to determine the pressure that will match the input h and S. This results in setting Tt and Pt.

Additional actions specific to thermo package allFuel

Status of static values = NO

The following are input parameters to the allFuel calculation function:
- h2 = ht = hin
- h = 1.0
- air = fracAir
- FAR = FAR*air
- war = WAR*air
- p = exp((0.24*log(hin/0.24) – Sin + 0.279)/0.06855)
- oar = har = 0.0
- herror = 0.1
- plog = log(p)
- entropy = s = Sin

The AllFuel calculation function:

```cpp
thofsc_(&s, &plog, &t, &h, &cs, &r, (char*)_jphase.data(), allFuelFlowStation::_tfuel,&FAR, &war, &air, &har, &oar)
```

Output parameters from the allFuel calculation function:
- entropy = s
- Tt = t
- Pt = exp(plog)
- Rt = r
- gamt = cs/(cs - r)
- entropy = s
- Cp = cs
- rhot = _Pt * 144 / (Rt * Tt * C_BTUtoFT_LBF )

Additional actions specific to thermo package FPT:

Sets the following FlowStation parameters:
Sets the total enthalpy to the input value.
Sets the entropy to the input value
Status of static values = NO

If the total pressure or total temperature have not been previously defined, they are set to STP values.

A total pressure solver is used to determine the pressure that will match the input enthalpy. The function setTotalSP(s, Pt) is called which will set the entropy, total pressure, total enthalpy, and total temperature.

**void setTotal_hPsV()**
Sets the thermodynamic state based on static enthalpy, static pressure, and velocity (Mach Number).

Actions specific to thermo package allFuel:

Calculates the following FlowStation parameters:
Sets the state using the given enthalpy and pressure: setTotal_hP( hin, Pin )
Calculates the enthalpy: ht = hin + 1./ 2. * get_rhot()/ C_GRAVITY / C_BTUtoFT_LBF * pow( Vf, 2 )
gams = get_gamt()
A total pressure solver is used to determine the Pt using the given static pressure.

Actions specific to thermo package CEA:

Calculates the following FlowStation parameters:
Sets the state using the given enthalpy and pressure: setTotal_hP( hin, Pin )
Calculates the enthalpy: ht = hin + 1./ 2. * get_rhot()/ C_GRAVITY / C_BTUtoFT_LBF * pow( Vf, 2 )
gams = get_gamt()
A total pressure solver is used to determine the Pt using the given static pressure.

Actions specific to thermo package FPT:

Calculates the following FlowStation parameters:
Sets the state using the given enthalpy and pressure: setTotal_hP( hin, Pin )
Calculates the enthalpy: ht = hin + 1./ 2. * fluidPtr->get_rhot(this)/ C_GRAVITY / C_BTUtoFT_LBF * pow( Vf, 2 )
gams = fluidPtr->get_gamt(this)
A total pressure solver is used to determine the Pt using the given static pressure.

Actions specific to thermo package Janaf:

Calculates the following FlowStation parameters:
Sets the state using the given enthalpy and pressure: setTotal_hP( hin, Pin )
Calculates the enthalpy: ht = hin + 1./ 2. * fluidPtr->get_rhot(this)/ C_GRAVITY / C_BTUtoFT_LBF * pow( Vf, 2 )
gams = fluidPtr->get_gamt(this)
Rs = get_Rt()
MN = Vf / sqrt( C_BTUtoFT_LBF * C_GRAVITY * gams * Rs * Ts )
A total pressure solver is used to determine the Pt using the given static pressure.

**void setTotalSP(real S, real P)**
Sets total conditions based on the input entropy and pressure.

Common to all thermo packages:

Sets the following FlowStation parameters:
Sets the FlowStation entropy to the input value.
Sets the FlowStation total pressure to the input value
Status of static values = NO
Actions specific to thermo package allFuel:

The following are input parameters to the allFuel calculation function:
- \( Pt = Pin \)
- \( entropy = s = Sin \)
- \( oar = har = 0.0 \)
- \( air = frac\text{Air} \)
- \( plog = (\text{float})\log(Pt) \)
- \( FAR = FAR*air \)
- \( war = WAR*air \)
- \( r = Rt \)

The AllFuel calculation function:
\[
\text{thofsc}(_{&s, &plog, &t, &h, &cs, &r, (\text{char})*_\text{jphase.data}()}, \text{allFuelFlowStation}::_{_ftfuel, &FAR, &war, &air, \&har, \&oar})
\]

Output parameters from the allFuel calculation function:
- \( Tt = t \)
- \( Rt = r \)
- \( ht = h \)
- \( gamt = cs/(cs - r) \)
- \( entropy = s \)
- \( Cp = cs \)
- \( rhot = Pt * 144 / (Rt * Tt * \text{C\_BTUtoFT\_LBF}) \)

Actions specific to thermo package CEA:

Sets the following FlowStation parameters:
- Sets total pressure to the input pressure
- Sets the entropy to the input value
- Status of static values = NO
- Calculates the total temperature: \( Tt = \text{pow} ((Pin / Pt), (gamt - 1)/gamt) * Tt \)

Calls CEA setting the state conditions to S and P:
\[
\text{startceasp}(_{\text{type}, \text{Pt}, \text{Stemp}, \text{Wreac1}, \_\text{Wreac2}, \_\text{Wreac3}, \_\text{Wreac4}, \_\text{Wreac5}, \_\text{Wreac6}, \_\text{Wreac7}, \_\text{Wreac8}, \_\text{Wreac9}, \_\text{Wreac10}, \text{Aratio}, \text{rhot}_r, \text{ht}_r, \text{Tt}_r, \text{gamt}_r, \text{Vson}_r, \_\text{print}, \_\text{transport}, \text{mu}_r, \text{k}_r, \text{Cp}_r, \text{Pr}_r, \text{R}_r, \_\text{error})
\]

Conversions are then done:
- \( \text{mut} = \text{mu}_r*6.7197e-5 \)
- \( \text{rhot} = \text{rhot}_r \times 0.0022046 / 0.00035315 \)
- \( \text{ht} = \text{ht}_r \times 1.798796 \)
- \( \text{Cpt} = \text{Cp}_r \)
- \( \text{kt} = \text{k}_r \times 1e-3 \times 0.0671519 \)
- \( \text{Prt} = \text{Pr}_r \)
- \( \text{Tt} = \text{Tt}_r \times 9 / 5 \)
- \( \text{Rt} = \text{R}_r \)
- \( \text{gamt} = \text{gamt}_r \)

Actions specific to thermo package FPT:

Sets the following FlowStation parameters:
Sets the entropy to the input value
Sets the total pressure to the input value
Gets the value of the total enthalpy as a function of \((s, Pt)\) though a call to \texttt{get\_h\_st(this)}
Gets the value of the total temperature as a function of \((ht, Pt)\) through a call to \texttt{get\_T\_ht(this)}
Status of static values = NO

Actions specific to thermo package Janaf:

Sets the following FlowStation parameters:
Sets the total pressure to the input value
\(Rt = \texttt{get\_Rt()}\)
\(phiT = \text{entropyin} + Rt \times \log(Pt)\)
Calculates \(Tt = \text{phi\_to\_t(\ phiT, kGasTotal )}\)
Calculates \(ht = t\_to\_h_\ ( Tt, kGasTotal ) + \text{formation\_heat( kGasTotal )}\)
Status of static values = NO

Recalculates these parameters if FAR > 0 and reconstitute != FROZEN through a call to:
\texttt{chem\_equilibrium(\ k\_type, Pin, q\_dum1, Stemp, q\_t\_react, q\_lost\_h, kGasTotal )}
\(Rt = \texttt{get\_Rt()}\)
\(phiT = \text{entropyin} + rt \times \log(Pt)\)
\(Tt = q\_t\_react\)
\(ht = t\_to\_h_\ ( Tt, kGasTotal ) + \text{formation\_heat( kGasTotal )}\)

\textbf{void setTotalStaticTP(real Ts, real Ps, real MN)}
Sets total and static conditions based on the input static temperature, static pressure, and Mach number.

Actions specific to thermo package Janaf:

Sets the following FlowStation parameters:
Calls \texttt{setMNDes(MNin)}
Sets the static temperature to the input value
Sets the static pressure to the input value
Calls \texttt{setTotalTP( Tin, Pin )}
\(hs = ht\)
\(gams = \texttt{get\_gamt()}\)
Calculates total enthalpy: \(ht = hs + MNin \times MNin \times \texttt{get\_Rt()} \times gams \times Tin / 2.0\)
Calculates the total pressure: \(Pt = Pin \times pow((1.0 + (gams-1)/2 \times MNin \times MNin), gams/(gams-1))\)
Calls \texttt{setTotal\_hP(ht, Pt)}
MachOrArea = MACH
Status of static values = NO
\(MN = MNin\)
Calls \texttt{setStatic()}\n
\textbf{void setTotalTP(real Tt, real Pt)}
Sets total conditions based on temperature and pressure.

Common to all thermo packages:

Sets the following FlowStation parameters:
Sets the total pressure to the input value
Sets the total temperature to the input value
Sets the total enthalpy through the call: \(ht = \texttt{get\_h(Tin, Pin)}\)
Status of static values = NO

Actions specific to thermo package allFuel:

Sets the following FlowStation parameters:
Sets the total pressure to the input value
Sets the total temperature to the input value
Status of static values = NO

The following are input parameters to the allFuel calculation function:
air = fracAir
FAR = FAR*air
war = WAR*air
oar = har = 0.0
plog = (float)log(Pin)
ts = Tin

The AllFuel calculation function:
thofc_(&ts, &plog, &h, &cs, &s, &r, (char*)jphase.data(), allFuelFlowStation::_tfuel, &FAR, &war, &air,
&har, &oar)

Output parameters from the allFuel calculation function:
ht = h
Rt = r
gamt = cs/(cs - r)
entropy = s
Cp = cs
rhot = Pt * 144 / ( Rt * Tt * C_BTUtoFT_LBF )

Actions specific to thermo package CEA:

Sets the following parameters:
Sets the total temperature to the input value
Sets the total pressure to the input value
Status of static values = NO

Calls CEA to calculate several parameters:
startceatp_( type, Pt, Tt, _Wreac1, _Wreac2, _Wreac3, _Wreac4, _Wreac5, _Wreac6, _Wreac7, _Wreac8,
_Wreac9, _Wreac10, dum, rhot_r, ht_r, S_r, gamt_r, _print, _transport, mu_r, k_r, Cp_r, Pr_r, R_r, _error )

Calculates and sets FlowStation parameters:
rhot = rhot_r *.0022046 / .000035315
ht = ht_r * 1.798796
entropy = S_r
gamt = gamt_r
mut = mu_r*6.7197e-5
Cpt = Cp_r
kt = k_r * 1e-3*.0671519
Rt = R_r
Prt = Pr_r

Actions specific to thermo package FPT:

Sets the following FlowStation parameters:
Sets the total pressure to the input value
Sets the total temperature to the input value
Calculates total enthalpy: ht = get_h_Tt(this)
Calculates entropy = get_s(this)
Status of static values = NO
Actions specific to thermo package Janaf:

Sets the following FlowStation parameters:
- Sets the total temperature to the input value
- Sets the total pressure to the input value
- Calculates total enthalpy: \( h_t = t_{to\_h}(T_t, k_{GasTotal}) + \text{formation\_heat}(k_{GasTotal}) \)
- Status of static values = NO

If \( FAR > 0 \) and reconstitute != FROZEN, recomputed the composition:
\[
\text{chem\_equilibrium}(k_{type}, _{Pt}, q_{dum1}, q_{dum2}, q_{t\_react}, q_{lost\_h}, k_{GasTotal})
\]
\[
h_t = t_{to\_h}(T_t, k_{GasTotal}) + \text{formation\_heat}(k_{GasTotal})
\]

\textbf{void setTotal\_u\_rho()}
Set total conditions based on internal energy and density

Common to all thermo packages:
- Sets up a total pressure solver to calculate \( P_t \) based on \( \rho \).
- Sets up a total enthalpy solver to calculate \( h_t \) based on internal energy.
- Calls: setTotal\_hP(\( h_t, P_t \))

Actions specific to thermo package allFuel:
- Sets up a total pressure solver to calculate \( P_t \) based on \( \rho \).
- Sets up a total enthalpy solver to calculate \( h_t \) based on internal energy.

\textbf{real getTotalComp("H2")}
Returns the mass fraction of the specified species (in CEA notation).

\textbf{void addEq("H2", 0.5)}
Adds a specified mass flow (lbm/s) of a specified species (in CEA notation) within the flow and then recalculates equilibrium. The species must be one of the six called in the "setThermoPackage" command.

\textbf{void subtractEq("H2", 0.5)}
Subtracts a specified mass flow (lbm/s) of a specified species (in CEA notation) within the flow and then recalculates equilibrium. The species must be one of the six called in the "setThermoPackage" command.

\textbf{void addNonEq("H2", 0.5)}
Adds a specified mass flow (lbm/s) of a specified species (in CEA notation) within the flow but does not recalculate equilibrium. The species must be one of the six called in the "setThermoPackage" command.

\textbf{void subtractNonEq("H2", 0.5)}
Subtracts a specified mass flow (lbm/s) of a specified species (in CEA notation) within the flow but does not recalculate equilibrium. The species must be one of the six called in the "setThermoPackage" command.

12.1.11 Independent Functions

\textbf{void revertToDefaults()}
Unlocks all "lockable" attributes in this independent.

12.1.12 InFileStream Functions

\textbf{void close()}
Closes the file.
int fileExists(string filename) Returns TRUE if the specified file exists.

int getByte() 
Reads a single character from the stream and returns it as an integer.

string getc() 
Reads a single character from the stream and returns it as a string (The NPSS does not have a char data type).

string getline([int expected]) 
On formatted streams, reads from the stream until the next newline character. For binary or unformatted streams, this just calls getString(). The optional expected argument is only used for binary or unformatted streams; it is ignored when used with a formatted stream.

int getInt([int width]) 
Reads an integer from the stream. If specified, at most width characters are read.

int getInt1D(string varName [, int expected [, int start]])
Reads an integer array from the stream into the variable. For formatted and binary streams, the expected argument is required and specifies how many values are to be read. For unformatted streams, expected is optional, and if > 0, it is checked against the read record length. The start argument is always optional. If specified, it is the starting index in the array variable for values to be stored. If start is not specified, the first value read is stored at index 0.

real getReal([int width]) 
Reads a real number from the stream. If specified, at most width characters are read.

string getString([int expected])
Reads a string from the stream. For formatted streams, the optional expected argument is ignored. For binary streams, expected is a required argument. If expected is zero, characters are read until a terminating zero. If expected is >0, that many characters are read. For unformatted streams, the expected argument is optional. If specified and >0, it is compared against the read record length.

void open(string filename) 
Opens a file, just as if assigning a function name. If you do not use the close() function to close an open file, the file will be closed automatically when you exit NPSS.

void reopen() 
This convenience function remembers the file name associated with an open InFileStream, closes the stream, then re-opens the stream using the memorized file name. It essentially rewinds an input file.

string peekc() 
Returns the next character which would be returned by getc() without actually removing it from the stream.

void ungetc(string chars)
Pushes one or more characters back onto the stream. These characters will only be seen by getc() or peekc().

12.1.13 Linear Model Generator Functions
See Section 8.3 for the Linear Model Generator member functions.

12.1.14 Link Functions

string getPlName() 
Returns the name of the first of two ports held together by this link.
string getP2Name()
Returns the name of the second of two ports held together by this link.

### 12.1.15 msgHandler Functions

**Note:** These functions have not been removed from the system but are OBSOLETE. Please see Section 4.14, Error Handling and Messages, for the latest information, including preferred functions.

void clear()
Clears all of the buffered errors, warnings, and informational messages and all provisionals.

void error(string errMessage, int ESI)
Generates an error message.

void message(string message, int showFullMessage)
Generates an informational message and optionally shows the file and line number information.

void provisionalError(string errMessage, int ESI)
Generates a provisional error message.

void throwError(string errMessage, int ESIcat)
Breaks flow of execution and generates an error message. If throwError() is called from interpreted code, then execution will resume on the next statement in the interpretive code.

void warning(string warningMessage, int ESI)
Generates a warning message.

### 12.1.16 OutFileStream Functions

To create an OutFileStream:

```
OutFileStream theFileOut;
```

void close()
Writes the file and then closes it. Using this function allows the closed file to be seen from other files.

int fileExists(string filename)
Returns TRUE if the specified file exists.

void flush()
Writes any pending output to the file, "flushing" the internal buffer. Typically unnecessary, as all output is flushed upon closing, or upon each newline if autoFlush is active (the default).

void open(string filename)
Opens a file, just like assigning a function name. Explicitly opened streams always create a file, even if it is never written to (resulting in a zero-length file). If you do not use the close() function to close the file, the file will automatically be closed when you exit NPSS.

void reopen()
This convenience function remembers the file name associated with an open OutFileStream, closes the stream, then re-opens the stream using the memorized file name. It essentially clears/resets an output file.

void print(any value [, string format])
Prints the given value to the stream. The optional format parameter is only used with formatted streams, and is described in the section for the global function toStr(). It is ignored with binary and unformatted streams.
void println()
Prints a newline to a formatted stream. Ignored by binary and unformatted streams.

void println(any value [, string format])
Calls print(). With formatted streams, it then prints a newline. With binary streams and a string value, writes a terminating zero to the stream. For unformatted streams, does nothing beyond print().

void printByte(int value)
Writes value directly to the stream. Note that some values can cause confusing results on a formatted stream.

void printVariable(string varName [, string format])
Calls print() with the given variable. More efficient with interpreted code and arrays, as it avoids copying the array to a temporary.

12.1.17 Port Functions

string PortName.isLinkedTo()
Returns the fullPathName of the Port that this Port is linked to. If the Port is not linked, returns NULL.

string[] getLinkName()
Returns the name(s) of the link(s) that this Port is attached to. If the Port is not linked, returns an empty array.

string[] getLinkedPortName()
Returns the name(s) of the remote Port(s) that this Port is attached to. If the Port is not linked, returns an empty array.

string getSpeedRef()
Used for ShaftInputPorts. Returns the reference speed variable name for the referenced port.

12.1.18 Socket Functions

int isEmpty()
This function is accessed from interpreted code. It returns 1 (TRUE) if the user has not instantiated a subelement to fill the given socket (i.e., the socket is "empty"), and returns 0 (FALSE) if the user has instantiated a subelement (i.e., the socket is NOT empty).

12.1.19 Solver Functions
Please see Section 16.1.2 for Solver member functions. Advanced Solver functions can be found in Section 16.6.

12.1.20 Table Functions

void applyIndepMods()
This function takes a table's modifiers (scalar and adder) and applies them to each table value. The original values are then overwritten with new scaled values. Each subtable in turn has its own modifiers applied. Finally, the scalar and adder are reset to their default values of 1 and 0, respectively.

string display()
Writes the Table definition into a string. If desired, the Table definition string can be saved to a file and read back into NPSS at a later time. The definition contains all of the Table's independent and dependent values, as well as any Table attributes that differ from default values.

real eval(real v1, real v2, ..., real vn)
Performs a table lookup and returns the resulting real value.
**Real[] evalStacked(real v1, real v2, ..., real vn)**

Performs a table lookup on a stacked table, i.e., a table with multiple dependents, and returns an array of real values.

**Real evalYX(real v1, real v2, ... , real vn)**

Performs an inverse table lookup, i.e., the table dependent becomes the independent and the lowest level independent becomes an independent. Note that evalXY will produce an error if the table dependents are not monotonic.

**Real evalYXiter(real v1, real v2, . . . real vn)**

Performs an inverse table lookup while iterating on the independent value to force the dependent value to match. The values from evalYX() will not give the same independent when put back into the table while evalYXiter() (described above) will. Note that evalXY will produce an error if the table dependents are not monotonic.

**String[] getDependentNames()**

Returns a string array containing the names of all of the dependents in the Table. Typically there will only be one name in the array unless the Table is stacked.

**String[] getIndependentNames()**

Returns a string array containing the names of all of the independents in the Table. The names appear in order from outermost to innermost subtable, which is not necessarily the same order as the argument list in the Table definition.

**Int getIndependentCount()**

Returns the number of independents in the Table.

**Void getIndependentArgNames(string names[])**

Writes the names of all the Independent arguments in the Table definition, into the string array, names[]. The Independent argument names appear in the same order as the argument list in the Table definition.

**Int getIndependentArgCount()**

Returns the count (or total number) of Independent arguments in the Table definition.

**Int getNumberDependents()**

Returns the number of dependents in the Table.

**Int getNumberIndependents()**

Returns the number of independents in the Table.

**Real[] getValues(string name, indep1, ..., indepN)**

Returns a real array containing the values of the requested independent or dependent. The first argument is the name of the requested independent or dependent, followed by a number of independent values that is sufficient to specify which set of independent or dependent values are desired. For example, for a Table containing three independents { "z", "y", "x" } where z is the outermost subtable, the independent "z" values can be retrieved in the following way:

```c
real vals[] = tableName.getValues("z");  // no independent args needed here
```

To retrieve a set of "x" independent values, two independent values must be supplied in order to specify which "z" subtable and which "y" subtable to retrieve the "x" values from. For example:

```c
vals = tableName.getValues("x",1.3,4.1);  // two independent args needed
```

Since the dependent values live in the innermost subtable, the same number of independent arguments is needed to retrieve them as is needed to retrieve the innermost independent. In this example, "x" is the innermost independent, so in order to retrieve a set of dependents (called "dep" in this example), you need two independent values to specify which dependent set that you want. For example:
```java
depVals = tableName.getValues("dep",1.3,4.1); // two indep. args needed
```

```java
string getTextRep(int saveLevel)
```

Returns a string containing an ASCII text representation of the current table. Parameter `saveLevel` can be set to either 1 or 0. A `saveLevel` of 1 gives a full save, delving into each level of the table and the variables contained within. A `saveLevel` of 0 saves a more skeletal state representation.

### 12.1.21 Time Discrete Model Objects

Section 17.4 contains the functions for time discrete model objects, which are for advanced users.

### 12.1.22 Transient Executive Functions

Section 17.1.7 contains transient executive functions and a usage example.

### 12.1.23 User-Supplied Subelement Functions

```java
NCPVal getValueFromParent (string name)
```

Returns the value of the named variable in the parent element or subelement.

```java
void setValueInParent (string name, NCPVal value)
```

Sets the named variable in the parent to the given value.

### 12.1.24 VariableContainer Functions

```java
void dump(int recurse, string wildcard)
```

This function writes all of the selected variables and their values to the standard output. If a wildcard string is supplied, only variables with pathnames that match the wildcard will be displayed. All output is sorted alphabetically and the names displayed are the pathnames relative to the starting scope. The “recurse” argument gives the recursion depth to report. This ranges from 0 to the number of levels to be examined. The “wildcard” string may be used for filtering reported information.

```java
void loadCompiledObjects(string DLMname,string objectname="",int absolute=0)
```

This function can be called from any VariableContainer and will allow specific function objects or other VCInterface object instances created in the DLM to be registered in that VariableContainer. For example, given an object called foo, calling `foo.loadCompiledObjects("myfuncDLM")` will load whatever objects are created in myfuncDLM.sc (UNIX) or myfuncDLM.dll (Windows). Note that this is different from a normal component DLM, which loads a CLASS. This type of DLM loads an instance or instances of class into a specific scope.

If the DLM supports it, a specific object can be loaded from the DLM if `objectname` is not empty. Table DLMs support this option.

If the optional argument `absolute` is passed as a non-zero value, the `DLMname` is taken literally as a filename.

Please refer to the `NPSS Reference Sheets`, specifically the VariableContainer Reference Sheet, for additional VariableContainer functions.

### 12.2 Global Functions

*Global functions* apply to the NPSS as a whole and are not associated with any particular object.

### 12.2.1 Bitwise Functions

Note: See Section 2.2.5.1 for bitwise *operators*. 
int bitget(int A, int B)
Returns the Bth bit (0 to 31) of A.

A = 1089
A = 0000 0000 0000 0000 0000 0100 0100 0001

bitget(A, 6) = 1
bitget(A, 8) = 0

int bitset(int A, int B, int C)
Sets the Bth bit (0 to 31) of A to C (0 or 1) and returns the result.

bitset(A, 12, 1) = 5185
0000 0000 0000 0000 0001 0100 0100 0001

int shiftL16(int A, int B)
Shifts the bits of A to the left by B places and returns the result with the 16 MSB masked out.

shiftL16(A, 7) = 8320
0000 0000 0000 0000 0010 0000 1000 0000

int shiftL32(int A, int B)
Shifts the bits of A to the left by B places and returns the result.

shiftL32(A, 7) = 139392
0000 0000 0000 0000 0100 0000 1000 0000

int shiftR16(int A, int B)
Shifts the bits of A to the right by B places and returns the result with the 16 MSB masked out.

A = 139393
A = 0000 0000 0000 0010 0010 0000 1000 0001

shiftR16(A, 1) = 4160
0000 0000 0000 0000 0100 0000 1000 0000

int shiftR32(int A, int B)
Shifts the bits of A to the right by B places and returns the result.

shiftR32(A, 1) = 69696
0000 0000 0000 0000 0001 0001 0001 0000 0100 0000

12.2.2 ESORegistry Functions
Please see Section 4.14.5 for ESORegistry functions, which are global functions. That section is a subsection of 4.14, Error Handling and Messages.

12.2.3 Units Functions
Please see Section 11.1 for units functions.
12.2.4 Miscellaneous Global Functions

*Functions which interface with the Operating System:*

```c
int chdir(string path)
Changes current directory to path then returns the exit code of the function
```

```c
string getcwd()
Returns current directory.
```

```c
string getenv(string environment_name)
Returns a string value for the environment name given.
```

```c
int getpid()
Returns process ID.
```

```c
int randomInt([int lowerBound,] int upperBound)
Returns a random integer from lowerBound to upperBound. The single-argument form returns an integer between zero and upperBound, useful for indexing an array.
```

```c
real randomReal([real lowerBound, real upperBound, int closed])
Returns a random real number from lowerBound to upperBound. If closed is non-zero, the upper limit of the interval is closed. The default no-argument form returns a real number in [0..1).
```

```c
void randomSeed([int seed])
Seeds the random number generator. It is not necessary for this to be called, the system will provide a constant seed value. If no seed argument is supplied the random number generator will be seeded based on the system’s time() function.
```

```c
int system(string shellCommand)
Executes the given string as a shell command and returns the exit code of the command.
```

*Functions which interface with NPSS:*

```c
string[] displayAliasTable()
Returns a 1-D string array containing the lines of a table showing all the alias names in the model with their corresponding model names. Note, this does not currently check security or access, so using this will allow the user to see anything that is in this table. It doesn't give them access to the objects, just the display of the names.
```

```c
string getAlias(string name [,errFlag])
Returns the alias for the object name. If there is none, a null string is returned. If the optional errFlag variable is set to 1, an error message will also be displayed. See the NPSS Developer's Guide for more information.
```

```c
string getModelName(string alias)
Returns the real model name for alias.
```

```c
any getVal(string string_ref_name)
Returns the contents of the string ref variable or attribute, if given.
```

```c
int isNaN(real val)
Boolean function which returns 1 (TRUE) if the given val is a NaN or 0 (FALSE) otherwise.
```

```c
int isInf(real val)
Boolean function which returns 1 (TRUE) if the given val is infinite or 0 (FALSE) otherwise.
```
void NPSSTerminationSequence()
A flexible approach to shutdown is the implementation of a mechanism in which NPSS executes a user-definable
sequence of objects upon termination. This sequence is called NPSSTerminationSequence (see section
2.2.4.10, Automatically Created Global Variables) and is a global string array that may contain a combination of
objects, no-argument functions, and viewers that the user would like to execute upon shutdown. Using this type of
structure allows the sequence to be built over time and insures that all termination requests are honored.

void parseEfile(string fileName [, string keyFile])
Reads in and executes the ENCRYPTED file specified by fileName. An encryption key may be provided in a
keyFile that NPSS can use to correctly decrypt the file before parsing (i.e., the same key used for encryption). This
key is generated in the CDgen program and is mainly for use with Customer Decks. If no keyFile is provided,
NPSS will look for the key in a global variable, which gets set by CDgen program. If the global variable is not set
and no keyFile is provided, an error message is generated and execution is halted.

void parseFile(string fileName)
Reads in and executes the file specified by the user.

void parseString(string stringName)
Reads in and executes the string defined by the user.

void setAlias(string name, string alias [,errFlag])
Creates an alias for name. The alias will have its access type set equal to the access type of the real variable name.
Aliases can have their access type set independently of the real variable. If the alias is given "RW" permission and
the real variable has "PRIV" access, the actual variable can be seen, but not written to, through the alias name. If
the optional errFlag variable is set to 1 and the variable name does not exist, a warning message will be displayed.
Care should be taken when setting an alias' access type different from that of the original variable. See NPSS
Developer's Guide for more information.

void setVal(string string_ref_name, any value)
Assigns the string ref name or attribute, if given.

int traceExecution(int onOff)
Controls execution tracing, as described by the trace command. Returns the previous on/off state.

void quit()
Exits the NPSS instance.

void setFlowFuelStablePointers(int flag)
Set flag to 0 (default) if flow station names and types are changing. Set flag to 1 for faster processing (station
name only resolved once) if station names and types are constant

Functions which Query the DataManager:

int canCreate(string Type)
Boolean function returns 1 (TRUE) if the executable can create the given component type, or 0 (FALSE) if it
cannot. Since CMFs (Creation Method Facilities) can dynamically bring in components, a user may be able to
create a component type that was not specifically mentioned in the output of listKnownTypes().

string[] findResource(string resource)
Returns a string that indicates where the DataManager hopes to get the resource when asked to load it or create it.
Resources are usually a component name, but they do not have to be. Traditionally the DataManager will either
check its internal stores to see if it knows about the object, or it will ask the CMFs if they can bring this resource
into the system. If the resource is in the DataManager's internal store, findResource will return <Internally
linked> if it is a compiled component or an already loaded DLM. For interpreted components that it knows about,
findResource returns the string <Internally cloned>. For CMFs, if a specific CMF can load the resource, look at that CMF’s documentation to see the data returned. In particular, DCLOD and ICLOD return the path to the item they would load, such as /NPSS/dev/AirBreathing/DLMComponents/sun/Duct.sc for DCLOD, or /NPSS/dev/AirBreathing/InterpComponents/Duct.int for ICLOD. In the future, resources will follow the standard Universal Resource format (URL).

string[] listKnownTypes(string base, int verbose)
Returns a list of all object types that NPSS is capable of creating either internally or through ICLOD or DCLOD (ECLOD not supported yet). The "base" argument (which defaults to "") specifies a base class, allowing the user to list all known Elements or Subelements, for example. The "verbose" flag defaults to 0, causing the function to return just a list of type names. If "verbose" is not 0, then each entry in the returned string array will contain the type name, a list of interfaces for that type, and a list of CMFs that can create that type, for example:

typeName^iface1,iface2,...iface_n^CMF1,CMF2,...CMF_n

string[] listLoadedCMFs()
Returns the names of all the loaded (active) CMFs. To display the list you will probably want to send it to an output stream, for example:

cout<<listLoadedCMFs()<<endl;

string[] listThermoPackages()
Returns a list of available thermodynamic package names as long as meta data has been supplied defining them. To display the list, you must send it to an output stream. (See Section 3.6 on metadata.)

string[] listTypes(string baseType)
Returns a list of available types for the given base type. The list of available base types are: "Assembly," "DataCollector," "DataViewer," "Element," and "Subelement." To display the list, you must send it to an output stream, for example:

cout << listTypes ("Subelement").

int loadDLM (string dlmName,[string objectName])
Loads the given DLM. Returns 1 (TRUE) if the DLM was successfully loaded. 0 (FALSE) if not. Since there is some uncertainty in properly initializing a DLM, this function could return 1 (TRUE) and the DLM may not have fully activated itself. The loadDLM command is intended for use with a path that specifies where the resource can be found. The second argument is an optional argument that explicitly states the object name to load instead of having the binary figure out the object name based on the file name. The path can be defined in one of the following three ways:

Absolute Path (searches from the top of the directory tree):

    loadDLM("/NPSS/dev/ThermoPackages/DLMComponents/sun/GasTbl.sc");

Relative Path (searches from current directory):

    loadDLM("./sun/GasTbl.sc");

Default Path (searches paths defined in LD_LIBRARY_PATH):

    loadDLM("GasTbl.sc");

Explicitly stating the object name:

    loadDLM("/lib/object1.dll","SpecialCompressor");

int loadResource(string resource)
Returns 1 (TRUE) if the resource was successfully loaded; throws an exception if it cannot be loaded. The loadResource command is generally more convenient to use than loadDLM. It combines a findResource with a loadDLM. In other words, it finds the sc file for you and loads it, without needing the absolute path. Keep in mind that loadResource checks for DLMs first, so if there is a DLM version you will get that BEFORE an int version. The equivalent loadResource call to the loadDLM (described immediately above) is:

    loadResource("GasTbl");
While `loadResource` is clearly easier, if you need to specify where the resource is coming from, then use `loadDLM`. If you use `loadResource` and are worried about where the resource came from, use the `findResource` command. The command `findResource("whatever")` will return the full path of where the resource got its definition, or it will return `<Internally Linked>` if it was compiled in.

**Security related functions:**

```cpp
tvoid activateSecureMode()
{ Returns the permission status for the object `name` (either "RW", "RO", or "PRIV"). See NPSS Developer's Guide for more information.

```cpp
tstring getAccess(string name)
{ Returns the current value of the static variable `defaultAccess` (either "RW", "RO", or "PRIV"). See NPSS Developer's Guide for more information.

```cpp
tstring getDefaultAccess()
{ Returns the current value of the static variable `defaultAccess` (either "RW", "RO", or "PRIV"). See NPSS Developer's Guide for more information.

```cpp
tvoid setAccess (string name, string access, int recurse)
{ Allows users to set a variable’s permission status to Read-Write, Read-Only, or Private. It supports both wildcarding and recursion. The first parameter, `name`, is the name of the object that you want to affect. If this name is a wildcard, all objects in the model which match the wildcard will be affected from the level at which the function is called and below. Allowable wildcard characters are the question mark (?) and the asterisk (*). In the absence of a wildcard, if the `recursion` parameter is set to 0, only the current object will be affected. If it is set to 1, all objects below the current object will also be affected. This action, of course, depends on the accessibility of the calling object to lower level objects. The object name can also be an alias, which will be treated as any other variable. See the NPSS Developer's Guide for more information.

```cpp
tvoid setAccessForLink (string linkName, string access)
{ The `setAccess()` function described above does not work for links (a.k.a., stations). The `setAccessForLink()` function allows users to set a link’s permission status to Read-Write, Read-Only, or Private ("RW", "RO", or "PRIV"). The first parameter is the link’s name. Recursion is handled internally as it applies to links. See the NPSS Developer's Guide for more information.

```cpp
tvoid setDefaultAccess(string access)
{ Sets the static attribute `defaultAccess` (either "RW", "RO", or "PRIV"). All objects created after this will have their `access` attribute set to this value. See the NPSS Developer's Guide for more information.

**Other Global Functions:**

```cpp
tint isRealValid(NCPReal x)
{ Tests the validity of a real number; it checks for Not a Number (NaN) and +/- infinity. If the number is a valid real number, the function returns a 1; otherwise the function returns zero.

**12.3 Interactive Mode Functions**

For more information on interactive mode, see Section 2.3.

```cpp
tvoid debug(int flag)
{ Activates or deactivates interactive debug mode based on the value of flag. If flag=TRUE, debug mode is activated. If flag=FALSE, debug mode is deactivated. The value of flag defaults to TRUE, so calling
debug() has the same effect as calling debug(TRUE). When debug is inactive, all breakPoints will be ignored and the where() command will not return the function call stack.

**icl.preBreak()**
This function executes every time an active breakPoint is encountered, but before the action specified in the breakPoint command is executed. It will also execute after a step, stepOver, or stepOut command has been issued.

**icl.preexecute()**
Executes once at the start of the interactive mode session. This function is similar to the element/subelement preexecute() functions.

**icl.postBreak()**
This function executes after a step, stepOver, stepOut or continue command has been issued.

**icl.start()**
Starts interactive mode execution at the point in the NPSS input file where the user enters this command.

### 12.4 Macros

The NPSS interpretive code capability allows users to create utilities that can perform repetitive tasks and/or augment model calculations. These utilities are functions, or suites of functions known as "macros."

As previously discussed in the syntax section on functions (see Section 2.2.6), one of the fundamental uses of a function is to perform repetitive calculations. A very powerful tool for this is the indirect membership operator ":>" (see Section 2.2.6.5). To reiterate the usefulness of this tool, see how it is used in the printData function below. The printData function will print all of the names and values for a list of variables passed into it via a string array.

```c
void printData(string Names[]) {
    int i;
    for (i=0; i<Names.entries(); i++) {
        // print each Name and its value
        cout << Names[i] << " = " << Names[i]->value << endl;
    }
    cout << endl;
}
```

The following two methods show how the printData function may be used:

**Method 1** – The user creates a list of names and calls the function.

```c
string names[];
names = { "name1", "name2", "name3" }
printData( names );
```

**Method 2** – The user calls the function to list all the values in the CmpH element.

```c
printData( CmpH.list("Variable",false) );
```

Several standard NPSS functions have been developed to facilitate non-viewer-based output, solver setup, bleed definition, and model execution. Summaries for these standard NPSS functions are contained in the following subsections. To access these NPSS standard function files, use #include in the input file. For example, to include the print functions:

```c
#include <print_macros.fnc>
```
12.4.1 Global Macro Functions

void help()
Prints to the display the name, arguments and descriptions of any user functions in memory that have descriptions available.

void man()
Same as help() function above. Prints to the display the name, arguments and descriptions of any user functions in memory that have descriptions available.

12.4.2 Bleed Macro

File bleed_macros.fnc contains the following:

bldPumpPwr
Pump power used by the bleed impeller.

FracBldW
Fraction of the source stream flow used as bleed.

FracBldWork
Work fraction of bleed flow.

void linkBleed
Creates bleed port and link into the model.

void linkBleed(string source, string sink, real fracBldWork, real PressureFraction, real bldPumpPwr, string name)
Location (name) of bleed source.

name
Root name of bleed port.

Pfrac
Relative interstage pressure level for bleed introduction to turbine (=1, front; =0, back).

sink
Location (name) of bleed sink.

12.4.3 Print Macro

File print_macros.fnc contains the following:

void printAll(string array)
Prints array of data names, values, units, IOstatus and description

void printData(string array)
Prints array of data names and values

void printDependents(string array)
Prints only solver dependent names and values

void printDescription(string array)
Prints array of data names, values and description
void printIndependents(string array)
Prints only solver independent names and values.

void printIOstatus(string array)
Prints array of data names, values and IOstatus

void printPride(string array)
Prints solver Indep and Dep names and values

void printUnit(string array)
Prints array of data names, values and units

void printValue(string array)
Calls printData

12.4.4 Solver Macro
File solver_macros.fnc contains the following:

void restoreIndependents()
Restores the saved solver independent values to the actual model parameters.

void saveIndependents()
Saves the current solver independent names and values for later use.

12.4.5 runSequence Macro
The runSequence function sends the run() command to all objects in the input string sequence. The user must ensure that only valid object names (capable of responding to the run message) are in the sequence. In most cases the user will give one of the following pre-defined sequences as the arguments in the case input. For example,

runSequence(executive.executionSequence);

File runSequence_macros.fnc contains the following:

void runSequence(string sequence[]){
}

12.5 Block Diagram Generator

string BlockDGen (string baseType, string component)
Returns a string with the diagram for the element or subelement specified in component. The string is in ASCII characters.

#include <BlockDGen.util>
...
string baseT, comp;
string blockDiagram;
...
comp = "Turbine";
baseT = "Element";
blockDiagram = BlockDGen(baseT, comp);
cout << blockDiagram << endl;
12.6 System/Scripting Functions

The file System.fnc defines a VariableContainer instance called sys, which implements various functions for general scripting, remote host access via ssh/scp, and managing external component servers. Since these functions use various system resources, a sys.onNPSStermination() function is provided to release these resources. System.fnc defines a global onNPSStermination() function by default, which simply invokes sys.onNPSStermination(). If you need to define your own onNPSStermination() function, define the preprocessor variable SYS_NOTERMINATION before including System.fnc.

12.6.1 General Functions

These functions provide a shell-like scripting environment which can be useful for writing indirect wrappers for external codes. Unless otherwise noted, these routines return the exit status of the underlying system call, where an exit status of zero indicates success.

string sys.dirSeparator
Character used to separate components of a directory path. "\" on Windows and "/" on UNIX.

Option sys.errorResponse
Selects how to respond to errors in sys functions. May be set to RETURN (return an error code), REPORT (report the error and return an error code), or FATAL (report the error and terminate execution). The default is FATAL.

string sys.npssConfig
Value of the NPSS_CONFIG environment variable.

string sys.pathSeparator
Character used to separate components of a search path. ";" on Windows and ":" on UNIX.

string sys.tmpDir
The path to a directory for temporary files. Set from environment variables NPSS_TMP, TMP, TEMP, or just C: on Windows and from the environment variables NPSS_TMP, TMP, TMPDIR, or just /tmp on UNIX.

int sys.traceLevel
Debug tracing level, 0=>off (default).

string sys.tstamp
Convenience attribute which evaluates to date+" "+timeOfDay+" ". Typically used for time stamping of log messages.

string sys.stdin[]
Lines to be supplied to stdin during sys.doCmd() or sys.spawn().

string sys.stdout[]
Lines retrieved from stdout during sys.doCmd() or sys.spawn().

string sys.stderr[]
Lines retrieved from stderr during sys.doCmd() or sys.spawn().

string sys.whitespace
A constant string containing the whitespace characters newline, carriage-return, tab, and space. Typically used with sys.first().

int sys.append(string src, string dst)
Appends one file to another.

string sys.basename(string path, string suffix)
Returns the file portion of path, with trailing suffix removed.

```c
void sys.broadcast(string varName)
```

Sends the value of varName to all known servers.

```c
int[] sys.calcSortingPermutation(real data[])
```

Returns a permutation array which can be used to index the data array in a sorted order. The algorithm is O(n**2), so this is not appropriate for large data sets. Can be useful for generating table data in a monotonic order.

```c
int sys.cd(string path)
```

Moves to a new directory path.

```c
int sys.checkDir(string path, int create)
```

Returns TRUE if directory path exists, create if necessary and create is non-zero.

```c
int sys.checkPID(int pid)
```

Returns TRUE if process pid exists.

```c
int sys.checkPIDs(int pids[])
```

Returns TRUE if all non-negative pids exist.

```c
int sys.chgrp(string group, string path)
```

Changes file group for path. Not implemented on Windows.

```c
int sys.chmod(string permissions, string path)
```

Changes file permissions for path. Not implemented on Windows.

```c
int sys.cp(string src, string dst)
```

Copies file src to dst.

```c
int sys.cpAll(string src[], string dst)
```

Copies all src files to directory dst. Returns number of files not copied.

```c
string sys.dirname(string path)
```

Returns directory portion of path.

```c
int sys.doCmd(string cmdLine, int in, int out, int err)
```

Executes cmdLine, returning its exit status. The in, out, and err flags should be set non-zero if the corresponding sys.stdin, sys.stdout, or sys.stderr arrays should be used for stream data.

```c
real sys.et(string t1, string t2)
```

Both t1 and t2 should be timeOfDay strings or tstamp strings. Returns t2-t1 in seconds.

```c
string sys.etStr(string t1, string t2)
```

Both t1 and t2 should be timeOfDay strings or tstamp strings. Returns t2-t1 formatted as “days hh:mm:ss”.

```c
int sys.execOK(string path)
```

Returns TRUE if path can be executed. Searches $PATH if path does not begin with ‘/’.

```c
void sys.exit(int status)
```

Exits with given status code.

```c
void sys.fatalError(string msg)
```

Prints msg and then throws a fatal error.
int sys.fileExists(string path)
Returns TRUE if path exists.

int sys.fileSize(string path)
Returns size of path in bytes, or -1 if it doesn't exist.

string sys.fileModified(string fileName)
Returns the latest modification time of fileName.

int sys.first(string str1, string str2)
Returns the index in str1 of the first instance of any character in str2, or -1 if no instance was found.

string sys.getenv(string name)
Gets value of environment variable name.

int sys getpid()
Returns process ID.

int sys.groupExists(string groupName)
Returns TRUE if the specified group exists. Not implemented on Windows.

int sys.isBigEndian()
Returns TRUE if the local machine is a 'big-endian' architecture.

int sys.isDirectory(string path)
Returns TRUE if path is a directory.

int sys.isExecutable(string path)
Returns TRUE if path is executable.

int sys.isFile(string path)
Returns TRUE if path is a regular file.

int sys.isInteractive()
Returns TRUE if this is an interactive session (stdin connected to a terminal). Not implemented on Windows.

int sys.isNewer(string path1, string path2)
Returns TRUE if path1 is newer than path2. The time granularity on UNIX is one second, and currently the time granularity on Windows is one minute.

int sys.kill(int pid)
Kills process pid.

void sys.killAll(int pids[])
Kills all positive-valued pids.

int sys.killKids(int parentPID)
Kills child process(es) of parentPID. Typically used with PIDs returned by sys.spawn().

int sys.last(string str1, string str2)
Returns the index in str1 of the last instance of any character in str2, or -1 if no instance was found.

int sys.mkdir(string path)
Makes directory path, including intermediate levels.

int sys.mv(string current, string new)
Moves/renames file current to new.

```c
void sys.multicast(string varName, string servers[])  
```
Sends varName to the given servers.

```c
void sys.onNPSStermination()  
```
Cleanup resources, to be called via NPSSTerminationSequence.

```c
string sys.popd()  
```
Returns to a previously pushed directory. Returns new current directory or null.

```c
int sys.pushd(string path)  
```
Moves to new directory path, pushing current directory on an internal stack.

```c
void sys.pushErrorResponse(string newResponse)  
```
Save the current errorResponse value on a stack, and the set errorResponse to newResponse.

```c
void sys.popErrorResponse()  
```
Restore errorResponse from the last value saved by pushErrorResponse().

```c
string sys.pwd()  
```
Returns current directory path.

```c
void sys.reconfigureHashTables(real targetACL)  
```
Reconfigures all VariableContainer hash tables to the given target average chain length, reporting changes on cout.

```c
string sys.relpath()  
```
Returns path of current directory relative to $HOME. Not implemented on Windows.

```c
void sys.reportError(string prefix)  
```
Reports error via error() using data in sys.stderr. The string prefix is prepended to the error message.

```c
int sys.rm(string path)  
```
Remove path.

```c
int sys.rmdir(string path)  
```
Remove path and everything in it.

```c
void sys.setenv(string name, string value)  
```
Sets environment variable name to value.

```c
int sys.showFile(string path, string stream)  
```
Show contents of path on stream. Returns TRUE if successful.

```c
int sys.showBinaryFile(string path, string stream, int start, int stop)  
```
Show binary contents of path from byte start to byte stop on stream. If stop < start, contents are shown until end of file. Thus to show the entire file use start, stop of 0,-1. Returns TRUE if successful.

```c
void sys.sleep(int secs)  
```
Sleeps for approximately secs seconds.

```c
int sys.spawn(string cmdLine, int in, int out, int err)  
```
Spawns a subprocess executing cmdLine, returns child PID (>=0) or error status (<0). The in, out, and err flags should be set non-zero if the corresponding sys.stdin, sys.stdout, or sys.stderr arrays should be used for stream data.
int sys.system(string cmdLine)
Executes cmdLine, returning its exit status.

int sys.testReadyAllServers()
Returns TRUE if all known servers are ready.

int sys.testReadyAll(string servers[])
Returns TRUE if all of the given servers are ready.

string[] sys.testReadyAnyServer()
Returns the names of all known ready servers.

string[] sys.testReadyAny(string servers[])
Returns the names of the given servers which are ready.

string sys.translate(string s, string from, string to)
Translates all occurrences of character from in string s to character to. The character to may be null, in which case all from characters are deleted.

void sys.unsetenv(string name)
Removes the definition of environment variable name.

void sys.waitReadyAllServers()
Returns when all known servers are ready.

void sys.waitReadyAll(string servers[])
Returns when all of the given servers are ready.

string[] sys.waitReadyAnyServer()
Waits until at least one known server is ready and returns the names of all known ready servers.

string[] sys.waitReadyAny(string servers[])
Waits until at least one of the given servers is ready and returns the names of the given servers which are ready.

12.6.2 SSH/SCP Support
Access to remote hosts and filesystems can be accomplished via the ssh/scp commands. These functions have been tested under Windows using Cygwin, but only with UNIX hosts as the remote target.

string sys.sshBinDir
If non-null, this directory is where to look for the ssh related executables. This is rarely used.

string ssh.identity
If non-null, this identity file is added to the local ssh-agent. Typically used only when the user's .ssh directory is shared among multiple hosts.

int sys.ssh(string user, string host, string cmdLine, int in, int out, int err)
Executes cmdLine as user on host via ssh. The in, out, and err flags should be set non-zero if the corresponding sys.stdin, sys.stdout, or sys.stderr arrays should be used for stream data. A null user argument defaults to the local user ID. Implicitely uses the –A and –X options to ssh.

int sys.scp(string src, string dst, int recurse)
Copies src to dst via scp. If recurse is TRUE, copies an entire directory tree.
12.6.3 External Component Server Support

Attributes and functions of a ServerSpec Object

Managing external component servers can be accomplished by specifying server configuration parameters in ServerSpec objects (defined in the ServerSpec.int file), where the name of the ServerSpec object matches the name of the type of object that the server supports. A ServerSpec object contains the following attributes:

string context
NamingService context to use.

string domain
CORBASec security domain name, currently not used.

string host
Server host name. Default is null; non-null implies a remote host to be accessed via ssh.

Option hostOS
Server host operating system type, either "Unknown," "UNIX," or "Windows." Currently a remote host is assumed to be of type "UNIX." Local host type determined from sys.npssConfig. The hostOS attribute controls what form of command line is sent to the remote host, either Windows syntax or Bourne syntax.

int port
Port to be used. Default is 0, port <= 0 implies the port should be selected by the system.

string user
User ID on host. Default is $USER.

string directory
Directory to run the server in.

int factory
Set this TRUE (non-zero) if the server is a server "factory." This affects server name generation. Note: This special naming distinction for server factories is expected to go away in the future.

int foreign
Set this TRUE (non-zero) if the server is a "Foreign" server developed with the NPSS CORBA Component Development kit (CCDK). This affects server argument name generation (see below). Default FALSE.

string command
Command to run. Note that any environment variables contained within command are expanded at the remote host, not locally. This is typically what is desired but does require that those environment variables be defined whenever a new session is started on the host. In some situations, it may be appropriate to write a small shell script which sets up the environment and runs the real command.

string arguments
Arguments for the command. If foreign is true, -NPSS_name and -NPSS_nameContext will be specified automatically. If foreign is false, the -name and -nc form for these arguments is used.

```c
int timeout
Time to wait for a server to start (seconds), default 120.

int bind
Forces binding in local NamingService, default FALSE. Typically used when for some reason the server can t bind to the NamingService on its own. For example, when a server is running on a remote host which does not have a NamingService running, or the remote NamingService has not been federated with the local NamingService.
```

After a server is launched, the following attributes have valid values:

```c
int pid
The server's process ID. If the server is remote, this is the PID of the local ssh process.

string outFile
The name of the (possibly remote) output file.

string fullCmd
The full command line used to launch the server.

int kill()
Kills this server. Not normally called by user code.

int launch()
Launches this server. Not normally called by user code.

int start()
Starts this server. Returns TRUE if successful.

int up()
Checks if this server is running. Not normally called by user code.
```

**Attributes and Functions of a ProxySpec Object**

In certain situations (typically due to network firewalls), an NPSSProxy server is required on a host. Parameters used to connect to such proxies are contained within ProxySpec objects (defined in the ProxySpec.int file). A ProxySpec object contains the following attributes:

```c
string host
Server host name (implies a remote host).

string iorFile
IOR filename on host.

string context
Local NamingService context name to bind to.

int bind(string user)
Bind to local Name Service. user is used for transferring the IOR file via scp.

void unbind()
Delete the entry in the local Name Service.
```
**Attributes and Functions in System.fnc File**

The *System.fnc* file contains the following attributes and functions to manage servers described by *ServerSpec* and *ProxySpec* objects:

**string sys.contextPrefix**  
Prefix applied to NamingService entries, default $USER.

**int sys.startServer(string serverSpec)**  
Start server described by serverSpec. If serverSpec refers to a host for which a ProxySpec has been supplied, a connection to that proxy is made and it is registered with the local NamingService. Returns TRUE if the server was successfully started. Remote Windows hosts have not been tested yet.

**int sys.startServers(string serverSpecs[])**  
Start multiple servers. Returns TRUE if all servers were started.

**int sys.startAllServers()**  
Start all servers for which a ServerSpec exists. Returns TRUE if all servers were started.

**int sys.killServer(string serverSpec)**  
Kill server serverSpec. Returns TRUE if server was found and told to shut down.

**int sys.killServers(string serverSpecs[])**  
Kill multiple servers.

**int sys.killAllServers()**  
Kill all servers started by the *sys* object.

**int sys.killNPSS(string context, string name, int pid)**  
Kill specified server via the killnpss.$NPSS_CONFIG tool, or by directly killing pid if the tool fails.

**int sys.nsBind(string context, string name, string ior)**  
Bind server specified by ior to NamingService context and name via the nsbind.$NPSS_CONFIG tool.

**int sys.nsDelete(string context, string name)**  
Unbind/delete the specified NamingService entry via the nsdelete.$NPSS_CONFIG tool.

The *sys.killServer()* functions normally do not need to be invoked; *sys.killAllServers() is called within *sys.onNPSStermination()*, which is normally invoked at the end of a simulation.

An example use of these facilities is shown below. In the example situation, two servers providing Overflow and Vulcan objects are run on a host behind a firewall, thus requiring an NPSSProxy whose IOR file has been stored at /tmp/NPSSProxy.IOR. A third server is running on another remote host and provides an ANSYS object.

```c
#include "System.fnc"

// IPG access configuration.
string ipgHost     = "sharp.as.nren.nasa.gov";
string ipgProxyIOR = "/tmp/NPSSProxy.IOR";

ProxySpec ipgProxy {
    host     = ipgHost;
    iorFile  = ipgProxyIOR;
    context  = "ISTAR";
}

// Overflow configuration.
```

12 Function Summary
ServerSpec Overflow {
    host       = ipgHost;
directory  = "/t/$USER/ISTAR/overflow";
command    = "NPSSObjServer.$NPSS_CONFIG";
arguments  = "-ns " + ipgProxyIOR + " -idle 0 -t 2";
arguments += " -I$NPSS_TOP/InterpIncludes overflow.mdl";
}

// Vulcan configuration.
ServerSpec Vulcan {
    host       = ipgHost;
directory  = "/t/$USER/ISTAR/vulcan";
command    = "NPSSObjServer.$NPSS_CONFIG";
arguments  = "-ns " + ipgProxyIOR + " -idle 0 -t 2";
arguments += " -I$NPSS_TOP/InterpIncludes vulcan.mdl";
}

// ANSYS configuration.
ServerSpec ANSYS {
    host       = "predator";
directory  = "/t/$USER/ISTAR/ansys";
command    = "NPSSObjServer.$NPSS_CONFIG";
arguments  = "-noreg -idle 0 -t 2";
arguments += " -I$NPSS_TOP/InterpIncludes -I. ansys.mdl";
    bind       = TRUE;
}

sys.startAllServers();
Overflow overflow {
    ...
}
Vulcan vulcan {
    ...
}
ANSYS ansys {
    ...
}

12.7 NPSS Built-in Math Functions

The following functions may be called from within any NPSS expression:

- real abs(real)  
- real acos(real)  
- real acosh(real)  
- real asin(real)  
- real asinh(real)  
- real atan(real)  
- real atan2(real x, real y)  
- Returns a value of the arc tangent of y/x in the range of –п to п.
- real atanh(real)  
- real ceil(real)  
- real cos(real)  
- real cosh(real)  
- real exp(real)  
- real floor(real)  
- real log(real)  
- real log10(real)
12.8 NPSS String Handling Capabilities

NPSS provides several user-accessible functions for manipulating strings and string variables, and for converting other data types to their string representation. These functions are toStr(), toInt(), toReal(), index(), split(), substr(), indexing using the [] operator, and the Tokenizer class.

12.8.1 String functions

int index(string substring, [int position])
Returns the starting position in the current string where substring is found, with first position being zero. If not found, -1 is returned. position allows the user to specify an index into string at which to start search, defaulting to zero (beginning) position.

Example:

```cpp
string myStr = "Four score and seven years ago";
int s_position;
    s_position = myStr.index("s");
cout << "First occurrence of 's' is at: " << s_position << endl;
    s_position = myStr.index("s", s_position+1);
cout << "Second occurrence of 's' is at: " << s_position << endl;
    s_position = myStr.index("s", s_position+1);
cout << "Third occurrence of 's' is at: " << s_position << endl;
    s_position = myStr.index("s", s_position+1);
cout << "Fourth occurrence of 's' is at: " << s_position << endl;
```

Results:

First occurrence of 's' is at: 5
Second occurrence of 's' is at: 15
Third occurrence of 's' is at: 25
Fourth occurrence of 's' is at: -1

int reverseIndex(string substring)
Similar to the index string function, reverseIndex returns the starting position in the current string where substring is found. However, reverseIndex starts searching from the end of the string backwards. If not found, -1 is returned.

In the examples below, position 18 is the location in the test string of the last dot '.' And position 37 is the location of the last 'Cleveland'

Also worthy of note, when using index to find the location of a substring, for example ‘Cleveland’ the location of the character ‘C’ is returned. When using reverseIndex to find the location of the substring ‘Cleveland’ the location of the last character ‘d’ is returned, since we are searching backwards.
Example:

```cpp
string bigStr, littleStr;
bigStr = "North.America.Ohio.Cleveland Cleveland Ohio";

int position = bigStr.reverseIndex(".");
cout << position << endl; // 18
littleStr = bigStr.substr(position, 1);
cout << littleStr << endl; // .

position = bigStr.reverseIndex("Cleveland");
cout << position << endl; //37
littleStr = bigStr.substr(position-8, 9);
cout << littleStr << endl; //Cleveland
```

Results:

```
18
.
37
Cleveland
```

**string insertEscapes()**

Returns a string with special characters replaced with their ‘escaped’ equivalents, for example “\t” in place of the tab character.

**string removeEscapes()**

Returns a string with escaped characters replaced with their special character equivalents, for example the tab character in place of “\t”.

**int length()**

Returns the number of characters in the variable. For example,

```cpp
string text = "How long am I?";
int l = text.length(); // l = 14
```

**string operator []**

The overloaded operator [] allows access to individual characters in a string.

Example:

```cpp
string S = "Hello gorld";
cout << S.index("gorld") << endl; // prints out 6
S[6] = "w";
cout << S << endl; // prints out 'Hello world'
```

Results:

```
6
Hello world
```

**string split([string delim])**

This function returns a string array derived from the original string using either a passed in set of delimiters, or a default set (“\t\n\0”).

Example:
string test="Hello:World";
string test1="Hello World";
string test2="Requirements Review\nGather User Requirements\tWVI interpret user requirements correctly";

test.split(":");
test1.split();
test2.split("\n\t")

Results:

{"Hello","World" }
{"Hello","World" }
{"Requirements Review","Gather User Requirements","WVI interpret user requirements correctly" }

string substr(int position, int size)
This function returns the substring starting at position of length size from the original string. A size <0 implies the rest of the string.

Example:

string bigStr, littleStr;
bigStr = "The United States of America";
int position = bigStr.index("America");
cout << position << endl; // 21
littleStr = bigStr.substr(position, 7);
cout << littleStr << endl; // America

Results:

21
America

If the combination of starting position and length exceeds the size of the string, just the maximum number of characters will be returned.

Example:

string src = "012345";
cout << src.substr(2,99);

Results:

2345

int toInt(string name)
Returns the integer value of name. name is the name of a string variable that will be converted to an integer value. If name contains leading non-numeric characters, the return value will be zero. If name begins with numeric characters and non-numeric characters follow, the non-numeric characters are ignored. name can also be the name of an integer variable.

Example:

string data = "127.5";
int iData = toInt(data);
cout << iData << endl; //127

Results:
real toReal(string name)

Returns the real value of name. name is the name of a string variable that will be converted to a real value. If name contains leading non-numeric characters, the return value will be zero. If name begins with numeric characters and non-numeric characters follow, the non-numeric characters are ignored. name can also be the name of a real variable.

Example:

```cpp
string data = "127.5";
real rData = toReal(data);
cout << rData << endl; // 127.5
```

Results:

```
127.5
```

string toStr(any name[, int precision])

Returns the string representation of name. name can be the name of any type variable. The optional precision argument is used to specify the number of digits for real values, default 6.

```cpp
real T1 = 103.52;
string T1Str = toStr(T1); // places the string "103.52" into string variable
```

string toStr(any name[, string format])

Returns the string representation of name. name can be the name of any type variable. The optional format argument is used to specify the formatting required. The formatting may be specified in the style used by DataViewers, or a second style allowing greater control. The DataViewer style is assumed if the first character of format is a “?”. The second style supports the following:

```
[arrayOrder][repeatCount] basicTypeAndFormat

[showPositive][leftJustify][lineRepeat][fill][width][.precision]
```

- If the first character is “r,” 2D and 3D arrays are output in row-major (default) order. Otherwise, if the first character is “c,” 2D and 3D arrays are output in column-major (Fortran) order.
- If the next character is a digit, then digits are collected into a line repeat count. A line repeat count > zero sets the maximum number of values printed on a single line.
- The next character sets basic data type and format:
  - A,a – left-justified string
  - E – scientific notation, upper case exponent indicator
  - e – scientific notation lower case exponent indicator
  - F - fixed point notation, upper case exponent indicator
  - f – fixed point notation, lower case exponent indicator
  - G – general real formatting, upper case exponent indicator
  - g – general real formatting, lower case exponent indicator
  - I,i – decimal integer formatting
  - O,o – octal integer formatting
S,s – right justified string
X,Z – upper case hexadecimal integer formatting
x,z – lower case hexadecimal integer formatting
- If the next character is “+,” then positive numbers show a leading “+.” Otherwise, if the next character is “-,” the value is left-justified.
- The next character specifies the fill character (default “ ”). “0” is only used as a fill character if it precedes a width (see below). “.” Is only used as a fill character if formatting a non-real value.
- If the next character is a digit, digits are collected into a field width. Default field width is zero, indicating that whatever width is required is used.
- If the next character is “.”, the digits following specify the number of digits after the decimal point to be displayed.

Only the basicTypeAndFormat character is required. Some examples:

- Integer formatted as 8 uppercase hexadecimal digits with leading zeroes: “X08”
- 2D real array in Fortran order, maximum 5 values per line, scientific notation: “c5E12.5”

```cpp
string toLower()
Returns the string converted to lower case.

string toUpper()
Returns the string converted to upper case.

string trim([string c])
Returns the string with leading and trailing spaces removed. The optional c argument is used to specify a character other than ‘space’ to be removed.

string trimLeft([string c])
Returns the string with leading spaces removed. The optional c argument is used to specify a character other than “space” to be removed.

string trimRight([string c])
Returns the string with trailing spaces removed. The optional c argument is used to specify a character other than “space” to be removed.

string append(string name)
Returns the string with the string name appended to the end.

int find_first_of(string substring, [int position])
Returns the first position in the current string where substring is found. If not found, -1 is returned. Optional parameter position allows the user to specify an index into string at which to start search, defaulting to zero (beginning) position.

int find_last_of(string substring, [int position])
Returns the last position in the current string where substring is found. If not found, -1 is returned. Optional parameter position allows the user to specify an index into string at which to start searching backwards, defaulting to zero (beginning) position.

myStr = "Four score and seven years ago";
cout << myStr << endl;
cout << myStr.find_first_of("r") << endl;  // s/b position 3
cout << myStr.find_first_of("r", 6) << endl; // s/b position 8

// Starting at position 9 and going backward, this s/b = 8
cout << myStr.find_last_of("r", 9) << endl;

// Starting at position 6 and going backward, this s/b = 3
cout << myStr.find_last_of("r", 6) << endl;

Four score and six years ago
3
8
8
3

int first(string substring)
Returns the first position in the current string where substring is found. If not found, -1 is returned.

int find_first_not_of(string substring, [int position])
Returns the first position in the current string where the members of substring are not found. If none are located, -1 is returned. Optional parameter position allows the user to specify an index into string at which to start search, defaulting to zero (beginning) position.

string lower_case = "abcdefghijklmnopqrstuvwxyz ,-";
string str = "this is the lower-case part, AND THIS IS THE UPPER-CASE PART";

// this s/b position 29 where the 'A' in AND is located
// this s/b position 29 where the 'A' in AND is located
cout << "first non-lower-case letter in str at: " <<
str.find_first_not_of(lower_case) << endl;

first non-lower-case letter in str at: 29

remove(int position, int count)
Removes count number of characters from string, beginning at location, position.

replace(string oldString, string newString)
Replaces the oldString within str, with newString.

12.8.2 Tokenizer Class
This class provides an object that can be created by users to parse a string and return substrings based on user-supplied delimiters, or the default delimiters (" \n\t\0"). The following functions are in the Tokenizer Class.

Tokenizer name;
Create Tokenizer. // name will be the Tokenizer name used to call the other functions.

string getToken([string delim [, string quote]])
This function will sequentially parse sourceString based on delim delimiter values. If no parameter is passed into this function, the default delimiters will be used (" \n\t\0"). The optional quote argument is typically used when treating quoted strings as a single token. Within a quoted section delimiters are ignored.

string S = "Now is the time";  // Creates new string
Tokenizer next;              // Creates tokenizer next
next.sourceStr = S;          // Sets string to be parsed
cout << "Using the default delimiters...\n";
string newStr;                 // Variable to hold parsed substrings
// This loop prints out "Now", "is", "the", "time", each on a different line
while ((newStr = next.getToken()) != "") {
    cout << "\"" << newStr << "\"" << endl;
}
// Parse based on another delimiter
next.sourceStr = S;  // Reset the string to be parsed
// This loop prints out "Now is ", "he ", "ime", each on a different line
while ((newStr = next.getToken("t")) != "") {
    cout << "\"" << newStr << "\"" << endl;
}

Results:
Using the default delimiter....
"Now"
"is"
"the"
"time"
Using 't' as the delimiter...
"Now is 
"he 
"ime"

SourceStr = sourceString;
Setting this variable sets the string to be parsed with sourceString.
13 AutoDoc

AutoDoc is a script that runs NPSS with AutoDoc.in as the input file. AutoDoc generates HTML files for a class or list of classes; the information generated may include descriptions of variables and functions, and other objects depending on the class. When AutoDoc runs, an AutoDocViewer is created so you can run it interactively during an NPSS session. See section 14.4.7 for information on the AutoDocViewer.

13.1 Using AutoDoc

When you run AutoDoc, a browser pops up and displays the file. Note that in Netscape 7.0, if you use the no color option (-DNOCOLOR), the background remains black (instead of changing to white) but the colored text changes to black. This makes the file unreadable (except for title page and index). This works in Internet Explorer.

To run AutoDoc, type AutoDoc followed by all of the desired options as described below. (See the examples at the end of this section.)

**Output Options**

- DTITLEPAGE // a title page with version information will be placed at the top of the output
- DINDEX // an index will be generated with an entry for each component
- DVERBOSE // progress information will be written to stderr
- DPAGEBREAK // a page break will appear between each component
- DNOCOLOR // output will be black and white
- DNOINHERIT // a separate output file will be generated for each component
- DNOINHERIT // information from the ancestor classes will be included
- DSHOWPRIVATE // private objects will show up in output

-DOUTDIR=outputDirectory // output file(s) will be placed in outputDirectory
-DOUTFILE=filePath // provides name of HTML output file
    // If outputFile is "cout," HTML will be written to standard output.
    // If outputFile is blank, the browser will be run automatically.
    // (Note: the browser must be in your path.)
-DDBROWSER=FullScreen //To run a different browser, set browserCommand to the
           // path name of the new browser. The default browsers are
           // netscape (UNIX) and start (starts IE on Windows).
-DLOADFIRST // forces AutoDoc to use interpreted versions of components if they are available.
    // Output for interpreted component functions will include argument names in
    // addition to argument types. This data is not available for compiled components.

**Class Selection Options**

-DSTD // generates HTML for all components in the standard distribution
-DALL // generates HTML for all components returned from the listKnownTypes() function
-DINTERFACES=iface1,iface2,...,iface_n // generates HTML for all components from listKnownTypes()
    // that have any of the interfaces specified by iface1...iface_n.
-DexAB // adds extra airbreathing components to AirBreathing list
-DexRock // adds extra rockets components to Rockets list
-DCLASS=className // generate HTML for the class specified by className
-DLIST=listName // generates HTML for list specified by listName
    // available lists:
    // Infrastructure
    // Solver
    // Thermos
    // Ports
// DataViewers
// airE - airbreathing elements
// airSE - airbreathing subelements
// exAirE - extra AB elements
// exAirSE - extra AB subelements
// AirBreathing (Elements + Subelements)
// rockE - rocket elements
// rockSE - rocket subelements
// exRockE - extra rocket elements
// exRockSE - extra rocket subelements
// rockThermos - rocket thermos
// rockPorts - rocket ports
// Rockets - all rocket components
// Hypersonic
// ControlsTB
// externE - external wrapper elements
// externSE - external wrapper subelements
// extContain - external containers
// ExtWrappers - all external comps
// hifiContainers – hifi containers
// hifiPorts – hifi ports
// hifiCodes – hifi codes
// HiFi – all hifi components

Other Options
-DTHERMO=thermoName  // sets thermo package to that specified by thermoName. The default is "GasTbl."
-I <includeDir>  // Adds includeDir to the NPSS search path. This is sometimes necessary to locate the DLM or .int file for the class to be auto-documented.

Examples:
To generate an html file named AB.html that has a title page and an index and contains all AirBreathing components, including extra ones (those not included with the standard NPSS build), type the following:

AutoDoc -DOUTFILE=AB.html -DexAB -DTITLEPAGE -DINDEX -DLIST=AirBreathing

To get output for all components included in a standard build (no extra airbreathing and extra rocket components) and automatically display it in a browser, type the following:

AutoDoc -DSTD

If you want output for all components as above but you want a separate output file for each component, type:

AutoDoc -DSTD -DNOMERGE

To generate html for a single class (Duct) and display it in a browser, type:

AutoDoc -DCLASS=Duct

To generate html for all known classes and display it in a browser, type:

AutoDoc -DALL
To generate html for all known Elements, Subelements, and Ports and display it in a browser with an index, type:

```
AutoDoc -DINTERFACES=Element,Subelement,Port -DINDEX
```

### 13.2 Adding or Modifying AutoDoc Component Lists

If you want to add a component to an existing list, locate the appropriate string array inside of `AutoDoc.in` and add your new component(s) to it. If your components do not match any of the existing lists, you can create a new list and put your components there. You must also add the name of your new list to the `LISTNAMES` array. If a list’s name is not in `LISTNAMES`, the user will not be able to use that list. Also, if you want your new list to be part of the output from the `-DSTD` option, you will have to add your list as an arg to the `viewClassList()` call inside of the `#ifdef STD` block.
14  **DataViewer Reference Sheets**

Values obtained from the model may be looked at by using one of the following data viewers which display formatted and unformatted output. Each DataViewer Reference Sheet includes a list of variables–also known as *attributes*–associated with the Data Viewer, a description of the variables, and any default values, where relevant.

14.1  **Data Viewer Common Characteristics**

All four data viewers–Variable Dump Viewer (VarDump Viewer), PageViewer, CaseColumnViewer, and CaseRow Viewer–share several attributes which are described in the following table.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>aliases</td>
<td>Array of &quot;name = alias&quot; strings which are used as aliases by the data viewers. Setting this array adds, instead of replaces, the values. Setting an empty alias causes that name to be removed from the list.</td>
<td>{ }</td>
</tr>
<tr>
<td>coverPage</td>
<td>When this string is not empty, a cover page will be printed. It will also force the printing of a concluding page, which is a blank page containing a pageHeader and pageFooter.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>isActive</td>
<td>When this attribute is set to FALSE, data viewers do not respond to update() and display() calls.</td>
<td>TRUE</td>
</tr>
<tr>
<td>outStreamHandle</td>
<td>The name of the output stream. Can also be set equal to &quot;cout&quot; or &quot;cerr,&quot; which correspond to the STDOUT and STDERR streams on the display.</td>
<td>&quot;cout&quot;</td>
</tr>
<tr>
<td>pageBreakStr</td>
<td>The string gets printed between pages. Setting the value to &quot;FORMFEED&quot; will print a form feed character.</td>
<td>3 line feeds</td>
</tr>
<tr>
<td>pageFooter</td>
<td>If this string is not empty, a footer will be printed at the bottom of each page.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>pageHeader</td>
<td>If this string is not empty, a header will be printed at the top of each page.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>pageHeight</td>
<td>The height of the viewable area on a page, in characters. If set to zero (0), this attribute will be ignored.</td>
<td>58</td>
</tr>
<tr>
<td>pageWidth</td>
<td>The width of the viewable area on a page, in characters. If set to zero (0), this attribute will be ignored.</td>
<td>76</td>
</tr>
<tr>
<td>UnitSystem</td>
<td>The unit system used in displaying the viewers.</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

1  Although aliases can be added to a PageViewer, they are not used since that viewer does not print the names of its variables.

2  For CaseViewers, changing an alias does not have any effect until the variableList is reset.

3  The default values to be used are "US" and "SI" but may be different if the function `resetConvTable(…)` was used.
14.2 CaseColumnViewer/CaseRowViewer
CaseColumnViewer and CaseRowViewer share the same syntax, so their reference sheets are identical. The difference between the two is that the CaseColumnViewer prints columns of cases while the CaseRowViewer prints columns of variables.

Table 41. CaseViewer Attributes

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>caseHeader</td>
<td>A shorthand for entering the caseHeaderBody and caseHeaderVars. The first element of the array is the body, the rest are vars.</td>
<td>{ &quot;Case ???&quot;, &quot;CASE&quot; }</td>
</tr>
<tr>
<td>caseHeaderBody</td>
<td>A string that will be printed at the top of each column on each page. This string can be only one line long (i.e., no &quot;\n&quot; characters).</td>
<td>&quot;Case ???&quot;</td>
</tr>
<tr>
<td>caseHeaderVars</td>
<td>A list of variables that will be used to fill in caseHeaderBody.</td>
<td>{ &quot;CASE&quot; }</td>
</tr>
<tr>
<td>defIntFormat</td>
<td>This is the default format which will be used for integers.</td>
<td>&quot;???????????&quot;</td>
</tr>
<tr>
<td>defRealFormat</td>
<td>This is the default format which will be used for real numbers.</td>
<td>&quot;?????.??&quot;</td>
</tr>
<tr>
<td>defSNFormat</td>
<td>This is the default format which will be used for real numbers that do not fit into the default real format.</td>
<td>&quot;???E??&quot; (UNIX) &quot;??E??&quot; (Windows)</td>
</tr>
<tr>
<td>defStrFormat</td>
<td>This is the default format which will be used for strings.</td>
<td>&quot;???????????&quot;</td>
</tr>
<tr>
<td>doReset</td>
<td>Determines whether the case viewers are reset after a display. If set to FALSE, won't reset.</td>
<td>TRUE</td>
</tr>
<tr>
<td>showErrors</td>
<td>When set to TRUE, cases with errors will be printed. If FALSE, bad cases will not be displayed().</td>
<td>TRUE</td>
</tr>
<tr>
<td>showHeaders</td>
<td>Allows you to omit header information. When set to TRUE, headers will be printed as usual. If set to FALSE, page break strings, page headers/footers, and case headers are omitted to create an output more suitable for plotter use. When set to -1, showHeaders represents ONCE, and will display the title and headers on the first page only. This variable will then revert to FALSE for subsequent pages.</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
### 14.3 PageViewer

PageViewers are made up of multiple text blocks. The starting point is the anchor, which is described below.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor</td>
<td>The name of the text block that appears in the upper left corner of the page. From this starting point, all other text blocks are arranged.</td>
<td>The name of the first text block declared</td>
</tr>
</tbody>
</table>

### 14.4 Text Blocks

All text blocks have the following two attributes in common.

#### Table 42. Text Block Attributes

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>The name of the text block which will be placed to the right of this text block.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>bottom</td>
<td>The name of the text block which will be placed below this text block.</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>
14.4.1 SimpleBlock

Table 43. SimpleBlock Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>body</td>
<td>The string–with embedded format strings–which will be used as the body for the text block.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>content</td>
<td>A shorthand combination of the body and vars variables. The first element of the array is the body, the rest are vars. Can be used with the other two.</td>
<td>{ &quot;&quot; }</td>
</tr>
<tr>
<td>vars</td>
<td>List of the variable names to be used with the body string.</td>
<td>{ }</td>
</tr>
</tbody>
</table>

14.4.2 DColTBlock, LinkColTBlock and DRowTBlock

The only difference between the row and column versions of the dynamic text blocks is the direction the attributes are prdev/Executive/src/Modelinted in. The DColTBlock prints attributes in columns, while DRowTBlock prints them in rows. A specialized text block called LinkColTBlock also exists. It is used to display port data, which is treated differently from other components.

Table 44. Dynamic Text Block Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>A shorthand for entering the titleBody and titleVars. The first element of the array is the body, the rest are vars. Can be used with the other two.</td>
<td>{ &quot;&quot; }</td>
</tr>
<tr>
<td>titleBody</td>
<td>The string–with embedded format strings–which will be used as the title for the dynamic text block.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>titleVars</td>
<td>List of the variable names to be used with the titleBody string.</td>
<td>{ }</td>
</tr>
<tr>
<td>compAttr</td>
<td>These are the attributes that will be looked at on the components of type compType that the model returned. The attribute name can also be followed by a colon (:) and a format string which will override the default (see below). It can also be followed by an equals sign (=) and an alias. (e.g., { &quot;dt:???.??=Delta Temp&quot; } will use &quot;???.??&quot; As the format and &quot;Delta Temp&quot; as the alternate label.)</td>
<td>{ }</td>
</tr>
<tr>
<td>compType</td>
<td>This is the type name of the component which the model will be queried for.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Variable</td>
<td>Variable Description</td>
<td>Default Value</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>compTypeFormat</td>
<td>This is the string format used when printing component names. If set to an empty string (default), the full name will be printed.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>defIntFormat</td>
<td>This is the default format which will be used for integers.</td>
<td>&quot;??????????&quot;</td>
</tr>
<tr>
<td>defRealFormat</td>
<td>This is the default format which will be used for real numbers.</td>
<td>&quot;?????.??&quot;</td>
</tr>
<tr>
<td>defSNFormat</td>
<td>This is the default format which will be used for real numbers that do not fit into the default real format.</td>
<td>&quot;???.??E??&quot; (UNIX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;???.??E???&quot; (Windows)</td>
</tr>
<tr>
<td>defStrFormat</td>
<td>This is the default format which will be used for strings.</td>
<td>&quot;??????????&quot;</td>
</tr>
<tr>
<td>componentList</td>
<td>This string array can contain a list of components to be printed (instead of always doing an automatic search based on compType). A blank &quot;&quot; value in this array results in a blank line being printed between the specified components. Labels can also be inserted between components by specifying a colon character &quot;:&quot; immediately before the label. An example would be &quot;:Compressor #1&quot;. The components displayed will retain the order in which they are listed. When this array is empty, all components matching compType will be listed. When this array is used, excludeCompTypes and useSolvSeqOrder are ignored in favor of the user component selection/order.</td>
<td>{ }</td>
</tr>
<tr>
<td>showUnits</td>
<td>If set to TRUE, displayed variables will also show unit information (example: rpm, lbm/sec, etc.)</td>
<td>FALSE</td>
</tr>
<tr>
<td>excludeCompTypes</td>
<td>This string array can contain component type names which will be removed from the list of components matching the type compType. If componentList has entries, this array is ignored in favor of the user preference of components.</td>
<td>{ }</td>
</tr>
<tr>
<td>showColHeader</td>
<td>(Only in DColTBlock and LinkColTBlock.) If set to FALSE, a DColTBlock and LinkColTBlock will not print the attribute names on the line below the title, allowing two blocks to be put very close together.</td>
<td>TRUE</td>
</tr>
<tr>
<td>useSolvSeqOrder</td>
<td>When set to TRUE, components are printed in the order they execute in. If a component is not in a solver sequence, then it will not be printed. If none of the componentList values is set to TRUE, all components matching compType will be listed.</td>
<td>TRUE</td>
</tr>
<tr>
<td>Variable</td>
<td>Variable Description</td>
<td>Default Value</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td>components are in a solver sequence, nothing will be printed. When set to FALSE, all components are printed.</td>
<td></td>
</tr>
</tbody>
</table>
14.4.3 GroupBlock

GroupBlocks, like PageViewers, are composed of text blocks, including other GroupBlocks.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor</td>
<td>The name of the text block that appears in the upper left corner of the page. From this starting point, all other text blocks are arranged.</td>
<td>The name of the first text block declared</td>
</tr>
<tr>
<td>addGutter</td>
<td>When set to TRUE, adds whitespace around text blocks.</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

14.4.4 EmptyTextBlock

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>The height of the block</td>
<td>0</td>
</tr>
<tr>
<td>width</td>
<td>The width of the block.</td>
<td>0</td>
</tr>
</tbody>
</table>

14.4.5 UserTextBlock

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>filledBlock</td>
<td>This string array must be filled by the overloaded update() function. It contains the text which will be given to the PageViewer to be printed.</td>
<td>{ }</td>
</tr>
</tbody>
</table>

14.4.6 VarDumpViewer

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>variableList</td>
<td>The list of variables whose values will be displayed. They can include wildcard characters: *'s and ?'s.</td>
<td>{ &quot; &quot; }</td>
</tr>
</tbody>
</table>
14.4.7 AutoDocViewer

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Default</th>
<th>Units</th>
<th>IO Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>browserCmd</td>
<td>command used to run the browser</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>doIndex</td>
<td>if TRUE, display an index page at the top of html output</td>
<td>0</td>
<td>unset</td>
<td></td>
</tr>
<tr>
<td>doPageBreak</td>
<td>if TRUE, insert page breaks between each component page</td>
<td>0</td>
<td>unset</td>
<td></td>
</tr>
<tr>
<td>doTitlePage</td>
<td>if TRUE, display a title page at the top of html output</td>
<td>0</td>
<td>unset</td>
<td></td>
</tr>
<tr>
<td>merge</td>
<td>if TRUE, writes all html output to a single file. If FALSE, each component will be written to &lt;compname&gt;.html and the index will be found in Index.html</td>
<td>1</td>
<td>unset</td>
<td></td>
</tr>
<tr>
<td>noColor</td>
<td>if TRUE, generate html for black and white output</td>
<td>0</td>
<td>unset</td>
<td></td>
</tr>
<tr>
<td>outDir</td>
<td>output directory for generated html files</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outFile</td>
<td>name of the generated html file, if merge==TRUE. If outFile==&quot;cout&quot;, html will be written to standard output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>showInheritance</td>
<td>if TRUE, displays separate pages for all ancestors of a given class</td>
<td>1</td>
<td>unset</td>
<td></td>
</tr>
<tr>
<td>showPrivate</td>
<td>if TRUE, displays private variables in html output</td>
<td>0</td>
<td>unset</td>
<td></td>
</tr>
<tr>
<td>verbose</td>
<td>if TRUE, writes progress information to standard err</td>
<td>0</td>
<td>unset</td>
<td></td>
</tr>
</tbody>
</table>

Functions

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>blockDGen (string) Returns a string containing an ASCII diagram of the given Element or Subelement</td>
</tr>
<tr>
<td>void</td>
<td>createIndexCategory (string, string[]) Associates a category in the index with a list of class names</td>
</tr>
<tr>
<td>string</td>
<td>viewClass (string) Generates an html page for the given class</td>
</tr>
<tr>
<td>string</td>
<td>viewClassList (string[]) Generates an html page for the given list of classes. Takes any number of args that can be combinations of strings and string[]s</td>
</tr>
<tr>
<td>string</td>
<td>viewInstance (string) Generates an html page for the given instance</td>
</tr>
</tbody>
</table>

Usage Notes

This class generates html pages for individual classes or lists of classes. These pages contain information about the class which includes a list of its variables and their descriptions, a list of its functions, and a list of other objects that reside in the class. The html output can be sent to a file, to standard output, or displayed immediately in a web browser.

AutoDocViewer has a baseType of DataViewer and DataViewer has a baseType of VariableContainer. See the DataViewer Reference Sheets in the NPSS Reference Sheets document for the functions of these ancestor classes.
15 Stream Reference Sheets

NPSS streams are used to direct input data from, and output data to, an outside source.

15.1 Common Attributes
The attributes listed below are common to both input and output streams.

Table 49. Common Attributes of Streams

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>If TRUE, line termination translations are disabled, and items are read/written in a binary form rather than text. Should be set before the file is opened.</td>
<td>FALSE</td>
</tr>
<tr>
<td>byteSwap</td>
<td>If TRUE, binary data is byte-swapped during read and write operations.</td>
<td>FALSE</td>
</tr>
<tr>
<td>filename</td>
<td>The name of the file to be opened for read/writing.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>good</td>
<td>Used to determine if a stream is still valid. The default value is only TRUE if the stream has been properly initialized.</td>
<td>TRUE</td>
</tr>
<tr>
<td>eof</td>
<td>Checks whether the stream is at the end of a file.</td>
<td>FALSE</td>
</tr>
<tr>
<td>singlePrecision</td>
<td>If TRUE, and binary or unformatted is TRUE, real values are read/written as single precision floating point format.</td>
<td>FALSE</td>
</tr>
<tr>
<td>unformatted</td>
<td>If TRUE, leading and trailing record length indicators are processed. Implies binary.</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
15.2 Output Streams
Output streams allow users to send data to output devices, i.e., a monitor, a file on disk, or a printer.

Table 50. Output Stream Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>If set to TRUE, it will append a file instead of truncating it. Must be set before the file is opened.</td>
<td>FALSE</td>
</tr>
<tr>
<td>autoFlush</td>
<td>If TRUE, output is flushed after every ‘bare’ newline (\n” or endl). This can be slow on some platforms.</td>
<td>TRUE</td>
</tr>
<tr>
<td>width</td>
<td>An integer value which determines the size of the field your data will be printed to.</td>
<td>0</td>
</tr>
<tr>
<td>precision</td>
<td>The precision that real numbers will be printed to, i.e., the number of places (digits) to the right of the decimal point.</td>
<td>6</td>
</tr>
<tr>
<td>showpoint</td>
<td>Used with the variable &quot;precision,&quot; it will pad significant digits with zeros, e.g., 4.100 instead of 4.1.</td>
<td>FALSE</td>
</tr>
<tr>
<td>scientific</td>
<td>When set to TRUE, real values will print in scientific notation.</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

15.3 Input Streams

Table 51. InputStreams Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>skipWhite</td>
<td>If TRUE, whitespace is skipped on string input. If FALSE, string's input will contain the whitespace characters as read.</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
16 The NPSS Solver Reference

This section gives a detailed inventory of the input/command syntax and attributes for the Solver. Where two slash marks, i.e., //, appear followed by text, the text is explanatory information or comments and not information the user must enter.

16.1 Solver (NPSSSteadyStateSolver class)

16.1.1 Solver Definition Syntax Example

The following example shows how to create a solver within an assembly:

```cpp
// Start definition of Engine Assembly
Element Assembly Engine {
    // start definition of solver
    Solver solver {
        defaultTolerance = 0.01;
    }
    // define other parts of the assembly
    // block definition of extra solver stuff
    solver {
        defaultPerturbation = 0.1;
        defaultPerturbationType = FRACTIONAL;
        defaultDxLimit = 0.01;
        defaultDxLimitType = FRACTIONAL;
    }
}    //end of Engine
// single solver attribute override outside the engine
Engine.solver.defaultToleranceType = FRACTIONAL;
```

16.1.2 Solver Input/Commands

The following commands are member functions of Solver, so in order to execute them, you must either add a dot (.) and append them to the name of a solver object, e.g., `solver.addDependent("myDep")`, or you must call them within the scope of a solver, for example:

```cpp
solver {
    addDependent("myDep");
    addIndependent("myIndep");
}
```

**Solver Member Functions**

- **void addDependent(dependent_name)**
  Adds the named Dependent or Integrator (which must have been previously defined) to the solver.

- **void addDSV(DSV_name)**
  Adds the named DiscreteStateVariable (DSV) (which must have been previously defined) to the solver.

- **void addIndependent(independent_name)**
  Adds the named Independent object (which must have been previously defined) to the solver.

- **void clear ()**
  Removes all independents and dependents from the solver. The removed objects are still defined, however.
void removeDependent (dependent_name)
Removes the named Dependent or Integrator (which must have been previously defined) from the solver.

void removeDSV (DSV_name)
Removes the named DiscreteStateVariable (DSV) object (which must have been previously defined) from the solver.

void removeIndependent (independent_name)
Removes the named independent object (which must have been previously defined) from the solver.

void resetConstraints()
Updates the solver active and auxiliary Dependent lists, based on current constrained Dependent definitions. It is no longer necessary to call this function whenever a constraint Dependent(s) is added to or removed from a target Dependent. The target Dependent will automatically notify any solver to which it has been added of the configuration change.

void revertToDefaults()
Unlocks all "lockable" attributes in independents, dependents, etc., so that the solver defaults can be used (i.e., it resets attributes you set for specific independents, dependents, etc).

void run()
Orders the solver to solve the Model for a single point with the current input conditions.

int varNameIsActiveIndep (variable_path_name)
Determines if the given full variable path name corresponds to the model variable controlled by any of the independent objects contained by the solver. It returns TRUE/FALSE.

16.1.3 The preconvergence Function
This is a user implementable function, much like the preexecute() and postexecute() functions in elements and subelements. It takes no arguments and returns void. The function is implemented in the user model input within the scope of the solver (e.g., solver { void preconvergence() {...}}). The function body can contain any code that the user wishes to have executed.

This function is called immediately before converging the continuous model. If there are no DSVs in the Solver, then this function is called only once per point. If, however, there are DSVs in the solver, then this function is called once each discontinuous iteration. When there are DSVs, it is called at a point after either the DSVs have been initialized (first discontinuous iteration) or after the DSV state values have been updated (subsequent discontinuous iterations) and immediately before the continuous model is (re)converged.

The preconvergence function can be used to perform setup/initialization/guess tasks prior to converging the model. The following is an example implementation that uses this function to guess the position of a bleed valve DSV on the first discontinuous iteration.

```c
void preconvergence() {
    if(disconIterationCounter == 1) {
        if(PLA <= 35) {
            CmpH.BleedValvePosition.stateInitial = "CDP";
        } else {
            CmpH.BleedValvePosition.stateInitial = "INTERSTAGE";
        }
    }
}
```
### 16.1.4 Solver User Input Attributes

#### Table 52. Solver User Input Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>broydenLimit</td>
<td>Compared to convergenceRatio to determine if a Broyden update should be performed on the Jacobian matrix. If the calculated value of convergenceRatio is smaller than the value of this variable, the speed of convergence is judged to be sufficiently rapid enough to forgo a Broyden update.</td>
<td>0.0</td>
</tr>
<tr>
<td>broydenCutoffMultiplier</td>
<td>Scale factor applied to iTolerance. When the absolute value of errorCon ≤ broydenCutoffMultiplier * iTolerance of all active Dependents, further broyden updates are not performed. The purpose is to keep from adjusting the matrix when the model is very close to the answer.</td>
<td>1.0</td>
</tr>
<tr>
<td>convergenceLimit</td>
<td>Compared to convergenceRatio to determine if an iteration is approaching convergence rapidly enough to be considered successful. If the calculated value of convergenceRatio is larger than this variable, the failedConvergenceCounter is incremented. Nominal range is between zero and one.</td>
<td>1.0</td>
</tr>
<tr>
<td>constraintNames</td>
<td>Returns or accepts a 1-D string array containing the full path names of constraint Dependents. Constraints are not added through this attribute. Instead they are added when the target Dependent with which they are associated is added to the Solver. When this attribute is on the right side of an assignment operator, it returns a 1-D string array of all the constraint Dependents currently in the Solver (in the order they appear in the Solver internal list). When it is on the left side of an assignment operator, this variable accepts a 1-D string array of constraint Dependent path names. The only effect of assignment is to check the number of names in the array being assigned against the number of constraint Dependents currently in the Solver. If they are not equal, an error message is generated.</td>
<td>{}</td>
</tr>
<tr>
<td>constraintHandlingOpt</td>
<td>This option turns all constraint handling on (TRUE) or off (FALSE).</td>
<td>TRUE</td>
</tr>
<tr>
<td>debugLevel</td>
<td>This defines the amount of diagnostic output generated. (See Section 16.7 for details).</td>
<td>&quot;NONE&quot;</td>
</tr>
<tr>
<td>defaultDxLimit</td>
<td>The amount that each independent variable may be changed during each iteration toward a solution. This value may be overridden individually in each independent object.</td>
<td>0.10</td>
</tr>
<tr>
<td>defaultDxLimitType</td>
<td>Defines whether the dxLimit value entered is taken as an absolute or fractional amount (compared to current independent value). Values are &quot;FRACTIONAL&quot; and &quot;ABSOLUTE.&quot;</td>
<td>&quot;FRACTIONAL&quot;</td>
</tr>
<tr>
<td>defaultPerturbation</td>
<td>The amount by which the solver independents will be varied in order to generate the Jacobian matrix. This value may be overridden in each independent object individually.</td>
<td>0.001</td>
</tr>
<tr>
<td>defaultPerturbationType</td>
<td>Defines whether the perturbation value entered is to be</td>
<td>&quot;FRACTIONAL&quot;</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>defaultTolerance</td>
<td>The largest convergence error (which may be different from matrix error) which qualifies each dependent as converged.</td>
<td>0.0001</td>
</tr>
<tr>
<td>defaultToleranceType</td>
<td>Defines whether the convergence tolerance is compared with the absolute or fractional error (relative to the size of the second dependent model variable) when testing convergence. Values are &quot;FRACTIONAL,&quot; and &quot;ABSOLUTE.&quot;</td>
<td>&quot;FRACTIONAL&quot;</td>
</tr>
<tr>
<td>defaultXLimitReport</td>
<td>Provides a default xLimitReport value for Independents to determine whether or not to report xMinLimit/xMaxLimit violations or predictor cut-backs to the limit.</td>
<td>FALSE</td>
</tr>
<tr>
<td>dependentNames</td>
<td>Returns or accepts a 1-D string array containing the full path names of unconstrained and target Dependents. This list does not contain the path names of any constraint Dependents. When on the right side of an assignment operator, returns a 1-D string array of all the unconstrained and target Dependents currently in the Solver, in the order they appear in the Solver internal list. When on the left side of an assignment operator, accepts a 1-D string array of Dependent path names. When an array of names is assigned, all the Dependents and Constraints currently in the solver are removed and the names in the assigned array are added. If any of the Dependents in the array have constraint Dependents associated with them, those Constraints are added along with their target Dependent. Constraints are added in the order they were added to the target Dependent and the order the target Dependents appear in this array. Assigning an array of names to this attribute invalidates any existing Jacobian matrix; thus, the forceNewJacobian flag is set to TRUE.</td>
<td>{}</td>
</tr>
<tr>
<td>divergenceCutoffMultiplier</td>
<td>When the square-root of the sum of the square (SQSS) of the errors (recorded in the attribute errorMagNorm) is within a divergenceCutoffMultiplier multiple of the SQSS of the fractional tolerances of all active Dependents, the divergenceLimit is ignored. The purpose is to keep the divergenceLimit from causing a new matrix to be generated when the model is very close to the answer.</td>
<td>1.0</td>
</tr>
<tr>
<td>divergenceLimit</td>
<td>Compared to convergenceRatio to determine if an iteration is causing the system to diverge above an acceptable rate. If convergenceRatio is larger a new Jacobian matrix is immediately generated. Nominal range is greater than one.</td>
<td>2.0</td>
</tr>
<tr>
<td>DSVnames</td>
<td>A string array attribute that returns or accepts a 1-D string array containing the path names of DSVs. When on the right side of an assignment operator, returns a 1-D string array of all the DSVs currently in the Solver in the order they appear in the Solver internal list. When on the left side of an assignment operator, accepts a 1-D string array of DSV path names. On assignment, any DSVs currently in the solver are removed and the names in the assigned array are added.</td>
<td>{}</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>DSVvalues</td>
<td>A string array attribute that returns or accepts a 1-D string array containing the state values of the DSVs. When on the right side of an assignment operator, returns a 1-D string array containing the state values of the DSVs currently in the Solver in the order the DSVs appear in the Solver internal list. When on the left side of an assignment operator, accepts a 1-D string array of DSV state values. The first DSV value is assigned to the first DSV in the solver list of DSVs. The DSV will generate an error if the state value is not contained in the DSV list of valid state values. The fact that the DSV value does appear on the list of valid DSV state values does not guarantee that the order of values in this array corresponds to the order of the DSVs since the same value may be valid for two or more DSVs. If there is any doubt about whether the order of the DSVs might have changed between the point where the DSVvalues array was saved and when it is restored, it is advised that the DSVnames and DSVvalues arrays be saved and restored as a set.</td>
<td>{}</td>
</tr>
<tr>
<td>executionSequence</td>
<td>List of objects executed in order during each solver pass. See Section 0 for more information. Aliased to solverSequence in the Assembly to support backward compatibility.</td>
<td>{}</td>
</tr>
<tr>
<td>firstNewJacobian</td>
<td>Specifies whether to use the final Jacobian matrix from the previous point as the initial matrix for the current point or to immediately generate a new matrix before beginning convergence iterations. Values are &quot;CARRY_OVER&quot; and &quot;CALCULATE.&quot;</td>
<td>&quot;CARRY_OVER&quot;</td>
</tr>
<tr>
<td>forceNewJacobian</td>
<td>Setting to TRUE causes new Jacobian to be immediately generated. After generating a new matrix, this option is reset to FALSE. Based on user options (like regenNewJacobian) and on other factors in the convergence process, the Solver uses this option to specify when a new Jacobian is required. The default value at the time of solver construction is TRUE, indicating that a new Jacobian is required to use the solver.</td>
<td>TRUE</td>
</tr>
<tr>
<td>independentNames</td>
<td>A string array attribute that returns or accepts a 1-D string array containing the path names of Independents. When on the right side of an assignment operator, returns a 1-D string array of all the Independents currently in the Solver in the order they appear in the Solver internal list. When on the left side of an assignment operator, accepts a 1-D string array of Independent path names. On assignment, any Independents currently in the solver are removed and the names in the assigned array are added. Assigning an array of names to this attribute invalidates any existing Jacobian matrix, thus the forceNewJacobian flag is set to TRUE.</td>
<td>{}</td>
</tr>
<tr>
<td>independentValues</td>
<td>A real array attribute that returns or accepts a 1-D real array containing the values in the independents. When on the right side of an assignment operator, returns a 1-D real array containing the values of the Independents currently in the Solver in the order the Independents appear in the Solver internal list. When on the left side of an assignment</td>
<td>{}</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>operator</td>
<td>This variable accepts a 1-D real array of values. The first value is passed to the ( x ) value of the first Independent. If mapping functions are present in an Independent, the value of the model parameter referenced by the Independent (and available through the ( xModel ) attribute of the Independent) will be different than the values contained in this array. The independentNames array does not have to be set in order for this array of values to be set, if the correct set of Independents is already present and in the correct order. An error will be issued if the number of Independents doesn’t match the number of values in this array. However, there is no way to tell if the Independents are in the same order in the solver as they were when this array of values was saved. If there is any doubt that the order of the Independents might have changed between the point where the values were saved and where they were reset into the Solver, it is best to save and restore the names and values together as a set.</td>
<td></td>
</tr>
<tr>
<td>jacobian</td>
<td>A real array attribute that returns or accepts a 2-D real array containing the full rectangular Jacobian matrix. The first ( n ) rows, where ( n ) is the number of columns, will contain the rows for unconstrained and target Dependents. The order of the rows is the same as the order of the Dependents list in the Solver. Subsequent rows will contain coefficients for any constraints present in the Solver. The order of these additional rows will be the order of the constraint Dependent list in the Solver. This will be the order of the rows even if the jacobian is obtained after convergence of a point with constraints active. When on the right side of an assignment operator, a 2-D real array is returned. When on the left side of an assignment operator, this variable accepts a 2-D real array that replaces any Jacobian matrix currently in the Solver. It is assumed that the order of the rows in any matrix corresponds to the order of the Dependents and Constraints, and the order of the columns corresponds to the order of the Independents. Restoring a Jacobian matrix through this attribute will cause the forceNewJacobian flag to be set to FALSE.</td>
<td>{ }</td>
</tr>
<tr>
<td>linearDependencyTolerance</td>
<td>If the term-by-term ratios of any two rows or columns in the Jacobian matrix are all the same, plus or minus the value of this attribute, then the matrix is singular due to linear dependency. This test is performed only if the determinant is less than singularityTest.</td>
<td>1e-09</td>
</tr>
<tr>
<td>maxBroydens</td>
<td>The maximum number of Broyden updates performed on a given Jacobian matrix before a new matrix is generated. Tested against the broydenCounter variable</td>
<td>25</td>
</tr>
<tr>
<td>maxConstraintProjections</td>
<td>The maximum number of error projection iterations (within a single solver convergence iteration) performed to select constraints. Setting this attribute to zero will disable the use of projected error for selecting constraints (current error will be used instead).</td>
<td>3</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>maxConvergeFailures</td>
<td>The number of consecutive times that the convergenceRatio may exceed the convergenceLimit before an new Jacobian matrix is generated. This counter is reset to zero if on any iteration the convergenceRatio is less than the convergenceLimit.</td>
<td>3</td>
</tr>
<tr>
<td>maxDisconIterations</td>
<td>Maximum allowed number of discontinuous convergence attempts. The continuous portion of the model must also converge for each discontinuous convergence attempt.</td>
<td>10</td>
</tr>
<tr>
<td>maxIterations</td>
<td>The maximum number of solution iterations allowed for any single point solution (case or time-step) of the continuous model. Does not include model passes required to generate the Jacobian.</td>
<td>25</td>
</tr>
<tr>
<td>maxFreeToggles</td>
<td>Default value for DSVs active in the Solver. Controls the number of times that DSVs may change state while converging the continuous portion of the model.</td>
<td>0</td>
</tr>
<tr>
<td>maxJacobians</td>
<td>The maximum number of Jacobians that may be generated from scratch for any single case.</td>
<td>10</td>
</tr>
<tr>
<td>maxPasses</td>
<td>The maximum allowed number of model passes, including those for Jacobian matrix generation. A value of zero signifies that any number of passes is allowed, since this is usually not a criterion dictated by the user.</td>
<td>0</td>
</tr>
<tr>
<td>minJacobianTerm</td>
<td>Defines the smallest non-zero term allowed when generating a new Jacobian matrix. If the calculated term is smaller than this attribute, that term is set to zero. The default value is zero, so unless changed, this attribute has no effect.</td>
<td>0.0</td>
</tr>
<tr>
<td>minBroydenUpdate</td>
<td>Defines the minimum fraction that each term in the Broyden update must change the corresponding term in the Jacobian before it is applied. If the term in the Jacobian is exactly zero, then the Broyden update term is applied if the update is greater than the (minBroydenUpdate * 1e-03). This makes sure that the update is significant. The default is zero, so this attribute has no effect unless a non-zero value is specifically set.</td>
<td>0.0</td>
</tr>
<tr>
<td>regenNewJacobian</td>
<td>This option controls when the Jacobian matrix is regenerated from scratch rather than simply being updated. Options are: &quot;WHEN_NEEDED,&quot; &quot;WHEN_FORCED,&quot; and &quot;EVERY_ITERATION.&quot;</td>
<td>&quot;WHEN_NEEDED&quot;</td>
</tr>
<tr>
<td>preExecutionSequence</td>
<td>List of objects executed in order before the solver assumes control of execution. Also aliased in the Assembly as presolverSequence to support backward compatibility.</td>
<td>{}</td>
</tr>
<tr>
<td>postExecutionSequence</td>
<td>List of objects executed in order after the solver exits. Object names are commonly placed here by the user to cause them to be executed once after the solver converges. Also aliased in the Assembly as presolverSequence to support backward compatibility.</td>
<td>{}</td>
</tr>
<tr>
<td>resolveMinMaxConflict</td>
<td>Option which determines what constraint will be invoked if a min and a max constraint are both violated at the same priority level. Options are &quot;MIN&quot; and &quot;MAX.&quot;</td>
<td>&quot;MAX&quot;</td>
</tr>
<tr>
<td>singularityTest</td>
<td>If the Jacobian matrix determinant is less than the value of this attribute, then a detailed singularity check is performed to identify either the zero rows and/or columns or the linearly dependent rows and/or columns.</td>
<td>1e-06</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Mathematically a matrix is singular only if the determinant is exactly zero. However round-off error in calculations can result in a non-zero determinant for a matrix that is indeed singular. This attribute sets the threshold.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>singularRowColTolerance</td>
<td>If the largest absolute value in any row or column is less than this attribute, the matrix is judged to be effectively singular. If this attribute is set to zero, only matrices that are mathematically singular are flagged as such. This test is performed only if the determinant is less than singularityTest.</td>
<td>1e-10</td>
</tr>
<tr>
<td>solutionMode</td>
<td>Options are &quot;STEADY_STATE,&quot; &quot;ONE_PASS,&quot; and &quot;TRANSIENT.&quot;</td>
<td>&quot;STEADY_STATE&quot;</td>
</tr>
<tr>
<td>testXConvergence</td>
<td>Determines whether the iteration-to-iteration change in the independent variables is also tested (relative to tolerance value) when determining convergence of the model solver.</td>
<td>FALSE</td>
</tr>
<tr>
<td>toleranceScaleFactor</td>
<td>A multiplier on the tolerance of all Dependents regardless of whether they are using the Solver defaultTolerance value of a locally specified value. Setting this value does not effect either the default tolerance or any local values in individual Dependents. This variable is only functional during steady-state execution. A variable of the same name in the TransientExecutive object transient controls tolerance scaling during convergence of time steps during a transient run.</td>
<td>1.0</td>
</tr>
<tr>
<td>switchYRefLock</td>
<td>Allows for different yRefLock behaviors: (&quot;ALWAYS&quot;, &quot;ONCE&quot;, &quot;JACOBIAN&quot;)</td>
<td>ONCE</td>
</tr>
</tbody>
</table>

### 16.1.5 Solver Output (Calculated) Attributes

#### Table 53. Solver Output Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aDSVchangedState</td>
<td>An integer flag which is TRUE if the changedState flag of any DSV is TRUE at the end of convergence. The DSV.changedState flag is true if the converged vale of state is not the same as stateInitial</td>
</tr>
<tr>
<td>aDSVtoggled</td>
<td>An integer flag which is TRUE if the toggled flag of any DSV is following a discrete iteration during convergence. By definition this flag will always be FALSE when the case is converged since each DSV is converged only when they have not toggled.</td>
</tr>
<tr>
<td>broydenCounter</td>
<td>The number of Broydens performed on the current Jacobian matrix. Reset to zero when a new Jacobian is generated. Tested against maxBroydens to determine if a new Jacobian should be generated.</td>
</tr>
<tr>
<td>constraintsActive</td>
<td>Signals whether any constraints were involved in the final solution at a given point. Values are TRUE and FALSE.</td>
</tr>
<tr>
<td>continuousConverged</td>
<td>Convergence indicator for the continuous portion of the model.</td>
</tr>
<tr>
<td>converged</td>
<td>Convergence indicator for the model and solver (including discontinuous model states).</td>
</tr>
<tr>
<td>convergenceRatio</td>
<td>Ratio of the square root of the sum of the squares of the convergence errors of the current iteration to that of the previous iteration. The convergence errors are</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>currentDSVlevel</td>
<td>The DSV level value being iterated</td>
</tr>
<tr>
<td>determinant</td>
<td>Determinant of the Jacobian matrix.</td>
</tr>
<tr>
<td>disconIterationCounter</td>
<td>Number of discontinuous convergence attempts.</td>
</tr>
<tr>
<td>DSVbounce</td>
<td>Integer flag. TRUE if the bounce flag of any DSV present in the Solver was set to TRUE when the Solver finished overall convergence.</td>
</tr>
<tr>
<td>dxLimited</td>
<td>Array of delta change actually applied to the variables pointed to by the Independent objects. Resulting from dxUnlim * scaleFactor. During Jacobian generation contains all zeros except the perturbation delta for the independent being perturbed. Array zeroed out after point converged.</td>
</tr>
<tr>
<td>dxLimIndepIndex</td>
<td>The index number, starting at 1, of the Independent that resulted in the scaleFactor used to limit the dxUnlim vector. If no scaling is required (scaleFactor = 1), this attribute is set to zero.</td>
</tr>
<tr>
<td>dxLimitIndepName</td>
<td>The full path name of the limiting Independent. If no scaling is required (scaleFactor = 1), this attribute is an empty string.</td>
</tr>
<tr>
<td>dxUnlim</td>
<td>Array of delta changes to the variables pointed to by the Independent objects. The values are those resulting from multiplying the inverted Jacobian by the errorsActive vector. The non-linear nature of the actual model means that this vector needs to be limited in the amount the independents can move in a given iteration. Thus this vector is multiplied by the calculated value of scaleFactor to yield the dxLimited vector. During Jacobian generation contains all zeros except the perturbation delta for the independent being perturbed. Array zeroed out after point converged.</td>
</tr>
<tr>
<td>errorConRMS</td>
<td>The root mean square value of the convergence errors of all active Dependents.</td>
</tr>
<tr>
<td>errorConverged</td>
<td>Array of individual Jacobian error convergence status(es) flags</td>
</tr>
<tr>
<td>errorsActive</td>
<td>Array of errorIter values for the active Dependents. Used to generate updates to the Independent variables.</td>
</tr>
<tr>
<td>errorsInactive</td>
<td>Array of the errorIter values of the inactive Dependents</td>
</tr>
<tr>
<td>errorToleranceRatio</td>
<td>The ratio of errorConRMS to toleranceRMS. This represents a general measure of how “close” to convergence the solution is. This is presented for user information only. It is not used to determine convergence or for any other determination. However, when this value is much greater than 1, it is clear that the model is far from the solution point and as it approaches 1 the solution as a whole is approaching convergence.</td>
</tr>
<tr>
<td>hasBeenRun</td>
<td>Integer flag set to TRUE if at least one point ran to convergence.</td>
</tr>
<tr>
<td>iterationCounter</td>
<td>Number of continuous-model convergence attempts.</td>
</tr>
<tr>
<td>J</td>
<td>n x n 2-D array containing the active Jacobian, where n is the number of Independents.</td>
</tr>
<tr>
<td>Jaux</td>
<td>m x n 2-D array containing the rows for the inactive Dependents, where m is the number of inactive Dependents and n is the number of Independents.</td>
</tr>
<tr>
<td>lastPerturbationPass</td>
<td>Set to TRUE when the Solver is running the last perturbation pass when generating a new Jacobian matrix. This is useful information because the passType is still perturbationPass when the newly generated Jacobian is used to calculate the dxLimited vector to be set at the start of the next iteration and when anything placed by the user in the userReport function is evaluated.</td>
</tr>
<tr>
<td>matrixSize</td>
<td>Number of rows and columns in the square Jacobian matrix. Is equal to the number of Independents and active Dependents.</td>
</tr>
<tr>
<td>maxDSVlevel</td>
<td>The maximum DSVlevel value of all DSVs currently present in the Solver</td>
</tr>
<tr>
<td>minDSVlevel</td>
<td>The minimum DSVlevel value of all DSVs currently present in the Solver</td>
</tr>
<tr>
<td>minMaxConflict</td>
<td>Set to TRUE if a conflict between a min and a max type constraint occurred. The</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>resolveMinMaxConflict</td>
<td>The resolveMinMaxConflict option variable determines which is selected.</td>
</tr>
<tr>
<td>numBroydens</td>
<td>Number of Broyden updates performed during the convergence of the current point. This counter is not reset when a new Jacobian matrix is generated.</td>
</tr>
<tr>
<td>numJacobians</td>
<td>Number of new Jacobian matrices generated during the convergence of the current point.</td>
</tr>
<tr>
<td>passCounter</td>
<td>Number of model execution passes (convergence iterations + perturbation passes).</td>
</tr>
<tr>
<td>passType</td>
<td>String containing description of the type of pass being made by the model. Possible values are: &quot;NoPass,&quot; &quot;firstPass,&quot; &quot;perturbationPass,&quot; and &quot;iterationPass.&quot;</td>
</tr>
<tr>
<td>perturbationCounter</td>
<td>Records the number of the perturbation pass while a Jacobian matrix is being generated. It is zero when passType is anything but perturbationPass.</td>
</tr>
<tr>
<td>scaleFactor</td>
<td>Largest scale factor (closest to zero) found when comparing the unlimited dx for each Independent against the allowed dx for that Independent. Applying this single scalar value to the dxUnlim vector will ensure that the change in each Independent variable is at or below its limit value. If scaling is required (scaleFactor &lt; 1.0), then the name and number of the limiting Independent is recorded in the dxLimIndepName and dxLimIndepIndex attributes.</td>
</tr>
<tr>
<td>toleranceRMS</td>
<td>The root mean square value of the normalized tolerance values of all the active Dependents. The normalized tolerance value is used regardless of the toleranceType in order to put all tolerance values in the same scale.</td>
</tr>
<tr>
<td>totalBroydens</td>
<td>Number of Broyden updates performed since the beginning of execution.</td>
</tr>
<tr>
<td>totalContinuousConvergences</td>
<td>Number of continuous model points converged since the beginning of execution. Includes reconvergence of the continuous portions of a model while converging any DSVs present.</td>
</tr>
<tr>
<td>totalIterations</td>
<td>Number of convergence iteration passes since the beginning of execution.</td>
</tr>
<tr>
<td>totalJacobians</td>
<td>Number of Jacobian matrices generated since the beginning of execution.</td>
</tr>
<tr>
<td>totalPasses</td>
<td>Number of model passes executed since the beginning of execution.</td>
</tr>
<tr>
<td>updateConverged</td>
<td>Array of individual dx convergence status(es)</td>
</tr>
</tbody>
</table>

16.2 Independent (NCPIndependent Class)

16.2.1 Independent Definition Example

```c
Element Assembly TurbineAssembly {
    // Element definitions, Solver definition, etc.

    // Define independent
    Independent ind_TurbinePR {
        varName = "TRB041.PRbase";
        dxLimit = 0.1; // Allow turbine PR to move by .1
        dxLimitType = "ABSOLUTE";
        initXFunction = "tan(xModel)"; // define mapping function
        xMappingFunction = "atan(x)"; // define inverse mapping function
        xMaxLimitWarn = 5.50;
        xMaxLimitFail = 6.00;
        xMinLimitWarn = 1.25;
    }
```

16  The NPSS Solver Reference 299
xMinLimitFail = 1.02;
}

// more model stuff

// set the autoSetup attribute to TRUE so that the next call to
// autoSolverSetup will include this Independent in the Solver Setup.
  ind_TurbinePR.autoSetup = TRUE;

// set another attribute outside the definition block
  ind_TurbinePR.perturbation = 0.01;
}

// set an attribute from outside of the assembly
  TurbineAssembly.ind_TurbinePR.indepRef = "2.5";

16.2.2 Independent Limits

The Independent class contains two groups of limit attributes. The original set attributes (xMaxLimitWarn, xMaxLimitFail, xMinLimitWarn and xMinLimitFail) actually do not limit the value the Independent passes to the model. Rather these attributes function to generate warning and error message if the model converges with independent values beyond the given limit values. These attributes are also single real number attributes, and so can not be defined, in and of themselves, as a function of any other parameter.

A second set of attributes has been added to the Independent. The primary purpose of these parameters is to place limits on the predictor value during a transient time step and the perturbation step during Jacobian generation. These limits provide a way to safeguard the model from values that represent a condition, such as negative pressure or area, that the simulation can not recover from during subsequent iterations to converge the time step.

These limits, like the original limit parameters, cannot limit the Independent during regular convergence iterations. Trying to limit the Independent value during convergence iterations will result in the allowed change to the Independent values to be zero once the limit has been reached. The result is that the scale factor on the Independent changes is zero and the Independents stop changing. So these limits, like the original set of Independent limits, only provide warnings if the limit value is exceeded at convergence. Application of the Dependent constraint system is required to keep the converged Independent value below the limit.

The xMinLimitExp and xMaxLimitExp parameters are string attributes which are assumed to contain an expression that when evaluated yields a single real number. This can be any valid NPSS expression including a function call. The results of these expressions are saved in the xMinLimit and xMaxLimit parameters. These are read-only parameters. Attempting to assign a value to the xMinLimit and xMaxLimit parameters will generate an error. The integer parameter xLimitCheck controls whether either set of the limit parameters is checked. Assigning an expression to either xMinLimitExp or xMaxLimitExp will have the side-effect of setting the xLimitCheck parameter to TRUE. The xLimitReport integer parameter controls whether cut-backs to limit values during prediction or perturbation or limit exceedence at convergence are reported or not. This parameter has no impact on whether the limits are checked or not, only whether the user is notified or not. So an xLimitCheck set to TRUE and xLimitReport set to FALSE will still limit the predictor to the limit, but will do so silently. And lastly the xLimitExceeded flag is an integer parameter that records whether a limit has been hit or exceeded. This parameter is set regardless of the value of xLimitReport.

16.2.3 Independent Input/Commands

void revertToDefaults()
Unlocks all "lockable" attributes in this independent.
### 16.2.4 Independent User Input Attributes

**Table 54. Independent User Input Attributes**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoSetup</td>
<td>An option which determines whether or not the Independent should be automatically added to the solver when autoSolverSetup() is called (TRUE or FALSE).</td>
<td>FALSE</td>
</tr>
<tr>
<td>dxLimit</td>
<td>The max allowed change for this particular independent variable. If not set, solver default value (solver.defaultDxLimit) is used.</td>
<td>solver.defaultDxLimit</td>
</tr>
<tr>
<td>dxLimitType</td>
<td>The method for computing dxLimit, as an ABSOLUTE value or as a FRACTIONAL portion of the xRef value. If not set, the default dxLimitType value set in the Solver is used.</td>
<td>solver.</td>
</tr>
<tr>
<td>indepRef</td>
<td>An expression that is evaluated to yield a reference value for the Independent. The result of the expression is stored in the xRef output attribute. Example: &quot;xDes&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>initXFunction</td>
<td>The expression for the mapping function between the model independent value and the solver value used internally. Should be the opposite in effect as expression as the xMappingFunction. For example, if xMappingFunction equals x*x, initXFunction should equal sqrt(x).</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>perturbation</td>
<td>The perturbation size for this particular model independent variable. If not set, the solver default value (solver.defaultPerturbation) is used.</td>
<td>solver.defaultPerturbation</td>
</tr>
<tr>
<td>perturbationType</td>
<td>Determines whether the perturbation value is used as an ABSOLUTE value or as a FRACTIONAL of the xRef value. If not set, the default perturbationType value set in the Solver is used.</td>
<td>solver.defaultPerturbationType</td>
</tr>
<tr>
<td>solutionMode</td>
<td>Local override for the overall Model's solutionMode. This attribute cannot be locked and so will be reset whenever the setOption is used at the Model level to set the overall solutionMode. Valid options are &quot;ONE_PASS,&quot; &quot;STEADY_STATE,&quot; &quot;TRANSIENT,&quot; and &quot;SET_DERIVATIVE.&quot;</td>
<td>&quot;STEADY_STATE&quot;</td>
</tr>
<tr>
<td>varName</td>
<td>The full path name to the independent variable in the NPSS model.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>xLimitCheck</td>
<td>Determines whether the limit checking on this xModel is performed.</td>
<td>FALSE</td>
</tr>
<tr>
<td>xLimitExceeded</td>
<td>Set to TRUE if the predicted or final value of x exceeded either xMinLimit or xMaxLimit. The value of xLimitReport does not affect whether this flag is set.</td>
<td>FALSE</td>
</tr>
<tr>
<td>xLimitReport</td>
<td>Controls reporting of whether the predicted value violated a limit and was cut back or if convergence exceeded the limits. The Solver parameter defaultXLimitReport provides a Solver wide default. If the local xLimitReport remains unset, the Solver default will be used. After setting a local value, to return to using the Solver default, the user must set the lockXLimitReport flag to FALSE.</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
### Attribute Name | Description | Default value
---|---|---
xMappingFunction | The expression for the mapping function between the solver internal independent and the model variable (should be the inverse of initXFunction). | "" 
xMaxLimitFail | Upper limit on model independent which will generate a provisional error indicating the value is out of bounds | 0.0 
xMaxLimitWarn | Upper limit on model independent which will generate a provisional warning indicating the value is high | 0.0 
xMinLimitFail | Lower limit on model independent which will generate a provisional error indicating the value is out of bounds | 0.0 
xMinLimitWarn | Lower limit on model independent which will generate a provisional warning indicating the value is low. | 0.0 
xMaxLimitExp | An expression that determines an actual hard maximum limit on predicted values during a transient simulation and the perturbed value when generating a matrix. | "" 
xMaxLimit | Real value resulting from evaluation of xMaxLimitExp | 
xMinLimitExp | An expression that determines an actual hard minimum limit on predicted values during a transient simulation and the perturbed value when generating a matrix. | "" 
xMinLimit | Real value resulting from evaluation of xMinLimitExp | 
xTolerance | The value used to test the convergence of this particular independent. If not set, the solver.defaultTolerance value used. | solver.defaultTolerance 
xToleranceType | Determines how xTolerance is used to test convergence of this particular independent. If not set, the solver default value (solver.defaultToleranceType) is used | solver.defaultToleranceType 
s | Scalar value applied to xRef via formula x*s+a. See default value of indepRef above. | 1 
a | Adder value applied to xRef via formula x*s+a. See default value of indepRef above. | 0 

### 16.2.5 Lock Attributes for Independents

The SolverExecutive sets some attribute default values for each of its associated independents. If you specifically set one of these attributes in an independent, the default value will be ignored. For each independent attribute which has a default in the SolverExecutive, the independent has a lock attribute.

When the corresponding attribute is set for a specific independent, the lock parameter is set to TRUE. When you want the default to be used again, you should set the corresponding lock attribute to FALSE again.

**Note:** The default value for all the "lock" variables below is FALSE.
Table 55. Independent Lock Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lockDxLimit</td>
<td>The default override lock on the dxLimit attribute.</td>
<td>FALSE</td>
</tr>
<tr>
<td>lockDxLimitType</td>
<td>The default override lock on the dxLimitType attribute</td>
<td>FALSE</td>
</tr>
<tr>
<td>lockPerturbation</td>
<td>The default override lock on perturbation attribute</td>
<td>FALSE</td>
</tr>
<tr>
<td>lockPerturbationType</td>
<td>The default override lock on the perturbationType attribute</td>
<td>FALSE</td>
</tr>
<tr>
<td>lockXLimitReport</td>
<td>The default override lock on the xLimitReport attribute</td>
<td>FALSE</td>
</tr>
<tr>
<td>lockXTolerance</td>
<td>The default override lock on the xTolerance attribute</td>
<td>FALSE</td>
</tr>
<tr>
<td>lockXToleranceType</td>
<td>The default override lock on the xToleranceType attribute</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

16.2.6 Independent Output (Calculated) Attributes

Table 56. Independent Output Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>baselineValue</td>
<td>Base value of the model independent for Jacobian generation</td>
</tr>
<tr>
<td>x</td>
<td>The solver independent value</td>
</tr>
<tr>
<td>xFirst</td>
<td>The value of x after one pass through the model (executionSequence has run once) at the start of the overall convergence attempt</td>
</tr>
<tr>
<td>xFirstErr</td>
<td>The percent difference between xFirst and x - ((xFirst - x)/x)*100</td>
</tr>
<tr>
<td>xFirstErrRef</td>
<td>The percentage result of (xFirst/xRef -1)*100</td>
</tr>
<tr>
<td>xModel</td>
<td>The model independent value</td>
</tr>
<tr>
<td>xModelFirst</td>
<td>The value of xModel after one pass through the model (executionSequence has run once) at the start of the overall convergence attempt.</td>
</tr>
<tr>
<td>xModelFirstErr</td>
<td>The percent difference between xModelFirst and xModel - ((xModelFirst - xModel)/xModel)*100</td>
</tr>
<tr>
<td>xModelFirstErrRef</td>
<td>The percentage result of (xModelFirst/xRef -1)*100</td>
</tr>
<tr>
<td>xPrevious</td>
<td>The solver independent value from the previous iteration</td>
</tr>
<tr>
<td>xRef</td>
<td>The independent reference value. If an expression is given in indepRef, then xRef is the result of evaluating that expression. If no expression is specified, then xRef is simply set equal the current value of x, unless x equals zero, in which case xRef is set equal to 1.0.</td>
</tr>
</tbody>
</table>

16.3 Dependent (NCPDependent class)

16.3.1 Examples of Dependent Definition

Element Assembly MainShaftAssembly {
    // Element definitions, Solver definition, etc.
    ...
    // define a dependent in which a single model parameter is driven to a constant value
    ShH {
        Dependent dep_netTrq {
            eq_lhs = "ShH.trqNet"; // the left-hand side of the
            // Dependent equality.
        }
    }
}
304 16  The NPSS Solver Reference

eq_rhs = "0.0000"; // value defaults to zero
tolerance = 0.1; // Override Solver default
toleranceType = "ABSOLUTE"; // Converged with +/- .1 ft-lb
}

// define a dependent in which two model variables will be
// driven equal to each other.
Dependent dep_matchSpoolSpeeds {
  eq_lhs = "LP_SHAFT1.Speed";
  eq_rhs = "LP_SHAFT2.Speed";
  tolerance = 0.0005;
}

// more model stuff

// add the dependent to the assembly's solver
solver.addDependent("ShH.dep_netTrq");
solver.addDependent("dep_matchSpoolSpeeds");

// set another attribute outside the definition block and revert back
// to using the default tolerance and tolerance type
ShH.dep_netTrq.eq_Ref = "1000.";
ShH.dep_netTrq.revertToDefaults();

// set an attribute from outside of the assembly
MainShaftAssembly.dep_matchSpoolSpeeds.tolerance = 0.0002;

16.3.1.1 Example of a Dependent with Constraints

To place constraints on a target Dependent, all constraint Dependents must be defined first. Constraint Dependents
should have their autoSetup attributes set to FALSE (the default), since they cannot exist in both the active and
auxiliary dependents list simultaneously.

// Define the Constraint Dependents first
Dependent T4Limit {
  eq_lhs = "Burner_Disch_Duct.Fl_O.Ts";
  eq_rhs = "1500.*Nozzle.Fn/RatedThrust";
  autoSetup = FALSE;
}

Dependent Overspeed {
  eq_lhs = "ShH.Nmech";
  eq_rhs = "CmpH.tipSpeedLimit";
  autoSetup = FALSE;
}

// Define the constrained Dependent
Dependent ThrustTarget {
  eq_lhs = "Nozzle.Fn";
  eq rhs = "RatedThrust";
  constraintNameList = { "T4Limit",  // list of constraint objects
                        "Overspeed"  // associated with this target
                      }
  constraintPriorities = {1, 2}; // T4Limit is top priority,
                              // then Overspeed
  limitTypes = { "MAX", "MAX" }; // both constraints are MAX types
  constraintSlopes = { 1, 1 }; // by default, the slope should be 1.
  solver.addDependent("ThrustTarget"); // add target and its two constraints
16.3.2 Dependent Input/Commands

void addConstraint (string name, string limitType, int priority, int slope);
Adds the named Dependent to the current Dependent's list of constraints, with the specified limit-type ("MIN" or "MAX"), priority (integer greater than 0), and slope (1 or –1).

void addConstraintGroup(string name);
Adds all of the constraint Dependent information contained in the named group to the Dependent's list of constraints. In addition to the names of the constraint, the limit-type ("MIN" or "MAX"), priority (integer greater than 0), and slope (1 or –1) specified for each is transferred from the ConstraintGroup to the internal storage in the Dependent.

string getLimitType(string name);
Returns the limitType attribute for the named constraint Dependent.

int getPriority(string name);
Returns the priority for the name constraint Dependent.

int getSlope(string name);
Returns the slope for the named constraint Dependent.

void invertConstraint(string name);
Changes the named constraint Dependent's limitType from "MIN" to "MAX" or vice versa and sets the associated constraint slope from 1 to –1 (or vice versa)

int removeConstraint(string name);
Removes the named constraint Dependent from the current Dependent.

void revertToDefaults();
Unlocks all "lockable" attributes in this dependent. This function is also available for objects derived from Dependent, such as the Integrator.

void setLimitType(string name, string limitType);
Sets the limitType for the named constraint to the input value ("MIN" or "MAX"). The limitTypes array elements can also be set directly by index.

void setPriority(string name, int priority);
Sets the priority for the named constraint to the input value. The constraintPriorities array elements can also be set directly by index.

void setSlope(string name, int slope);
Sets the slope for the named constraint Dependent to the input value. The constraintSlopes array elements can also be set directly by index.
16.3.3 Dependent User Input Attributes

Table 57. Dependent User Input Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoSetup</td>
<td>An option which determines whether or not the Dependent should be automatically added to the solver when autoSolverSetup() is called (TRUE or FALSE).</td>
<td>FALSE</td>
</tr>
<tr>
<td>constraintNameList</td>
<td>A list of names for Dependent objects which will act to constrain this Dependent's solution.</td>
<td>{ }</td>
</tr>
<tr>
<td>constraintPriorities</td>
<td>A list of integers specifying the priority of the corresponding constraint in the constraint name list. It is used strictly to break a tie between min-limit and max-limit constraints if both types are violated at any point. If no such conflict exists, the most restrictive constraint will be invoked, regardless of priority. Higher values indicates higher priority. If not set, all priorities default to 1.</td>
<td>{ }</td>
</tr>
<tr>
<td>constraintSlopes</td>
<td>An integer array of slope modifiers for the Dependent constraints. The elements of this array correspond to those in the constraintNameList. The assumed values of the array are limited to 1 and -1, although this is not enforced through error checking. The purpose of the constraintSlopes is to invert the variation of each constraint's error value with respect to the model independents. These slopes affect only the logic for invoking constraints. Typically, when an element of this array is switched from 1 (its default) to -1, the corresponding element of the limitTypes array is switched from MIN to MAX or vice-versa. Both actions can be performed using the invertConstraint method.</td>
<td>{ }</td>
</tr>
<tr>
<td>eq_lhs</td>
<td>Any valid expression that when evaluated yields a real value for the left-hand side of the dependent condition. The results of the expression evaluation are stored in y1. The expression can basically be anything that yields a real number when evaluated. This includes simple constants or single variable references. It can also be any mathematical expression referencing one or more variables. The expression can even be a function call that performs any arbitrarily complex calculations and returns a single real value.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>eq_rhs</td>
<td>Any valid expression that when evaluated yields a real value for the right hand side of the dependent condition. The result of the expression evaluation is stored in attribute</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>---------------</td>
</tr>
<tr>
<td>y2.</td>
<td>eq_Ref</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>Any valid expression that when evaluated yields a real number reference value. The absolute value of the results of the expression evaluation is stored in attribute yRef. The error term, stored as attribute errorCon, is calculated as (y1-y2)/yRef, and is used to check convergence. The value of yRef at the point where a Jacobian is generated is stored in the attribute yRefLock and is used to calculate the error term errorIter. When an expression is not given for eq_Ref (or it evaluates to zero), the value of eq_rhs (y2) is used as the reference value. If eq_Ref is not set and y2 is equal to zero, then a yRef value of 1 is used.</td>
<td></td>
</tr>
<tr>
<td>limitTypes</td>
<td>A list of constraint types (min or max) corresponding to the constraints in the constraintNameList. Allowed values are &quot;MIN&quot; or &quot;MAX&quot;.</td>
<td>{}</td>
</tr>
<tr>
<td>resolveMinMaxConflict</td>
<td>Option which determines what constraint will be invoked if a min and a max constraint are both violated at the same priority level. Options are &quot;MIN&quot; and &quot;MAX.&quot;</td>
<td>solver.resolveMinMaxConflict</td>
</tr>
<tr>
<td>testRealValid</td>
<td>If set to TRUE, then the error returned by the Dependent will always be checked to verify that it is a valid number (not an NaN or inf). If the test is performed and the error is found to be not a valid number, then the y1, y2, and yRef terms are checked to see which of these is not a valid number.</td>
<td>TRUE</td>
</tr>
<tr>
<td>tolerance</td>
<td>The tolerance used to test the convergence of this dependent error term. If no value is set, the solver.defaultTolerance value is used.</td>
<td>solver.defaultTolerance</td>
</tr>
<tr>
<td>toleranceType</td>
<td>Determines how tolerance is used to determine convergence for this dependent error term. The options are &quot;FRACTIONAL&quot; or &quot;ABSOLUTE.&quot; If no value is set, the solver.defaultToleranceType is used. When toleranceType = &quot;ABSOLUTE&quot;, the given absolute tolerance value is divided by yRef to yield a fractional tolerance for comparison to errorCon. This fractional tolerance is recalculated each iteration so that the convergence is to the given absolute tolerance value.</td>
<td>solver.defaultToleranceType</td>
</tr>
<tr>
<td>useConstraints</td>
<td>When FALSE, the constraints associated with the Dependent are disabled. This is an easy way to disable constraints without clearing the constraintNameList.</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
16.3.4 Lock Attributes for Dependents

The SolverExecutive sets some attribute default values for each of its associated dependents. If you specifically set one of these attributes in an dependent, the default value will be ignored. For each dependent attribute which has a default in the SolverExecutive, the dependent has a lock attribute.

When the corresponding attribute is set for a specific dependent, the lock parameter is set to TRUE. When you want the default to be used again, you should set the corresponding lock attribute to FALSE again.

Table 58. Dependent Lock Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lockResolveMinMaxConflict</td>
<td>The default override lock on resolveMinMaxConflict</td>
<td>FALSE</td>
</tr>
<tr>
<td>lockTolerance</td>
<td>The default override lock on tolerance</td>
<td>FALSE</td>
</tr>
<tr>
<td>lockToleranceType</td>
<td>The default override lock on toleranceType</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

16.3.5 Dependent Output (Calculated) Attributes

Table 59. Dependent Output Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflictingMax</td>
<td>When minMaxConflict is TRUE, this attribute contains the path name of the conflicting max constraint.</td>
</tr>
<tr>
<td>conflictingMin</td>
<td>When minMaxConflict is TRUE, this attribute contains the path name of the conflicting max constraint.</td>
</tr>
<tr>
<td>constraintGroupNames</td>
<td>1-D string array containing the names of any constraint groups used to associate groups of constraints with the target Dependent</td>
</tr>
<tr>
<td>deltaErrorCon</td>
<td>The difference between the errorCon on the current and previous iterations. On the first iteration deltaErrorCon is equal to errorCon</td>
</tr>
<tr>
<td>errorCon</td>
<td>The error used to determine convergence. This error will always be the normalized or &quot;non-dimensional&quot; difference between the left- and right-hand sides of the dependent condition divided by ( yRef ). errorCon = ( (y1 - y2)/yRef ) regardless of the value of the Solver useNormalizedError flag. If the tolerance given is an absolute tolerance (as determined by toleranceType), the given tolerance will be normalized internally before comparison to this error.</td>
</tr>
<tr>
<td>errorConFirst</td>
<td>The first value of errorCon. It does not change if a DSV changes state and the continuous model is reconverged.</td>
</tr>
<tr>
<td>errorIter</td>
<td>The error used to form the Jacobian matrix and then during subsequent iterations with the LU decomposed Jacobian to calculate the unlimited dx values. Whether this error is calculated as either a normalized or &quot;non-dimensional&quot; (errorIter = ( (y1-y2)/yRefLock )) or a delta (errorIter = ( y1 - y2 )) is controlled by the Solver useNormalizedError flag.</td>
</tr>
<tr>
<td>errorProjected</td>
<td>The error value that is anticipated at the end of the current iteration (based on current independents and Jacobian matrix). This parameter is used for constraint selection and is calculated only when the solver maxConstraintProjections attribute is non-zero. This error is always expressed as an error normalized by ( yRef ), so that it can be used in the same way as errorCon to determine constraint selection.</td>
</tr>
<tr>
<td>minMaxConflict</td>
<td>An integer flag that is set to TRUE when a min type and max type constraint associated with the given target Dependent are simultaneously violated</td>
</tr>
<tr>
<td>y1</td>
<td>The value of eq_lhs expression</td>
</tr>
</tbody>
</table>
### Attribute Name | Description
--- | ---
y1First | The first value of `eq_lhs` expression in each overall convergence. It does not change if a DSV changes state and the continuous model is reconverged.
y2 | The first value of `eq_rhs` expression.
y2First | The first value of `eq_rhs` expression. It does not change if a DSV changes state and the continuous model is reconverged.
yRef | The value of the `eq_Ref` expression.
yRefFirst | The first value of `eq_Ref` expression. It does not change if a DSV changes state and the continuous model is reconverged.
yRefLock | The value of `yRef` at the point the current Jacobian matrix was generated. When a new matrix is generated, this attribute is updated.

### 16.4 ConstraintGroups (ConstraintGroup class)

#### 16.4.1 ConstraintGroup Input/Commands

**void addConstraint (string constraintDepName, string limitType>, int priority, int slope)**

Adds the named dependent object (which must have been previously defined) to the ConstraintGroup.

**int containsConstraint(string constraintDepName)**

Returns TRUE (1) if the named constraint is contained by the group. Returns FALSE (0) if the named constraint is not contained by the group.

**void invertConstraint(string constraintDepName)**

Reverses the limit type from MAX to MIN or the reverse, and changes the sign of the slope indicator integer associated with the named constraint.

**int removeConstraint (string constraintDepName)**

Removes the named dependent object from the ConstraintGroup. Returns TRUE (1) if the named constraint was found and removed from the group. Returns FALSE (0) if the named constraint was not contained by the group.

#### 16.4.2 ConstraintGroup User Input Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraintNameList</td>
<td>1D string array containing the path names of the constraints to be contained by the group</td>
<td><code>{ }</code></td>
</tr>
<tr>
<td>limitTypes</td>
<td>1D string array containing the limit type (either &quot;MAX&quot;, or &quot;MIN&quot;) that corresponds to the constraints contained in the constraintNameList.</td>
<td><code>{ }</code></td>
</tr>
<tr>
<td>constraintPriorities</td>
<td>1D integer array containing the priorities (positive, non-zero integers) that corresponds to the constraints contained in the constraintNameList.</td>
<td><code>{ }</code></td>
</tr>
<tr>
<td>constraintSlopes</td>
<td>1D integer array containing the slope indicators (either −1 or 1) that corresponds to the constraints contained in the constraintNameList.</td>
<td><code>{ }</code></td>
</tr>
</tbody>
</table>
16.4.3 ConstraintGroup Output (Calculated) Attributes

There are no calculated attributes available from this class.

16.5 DiscreteStateVariable (DSV)

_DiscreteStateVariables (DSVs)_ describe aspects of the model that can only have specific values. A classic example is a bleed selector valve. When engine speed is low, and hence compressor pressure is low, the selector valve is set to draw all customer bleed air from the compressor discharge. At some point, as power is increased, the pressure at the interstage port is sufficient, and the selector valve, from a steady-state perspective, abruptly transitions to taking all bleed air from this port. There is no intermediate steady-state setting where bleed air is pulled from both stations. The bleed valve can therefore be seen to have two discrete settings, or states. DSVs are not restricted to two states. For example, for a large compressor there may be two intermediate bleed ports in addition to the discharge port.

Changing between discrete states causes a discontinuity in the model output. The impact of the discrete change on the performance of the model may be that it tends to drive the model back across the discontinuity, causing the model to endlessly toggle between the two states. This can make convergence difficult to achieve at or near the boundary between discrete states.

The DSV objects provide a means for the Solver to control the behavior of the model when there are discrete state variables present. The DSV object sits between the logic that calculates the demand value of the discrete state (determining where the bleed air should be drawn, for example) and the portions of the model that use the discrete state as input (the bleed valve itself). The starting guess for the DSV state variable is set to an initial value via the DSV stateInitial variable which can be set through user input. If the user does not set the stateInitial, the value used is simply the converged value of the state from the previous point.

During the execution of the model the state value is calculated based on other parameters in the model. This calculated value of the state is termed the state demand and is assigned to the DSV stateDemand variable. The Solver compares the state value used by the model to the calculated stateDemand value. If they are the same, then the DSV is converged. Since these are discrete values, no tolerance factor is involved. If they are not the same, the state value is set equal to the stateDemand and another iteration is run. This process continues until all DSVs and Dependents are converged or until the maximum continuous or discrete number of iterations is reached.

When a DSV is toggled after a continuous solution, the internal counters that track the number of iterations are effectively reset. This means that the maximum iterations test is applied for each continuous solution. For example: If maxIterations=25 and there are 3 DSVs active, all of which have to systematically toggle, this will require 8 continuous solutions. If each continuous solution takes 20 iterations, the internal iteration counter will be 160 at the final solution. The point would not fail because none of the continuous solutions ever exceeded 25.

The DSV state can be set equal to the stateDemand either between iterations converging the continuous portion of the model (termed free toggling) or between continuous model convergences as part of an outer discrete iteration loop. Allowing the DSV state to free toggle while trying to converge the continuous model has the potential to cause convergence stability problems. Therefore, a limit, via the Solver and/or DSV maxFreeToggles variable, is placed on how many times a DSV can free toggle before it is locked and the state value for that particular DSV is only allowed to change outside the continuous convergence loop.

The default behavior is to not allow free toggling (maxFreeToggles defaults to zero). If free toggling is enabled, care should be taken to make sure that when the model is being converged at or near a point where the DSV changes state, that doing so doesn’t cause the model to chatter needlessly back and forth across the discontinuity. One possible solution is a Schmidt trigger in which the logic that determines the DSV stateDemand provides for an overlap or hysteresis band in which the calculated stateDemand has two values depending on from which “side” the current iteration started. For the bleed valve example, the selector logic might move the stateDemand from discharge to interstage at a compressor discharge pressure of 102 psia, but not move from interstage back to
compressor discharge until the pressure falls to 98 psia. The result is a 4 psia hysteresis band. If one or more DSVs hit the maximum allowed number of free toggles during continuous model convergence, or if free toggling is not permitted, then convergence of the DSVs is moved to a loop around the convergence of the continuous model. In this situation the stateDemand value is passed through to the state value of the DSV only after the continuous model is converged. If any DSV state value changed, the entire continuous model is reconverged with the new state value(s). This process repeats until all the DSVs are converged. If after the maximum number of these discrete iterations the DSVs are still not converged, then the point is reported to have failed to converge, even though the continuous portion might have successfully converged with the given state values. And of course, if the continuous model cannot find a converged solution with the given state value(s), then the point also returns as failed to converge. Diagnostics are available to report the problem was in the continuous or discontinuous iterations.

16.5.1 Nested vs. Simultaneous DSV Solution

A model may contain more than one DSV. The user may select a simultaneous, nested or combination approach to solving those DSVs. A simultaneous solution is achieved by updating the state value of all DSVs at the same time and reconverging the continuous portion of the model with the new set of discrete state values, checking to see if all the DSVs are converged (state equals stateDemand), and continuing to iterate if they are not. However, updating all the DSVs at once may introduce instabilities to the convergence process. The nested solution approach is designed to address this situation.

In the pure nested approach, each DSV is assigned its own solution level, using the level attribute in each DSV. When the Solver attempts to converge the discrete portions of the model, it begins iterating the lowest level DSV. If needed, it updates the state value, reconverges the continuous model and checks the DSV for convergence. If the DSV is converged, the Solver moves to the next higher level DSV and checks to see if it too is converged. If it is not, the Solver updates the state value with the demand value, resets lower level DSVs using their stateInitial value, and begins again at the lowest level DSV and recovers each level. This continues until the DSVs at all levels are converged. Since each discrete iteration requires a complete convergence of the continuous model, the computational requirements to converge a single operating point grows exponentially with each additional DSV level.

The Solver can accommodate a combination of nested and simultaneous discrete solutions. If more than one DSV exists at the same solution level, the Solver will only move on to the next higher level when all the DSVs at a given level are converged.

16.5.2 Controlling Movement through the Allowed State Values

The default behavior is to allow the state value to move directly to the stateDemand value when instructed by the Solver to do so. However, at times it may be advantageous to restrict the movements through the possible state value to be sequential. The sequential attribute in each DSV object controls this behavior. When set to TRUE, the state value of the DSV will only move to the next higher or lower value in the list of allowed state values. The stateDemand value is then only used to determine which direction to move.

16.5.3 Controlling Repetition of Previously Failed Iterations

When the DSVs are configured in the Solver in a nested solution structure with only one DSV per level, it is unproductive for any DSV to go back to a value that has already been attempted during a specific convergence attempt of that DSV. For example, if a DSV has state values of "A", "B", "C" and "D", and a solution with a value of "B" has already been attempted without success, then "B" will not yield a solution if tried again. When this happens, the solution is said to be "bouncing" back to "B". The ability to detect and accommodate this bounce condition has been built into each DSV. This is not true when trying to reconverge a lower level DSV when a higher level DSV has changed state and all lower level DSVs are being reconverged. Then it is acceptable for the state to have values that were tried in previous convergences.
Before updating the state value, the new value is checked against the list of previous values attempted during the current convergence attempt. The previous values are stored in the attribute DSVPath. If the new value is has already been tried, the bounce attribute is set to TRUE. The input attribute controlBounce is checked to determine if any action should be done if a bounce is detected. If controlBounce is TRUE, then the option attribute resolveBounce is used to determine how to resolve the situation. Until the DSV is reset, all attempts to change the state value will be ignored. Also a DSV in a bounce situation will report that it is converged in the hope that an iteration of a higher level DSV will help resolve the repeating loop the current DSV is caught in. The Solver will not, however, report the discrete portion of the model as converged as long as the bounce attribute of any of the DSVs is still TRUE. This basically is a last attempt to find convergence for a model that is clearly going in circles.

Controlling this type of bounce situation is only compatible when DSVs are being solved in the nested style of solution as discussed in section 16.5.1. The simultaneous solution methodology couples the solution of the discrete and continuous portions of the model. It is not possible to determine if a DSV is bouncing because the continuous portion on which it depends is still not converged.

Further bounce control requires that only one DSV be present at each level. If there are two or more DSVs at a given level, that level is “bouncing back” to a previously tried solution only if the combination of states for each of the DSVs at that level have been tried previously. Currently there is no built-in way to detect this repetition of multiple states, though user code placed in the userReport function might be an avenue to detect the repetition of multiple states.

### 16.5.4 Example Definitions of DiscreteStateVariables

A DSV can be created individually at any point in the model. It can also be defined as part of an element or subelement class definition. As part of a class, a DSV is created for each instance of the class created in a model. To function, a DSV, like Independents and Dependents, has to be added to the Solver. This is done manually using the addDSV function on the Solver. Alternatively, the autoSetup flag in the DSV can be set to TRUE, in which case the DSV will automatically be added to the Solver each time the autoSolverSetup function is run.

```c
// Create a DSV at the top level outside the scope of any Element

DSV checkValveState {
    allowedValues = { "OPEN", "CLOSED" }
    autoSetup = FALSE;
}

// Manually add this DSV to the Solver

solver.addDSV("checkValveState");

// Creating a DSV in the scope of an element.

Element Compressor CmpH {
    DSV BldValvePos {
        allowedValues = { "IP1", "IP2", "CDP" }
        level = 1;
        controlBounce = TRUE;
        autoSetup = TRUE;
    }
}

// Running autoSolverSetup clears the Solver of all DSVs
// and then automatically adds those with autoSetup == TRUE
// Those DSVs with autoSetup == FALSE can be added Manually
// after autoSolverSetup() has run.

autoSolverSetup();  // Only CmpH.BldValvePos in Solver
Solver.addDSV("checkValveState");  // add back manually
```
// Defining a DSV as part of an element class.

class Demo extends Element {
    real x, y, z;
    DSV discreteVar {
        allowedValues = { "TOP", "MIDDLE", "BOTTOM" }
        autoSetup = TRUE;
        }
    void calculate() {
        ...
    }
}

// A discreteVar DSV with the allowedValues defined in the
// class definitions is created each time an instance of
// Demo is created
Element DemoDemo1; // DSV Demo1.discreteVar now exists
Element DemoDemo2; // DSV Demo2.discreteVar now exists
autoSolverSetup(); // Demo1.discreteVar, Demo2.discreteVar
// and CmpH.BldValvePos added to Solver

16.5.5 Using a DSV in a Model

A DSV serves to hold and control the value of a discrete state variable. It does not determine what the state value
should be. Instead the model must contain logic to calculate a demand value for the discrete state. The calculated
or demand value is then passed to the DSV. The logic within the DSV then determines whether or not to set the
state value to the demand value, and when to do so. The following illustrates this process using a bleed valve that
has three discrete settings.

The DSV is created as part of the HP compressor. The logic that utilizes the DSV state value is placed in the
preexecute function of the compressor since the active bleed port and the amount of bleed air from that port
needs to be determined before the compressor is executed. The logic that determines the position of the bleed
valve is in the compressor postexecute function since the data required to make the selection is only available
after the compressor has been executed. The DSV serves to pass the calculated bleed valve position from the
postexecute function to the preexecute function and to make sure that the value used is the same as the value
calculated.

Element Compressor CmpH {
    InterStageBleedOutPort Bld_IP1 {
        fracBldWork = 0.45; fracBldP = 0.4;
    }
    InterStageBleedOutPort Bld_IP2 {
        fracBldWork = 0.65; fracBldP = 0.6;
    }
    InterStageBleedOutPort Bld_CDP {
        fracBldWork = 1.0; fracBldP = 1.0;
    }
    // The DSV has three possible states with the initial
    // demand value set at creation to be "CDP".
    DSV BldValvePos {
        allowedValues = { "IP1", "IP2", "CDP" }
        autoSetup = TRUE;
        stateInitial = "CDP"; // Initial guess
        sequential = TRUE;
    }
314 16  The NPSS Solver Reference

16.5.6 DSV Integer Associations

Each allowable string value is also associated with an integer value. By default, the allowable strings are associated from left to right with consecutive integers beginning with zero. Thus, in the BldValvePos example above, "IP1" is associated with 0, "IP2" with 1, and "CDP" with 2.

The integer value associated with the current string value of a DSV attribute (state, stateInitial or stateDemand) is held in attribute intValue. Setting attributes stateInitial and stateDemand to an integer value also sets the string value:

```plaintext
BldValvePos.stateInitial = 1;  // Sets BldValvePos.stateInitial to "IP2"
BldValvePos.stateDemand = 2;  // Sets BldValvePos.stateDemand to "CDP"
```

Thus integer associations provide a second way for the user to set the values of stateInitial and stateDemand. An error results if the user attempts to assign an integer value not associated with one of the allowable string values. An error will also result if the user attempts to set intValue or set state because these are both DSV outputs.

When a DSV is created, the user may assign integer values other than the defaults to the allowable string values of the variable. This is illustrated in the following example:
DSV dsv_days;
dsv_days.allowedValues = { "TU=3", "SU=1", "MO=2.6", "WE", "TH", "FR", "SA" };

Any integer values can be assigned – they need neither be consecutive nor ascending. If a real value is provided, it is truncated to an integer ("MO" is associated with integer 2). If integer values are not explicitly assigned to some string values, consecutive default integers are assigned, left to right, beginning with one greater than the largest integer explicitly assigned. Thus "WE" is associated with 4, "TH" with 5, "FR" with 6, and "SA" with 7.

Note that the DSV state attribute itself is a string. Thus printing variable dsv_days.state will print a string, not an integer. The value of dsv_days.state can only be compared with operators == or != (see Section 2.2.5.1) to another string, not to an integer.

    if(dsv_days.state == "SU")  {  }

intValue may be used for integer comparisons:

    if(dsv_days.state.intValue == 1)  {  }

16.5.7 DSV allowedStrings and allowedInts

In addition to allowedValues and intValue, DSVs have two other special attributes accessible to the user. Each of these is an array, and can only be read by the user (not written to).

In the example above, dsv_days.allowedStrings = { "TU", "SU", "MO", "WE", "TH", "FR", "SA" } and dsv_days.allowedInts = { 3, 1, 2, 4, 5, 6, 7 }.

16.5.8 DSV Input Commands

Other than to set attribute values and options, there are no user commands to the Discrete State Variable objects.

16.5.9 DSV User Input Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowedValues</td>
<td>Lists the allowed discrete state values (as strings). The allowedValues must be defined for every DSV created. The number and content of the strings can be different for each DSV, but they do not have to be. An error message will be generated if the stateDemand or stateInitial are set to anything other than one of the strings in this list. The order that the allowed values are specified is important if the movement of the state value through the allowed value is restricted to being sequential, since the state value can only move from its current value to the adjacent values in the allowedValue list.</td>
<td>{}</td>
</tr>
<tr>
<td>allowedStrings</td>
<td>String values to which the option variable may be set.</td>
<td></td>
</tr>
<tr>
<td>allowedInts</td>
<td>Integer values associated with the allowable string values (and to which attribute intValue may have been set).</td>
<td></td>
</tr>
<tr>
<td>controlBounce</td>
<td>Determines if action above and beyond locking the DSV should be done if the bounce flag is TRUE. If controlBounce is set to TRUE, then the resolveBounce attribute is used to determine the action. If controlBounce is set to FALSE, then the bounce flag is ignored and the Solver is allowed to update the state value.</td>
<td>FALSE</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>description</td>
<td>The description field for defining a generic DSV.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>iDescription</td>
<td>The description field for defining a specific DSV.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>level</td>
<td>An integer value that determines the solution level in which the DSV is solved. The Solver converges the DSV from the lowest level to the highest level. It is allowable to have solution levels without a DSV (e.g., a model with three DSVs can have level values of 1, 5, and 8). It is also allowable to have more than one DSV with a given level value. When there is more than one DSV at a given level, all the DSVs at that level are iterated as a group and all must be converged before convergence of the next higher level DSV(s) is (are) attempted.</td>
<td>0</td>
</tr>
<tr>
<td>maxFreeToggles</td>
<td>The number of times the DSV may change state while the continuous portion of the model is being converged before the DSV is locked and converged in the outer discrete iteration loop</td>
<td>0</td>
</tr>
<tr>
<td>resolveBounce</td>
<td>Determines the action to take if controlBounce is TRUE and a bounce situation is detected. &quot;HOLD&quot; indicates that the current state value should be retained. &quot;HIGH&quot; indicates that if the stateDemand value has a higher index value in the allowedValues array than the state, then the state is set to the stateDemand. &quot;LOW&quot; indicates the converse. &quot;INITIAL&quot; indicates that if the stateDemand is taking the state towards the stateInitial in the array of allowed values, then set the state to the stateDemand. Otherwise, keep the state at its current value. Movement of the state in response to a bounce situation is still moderated by the sequential flag. If the sequential flag is TRUE, then the state is still restricted to moving only one index location in the indicated direction.</td>
<td>&quot;HOLD&quot;</td>
</tr>
<tr>
<td>sequential</td>
<td>Integer that controls the movement of the DSV state value through the list of allowed values. If TRUE, the state value is allowed to move only to the next higher or lower allowed value regardless of the stateDemand value. The stateDemand value is used to determine the direction the state value moves. If FALSE, then the state is allowed to move directly to the stateDemand.</td>
<td>FALSE</td>
</tr>
<tr>
<td>stateDemand</td>
<td>The demand, or calculated value of the discrete state, is based on model conditions. The stateDemand value must be in the allowedValues array or an error message is issued. The state is set to the stateDemand only between convergences of the continuous model. However, before doing this, the stateDemand value is compared to the state value. If they are the same, then the DSV is considered converged.</td>
<td>allowedValue(0)</td>
</tr>
<tr>
<td>stateInitial</td>
<td>Serves as a means to set an initial guess for the state value. When assigning a value to this attribute, it is first saved in this attribute, which remains the same for the rest of the convergence attempt. In addition, the state attribute is also directly set to the given value, the bounce and lockFlag attributes are set FALSE, and the DSVpath is cleared and the stateInitial value inserted as the first element of a new DSVpath. The value of the sequential flag is ignored in this situation. If the user does not set stateInitial, then stateInitial is set to the converged value of state from the previous point. stateInitial can be set at any point, but care must be taken since setting this attribute short-circuits the real</td>
<td>allowedValue(0) or previously converged state value</td>
</tr>
</tbody>
</table>
16.5.10 DSV Output (Calculated) Attributes

Table 62. DSV Output Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounce</td>
<td>A integer flag indicating that the DSV was passed a value that has already been attempted since the DSV was locked. The controlBounce and resolveBounce attributes control how the DSV responds when a bounce condition is detected.</td>
</tr>
<tr>
<td>DSVpath</td>
<td>A string array containing all of the state values to which the DSV was set since the DSV was reset. A DSV is reset at the beginning of each new point, when the stateInitial value is set and when a higher level DSV is iterated. The reset path is initialized with the value of stateInitial.</td>
</tr>
<tr>
<td>lockFlag</td>
<td>The lockState is set to TRUE (1) by the DSV if setting the state value to (or towards) the stateDemand value would cause the counter for the allowed value to which the state would be set to exceed the maxFreeDisconToggles value. This check is done prior to actually setting the state value, so the state value is not changed if doing so would violate the toggling limit. When the DSV is locked, the Solver moves the updating of the state value from between every iteration of the continuous model to between convergences of the continuous model.</td>
</tr>
<tr>
<td>lockMaxFreeToggles</td>
<td>Set by automatically to TRUE when maxFreeToggles is set by the user. Manually set to FALSE to return to using the Solver default value.</td>
</tr>
<tr>
<td>numFreeToggles</td>
<td>The number of times the DSV changed state during continuous model convergence. If equal to the maxFreeToggles, the DSV is locked</td>
</tr>
<tr>
<td>state</td>
<td>The attribute that should be used by code that uses the value of the DSV as input. Attempts to set the DSV value must go through the stateDemand attribute. The exception is that before a point is run, setting the stateInitialValue attribute sets the state value, thus providing a mechanism to make an initial guess of the state value. Attempts to directly set this attribute will result in an error message and the input will be ignored. The DSV controls how and when this attribute is updated. If the lockState is FALSE, then this attribute is updated between iterations of the continuous model. If the lockState is TRUE, the Solver updates this attribute only between convergences of the continuous model.</td>
</tr>
</tbody>
</table>

16.6 Advanced Solver Member Functions

The following user-accessible member functions are intended for advanced users. If desired, these functions can be used to define a solution process different from that which is normally used in the NPSS. These functions comprise the basic steps used in obtaining a solution (see NPSS Software Design Document).
The user cannot change how each step in the process is implemented but can modify the order in which these steps are executed and under what conditions.

**Example 1**

To demonstrate the potential usefulness of this capability, consider the following example. The NPSS solver normally performs a Broyden update on every solver iteration, using the error convergence rate to determine when a new (rather than updated) Jacobian matrix must be generated. A given user may instead prefer to perform a Broyden update only when the convergence rate is unacceptable. They may prefer to generate a new Jacobian after a certain number of Broyden updates have been performed. To implement this alternative algorithm, the following function would be created by the user.

```c
void runMySolver() {
  // create a local variable to test convergence and initialize
  // it to FALSE.
  int errsConverged, totalBroydens;
  errsConverged = 0; totalBroydens = 0; solver.numBroydens = 0;

  // generate an initial Jacobian.
  solver.generateJacobian();
  solver.numJacobians = 1;

  // collect errors for solver to use
```
solver.updateErrors();
solver.iterationCounter = 0;

// begin iteration loop.
do {

    // increment the iterationCounter.
solver.iterationCounter = solver.iterationCounter + 1;

    // set the independents to drive errors to zero.
solver.updateIndependents();

    // run the model and compute the errors, check to see if they
    // are all converged.
errsConverged = solver.updateErrors();
if (errsConverged) {
    solver.converged = TRUE;
    break;
} else {
    solver.converged = FALSE;
}

    // check the RMS error convergence rate; if it is below a
    // given threshold, update the Jacobian using the Broyden
    // technique.
if (solver.rmsConvRate < 0.30) {
    solver.doBroydenUpdate();
solver.numBroydens = solver.numBroydens + 1;
totalBroydens = totalBroydens + 1;
}

    // after a certain number of Broyden updates have been
    // performed, generate a new Jacobian from scratch.
if (numBroydens > maxBroydens) {
    solver.generateJacobian();
solver.numJacobians = solver.numJacobians + 1;
solver.numBroydens = 0;
}
} while ( !errsConverged &&
    solver.iterationCounter <= solver.maxIterations &&
    solver.numJacobians <= solver.maxJacobians );

// end of the iteration loop
// test for conditions which terminated the process.
if (errsConverged) {
    cout << "Model converged in 
    << solver.iterationCounter << " iterations, 
    << solver.numJacobians << " Jacobians, and 
    << totalBroydens 
    << " total Broyden updates" << endl << endl;
} else if (solver.iterationCounter > solver.maxIterations) {
    cout << "Model failed to converge in 
    << solver.maxIterations 
    << "iterations." << endl << endl;
} else if (solver.numJacobians > solver.maxJacobians) {
    cout << "Model failed to converge with 
    << solver.maxJacobians 
    << " new matrix generations." << endl << endl;
}
else {
    cout << "Model failed to converge but reason is unknown" << endl << endl;
}

} // end of the custom solver convergence function.

Example 2

To run a single pass through the model, users must write their own function (a single-pass member-function is not built into the solver). Examples of how this might be done is shown below.

```cpp
void runSolverPass() {
    int i;
    // iterate over objects in the execution sequence.
    for (i=0; i < executionSequence.entries(); i++) {
        // execute each object in the sequence.
        executionSequence[i]->execute();
    }
}

void runPreSolverPass() {
    int i;
    // iterate over objects in the pre-execution sequence.
    for (i=0; i < preExecutionSequence.entries(); i++) {
        // execute each object in the sequence.
        preExecutionSequence[i]->execute();
    }
}
```

16.7 Solver Diagnostic Output

The NPSS solver produces a great deal of data at every iteration of the model during the convergence process. A record of this data is often useful in troubleshooting a model that fails to converge. The volume and format of the solver diagnostic data does not easily lend itself to the report-generation functions of the NPSS, and so this capability has been built into the solver software.

As a user, you can select from several levels of diagnostic detail. Each level of detail adds information to the one below it. The desired level is set using the Solver debugLevel attribute.

- `debugLevel = "ITERATION_DETAILS";`  
  Data on individual independents, dependents, and states at each iteration, including those which have not failed convergence at each step.

- `debugLevel = "ITERATION_FAILURES";`  
  Provides data on individual independents, dependents, and states including only those which fail convergence at each step.

- `debugLevel = "MATRIX_DETAILS";`  
  Provides data about generation and update of the Jacobian matrix and its normalized inverse.

- `debugLevel = "NONE";`  
  No special solver diagnostic is generated.

- `debugLevel = "RUN_SUMMARY";`  
  Provides overall statistics on the solver performance during each converged point.

- `debugLevel = "SETUP_DETAILS";`  
  Provides data on solver setup.
The default value of debugLevel is "NONE."
You may specify the name of the file you want the output sent to by setting the Solver attribute as shown below:

```python
solver.diagnosticFile = "mySolverOutput";
```

If you do not set this attribute, the output is sent to the standard error stream.

**Note:** If you are using multiple solvers, each solver can have its own diagnostic output file.

### 16.7.1 Searching Diagnostic Output

The diagnostic output contains certain keywords which can be used to file desired information using file-editing software or the UNIX "grep" command.

The following rules apply for finding information with a text-editing program:

- The setup information section begins with the words "SOLVER SETUP" and ends with "END SOLVER SETUP INFORMATION."
- When a Jacobian matrix is being generated from scratch, the output for that process begins with the words "JACOBIAN BEING GENERATED."
- When the solver is performing a Broyden update, the output generated starts with the words "BROYDEN UPDATE."
- When the constraint handling logic indicates a change in dependents, the potential changes are noted in a message containing the words "indicates that." The rest of the message includes the names of specific dependents and constraints; unless you want information about a specific ConstrainedTarget, the message is not useful for searching.
- When a change has actually been made in the dependents for constraint handling, a message is output which contains the words "has been switched." The rest of the message includes the names of specific dependents and constraints, so most of it is not useful for searching unless you want information about a specific named dependent or constraint.
- Information about the final state of model convergence can generally be found by searching for the words "convergeContinuousModel" or "Discontinuous convergence." If you are looking for problems, search for the word "FAILED" or "failed."

You can use any number of key phrases to find desired information in the output, and you will eventually decide on the keywords you find most useful. The rules above are intended only as a starting point.

### 16.7.2 Grepable Output

The UNIX "grep" command is useful for finding and displaying single lines which contain a specified keyword. The perturbations and updates to the solver independents, and the dependent errors on each iteration of the convergence process are output with a coded string at the end of each line to facilitate the use of `grep`.

Independent perturbations include a string starting with the letters "PX," followed by the number index of the independent being perturbed. For example, the perturbation of independent 5 (the fifth independent in the list) would include the string `PX5`.

The dependent response to each independent perturbation also includes the letters "PY," followed by the number index of each dependent. The response of the third dependent to perturbation of the fifth independent would therefore contain the string `PX5PY3`.

If you perform a grep on the string "PX5," you will see the perturbation of the fifth independent, followed by the responses of all dependents to that perturbation, for every Jacobian generation in the run.

Independent updates for each iteration contain the letters "DX," followed by the number index of the independent being updated, followed by the letter "I," followed by the number of the current iteration. Therefore, the independent update of the sixth independent in the twenty-third iteration would contain the string `DX6I23`. 
If you perform a grep on the string "DX6," you will see the updates to the sixth independent for every iteration in the run.

Similarly, the variations in the dependent errors on each iteration contain the letters "DY," followed by the number index of the dependent, followed by the letter "I," followed by the number of the current iteration. For example, the output for the third dependent error in the eighth iteration would contain the string DY3I8. In each case, you must read the setup information for each run to determine the independent or dependent number you are interested in. Considering the large amount of data commonly generated for solver diagnostics, the grepable strings should help you find specific information in the output file.

16.7.3 Option Variables

In addition to the debugLevel, the user has access to the option variable switchUBC, which is a scalar string variable that can take on only two values ("OFF", "ON"). Setting switchUBC to "ON" will activate the Used Before Calculate logic debug mode. For every pass through the solver, another pass will be run with no change to the Independents. On the first “normal” pass the error conditions will be saved. On the next pass with the same Independents, the saved error conditions will be compared with the newly generated error conditions. If there is a difference a UBC error will be generated. The displayed error will include the pass number and name of the error condition that failed.

To turn on switchUBC:

```plaintext
solver.switchUBC = "ON";
```

16.7.4 Interactive Debug Output Functions

In addition to the debugLevel described above, the user has access to several interactive functions which supply specific bits of diagnostic data on demand. These functions can be used from the interactive command line, or may be included in functions to create customized batch diagnostic output.

- **NCPString auxDepStats();**
  Returns data about the status of the auxiliary dependents (inactive target or constraints).

- **NCPString constraintStats();**
  Returns data about the status of the constrained-targets and their associated constraints.

- **NCPString constraintsHit();**
  Returns a list of constraints currently invoked by the solver.

- **NCPString [or string for interpreted function call] convRateInfo();**
  Returns convergence rate statistics for the most recent error update pass (solver iteration).

- **NCPString depDeltas();**
  Returns data showing how much the errors changed during the most recent solver iteration.

- **NCPString depFailStats();**
  Returns data about only those dependents which are currently outside their desired tolerance.

- **NCPString depStats();**
  Returns data about all dependents.

- **NCPString dsvFailStats();**
  Returns data about only those DSVs (DiscreteStateVariables) which have not converged.

- **NCPString dsvStats();**
  Returns data about all DSVs (DiscreteStateVariables).
NCPString indepDeltas();
Returns data showing how the independents were changed during the most recent solver iteration.
The first line contains the value of the scaleFactor applied to the dxUnlim vector and, if this value is less than one, the name and index of the most limiting Independent are also listed, with the indexing starting at one.

NCPString indepStats();
Returns data about the independents.

NCPString setupInfo();
Returns a complete description of the current solver setup.

NCPString stats();
Returns current convergence statistics.

NCPString whyFailed();
Returns a description of way the most recent run failed to converge.

Each of these functions Returns an NCPString, which can be redirected to any desired output stream. For example, to output the current convergence statistics to the screen, type:

```
cout << solver.stats() << endl;
```

### 16.7.5 userReport: User-defined Solver Output Function

A user-definable Solver function, `userReport()`, allows the user access at key points in the internal calculations of the Solver that represents the true end of a Solver iteration. This function is like the `preexecute` and `postexecute` functions for elements and subelements in that implementation is left to the user and, while nominally for Solver diagnostic output, may contain any code the user wishes. This function takes no arguments and returns `void`.

This method is called once during each pass after the objects in the executionSequence list have been evaluated, the dependent errors calculated, constraints checked and any updates to the independent parameters are calculated, but before those updates are applied. The information available is different for each of the pass types the Solver makes during a convergence attempt. The Solver attribute, `passType`, can be used to distinguish the type of pass the Solver is performing when the function is called or to specify that there is no pass.

During a first pass, no independent updates are calculated when the Jacobian is carried over from the previous point. If a Jacobian is being generated immediately, then the independent updates contain the perturbation amount of the first Independent. During an iteration pass, the function is called after the independent parameter updates have been calculated in response to the Dependent error values, but before those updates have been applied to the independent parameters. During a perturbation pass the function is called before the perturbed independent parameter is set back to its baseline value. A special case is the last perturbation pass when generating a new Jacobian matrix. After the matrix is generated a new vector of updates to the independent parameters is calculated with the new matrix and this function is called while the passType is still perturbationPass. The Solver attribute `lastPerturbationPass` is TRUE in this situation. Testing to see if the passType is `iterationPass` OR `perturbationPass` and `lastPerturbationPass` is TRUE will allow all independent update vectors to be captured and output.

The following example implementation will give a results similar to the built-in solver diagnostic output. It demonstrates how to use the individual reporting functions as building blocks. This example sends output to cerr.

To send output to a file, create a output file stream in your model and modify this function to send output to that stream:

```c++
// Define the userReport function

solver {
    // Add flag diagPrintFlag to control level of detail of
```
int diagPrintFlag = 0;

// 0 = no print
// 1 = just convergence summary after point converges
// 2 = adds convergence rate information for each
//    iteration pass
// 3 = adds Independent and Dependent stats for each
//    iteration pass
// 4 = adds Independent and Dependent stats for
//    perturbation passes
// 5 = adds the Jacobian matrix following generation

// jacobianGenFlag and pertPassNum are used internal
to userReport, do not assign a value to them.

int jacobianGenFlag = 1;
int pertPassNum = 0;

// indepDepSummary() - userReport helper function

void indepDepSummary() {
  cerr << indepStats() << depStats() << auxDepStats() << indepDeltas() << endl;
}

void userReport() {
  if(diagPrintFlag != 0) {
    if(passType == "firstPass") {
      if(diagPrintFlag >= 2) {
        cerr << "\n\n*********
CASE = " << CASE << "*********
\n\n=======  FIRST PASS =======
";
      }
      if(diagPrintFlag >= 3) {
        indepDepSummary();
      }
    }
    if(passType == "perturbationPass") {
      if(diagPrintFlag >= 2 && jacobianGenFlag == 1){
        cerr << "\n\n*********
CASE = " << CASE << "*********
\n\n======= GENERATING JACOBIAN #" << numJacobians << " ======
";
        pertPassNum = 0;
        jacobianGenFlag = 0;
      }
      if(diagPrintFlag >= 4) {
        pertPassNum += 1;
        cerr << "\n\n======= PERTURBATION PASS #" << pertPassNum << " ======
";
        indepDepSummary();
      }
    }
    if(passType == "iterationPass") {
      if(diagPrintFlag >= 5 && jacobianGenFlag == 0) {
        cerr << "\n\n*********
CASE = " << CASE << "*********
\n\n======= ITERATION PASS #" << pertPassNum << " ======
";
        indepDepSummary();
      }
    }
  }
}
cerr << "\n\n *** NEW JACOBIAN MATRIX *** \n\n" << "Main Jacobian (active Dependents)\n\n" << J << "\n\n" << "Auxiliary Jacobian " << "(inactive constraints and targets)\n\n" << Jaux << "\n\n" << "\n\n";
}
jacobianGenFlag = 1;

if(diagPrintFlag >= 2) {
cerr << "\n\n=======  ITERATION PASS #" << iterationCounter << " =======" << endl;
}
if(diagPrintFlag >= 3) {indepDepSummary();}
if(diagPrintFlag >= 2) {
cerr << convRateInfo() << endl;
}
if(converged) {
    if(diagPrintFlag >= 1) {
cerr << stats() << endl;
    }
}
}

16.7.6 projectionReport: User-defined Solver Output Function

A user-definable Solver function, projectionReport(), allows the user to create a debug hook inside the error projection loop to help with the error projection analysis.

The definition syntax for the function is shown below:

```c
solver {
    void projectionReport() {
        // function implementation
    }
}
```

16.7.7 dxLimitAll: User-defined Solver Output Function

A user-definable Solver function, dxLimitAll(), allows the user to add iteration dependent dxLimited bounding. This function is like the preexecute and postexecute functions for elements and subelements in that implementation is left to the user. The developer is responsible for all variable declarations.

Here are some examples of what could be in dxLimitAll():

```
1) Early damping: CVARGA: default=1, 0<=range<=1,
   typically might be 0.0 for poorly guess models:
   IITER = iterationCounter;
```
The definition syntax for the function is shown below:

```c
solver {
    void dxLimitAll() {
        // function implementation
    }
}
```

### 16.8 Solver Diagnostic and Trouble-Shooting Reference

The Solver Subsystem may post a number of different warnings and exceptions. The messages that are output can be used to help diagnose and fix the problems with the Solver setup. This table describes several of the most common problems related to Solver operation and suggests remedial actions that you may take to correct the problems. This list is by no means exhaustive, however. Some problems can only be diagnosed and corrected based on experience with numerical solvers, their design and operation.

<table>
<thead>
<tr>
<th>Type of Error</th>
<th>When Error Occurs</th>
<th>Remedial Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>User sets an illegal attribute</td>
<td>Solver setup</td>
<td>List solver variables or consult design documentation to identify valid attribute names</td>
</tr>
<tr>
<td>User sets an illegal option value</td>
<td>Solver setup</td>
<td>List the valid options or consult design documentation to identify and set valid option values.</td>
</tr>
<tr>
<td>Attempt to add Independent, Dependent or Integrator which already exists in the Solver</td>
<td>addIndependent addDependent</td>
<td>List the independents and dependents currently defined by the model and existing in the solver. Identify independents which access the same model variable and dependents which access the same pair of model variables. Avoid adding redundant objects to solver.</td>
</tr>
<tr>
<td>Solver refers to elements or variables not in the model</td>
<td>verify</td>
<td>Confirm that named elements and variables exist in the model and modify definition of solver objects accordingly.</td>
</tr>
<tr>
<td>Unequal numbers of independents and dependents in Solver</td>
<td>verify</td>
<td>List the independents and dependents currently added to the solver. Identify missing or superfluous solver terms. Add or remove objects as necessary.</td>
</tr>
<tr>
<td>No mapping expressions defined</td>
<td>verify</td>
<td>If mapping is supposed to be enabled for a given solver term, set a mapping expression. If mapping is not intended, make sure the mapping expressions are set to &quot;&quot; (blank).</td>
</tr>
<tr>
<td>Row Singularity</td>
<td>run-time</td>
<td>Review the model to ensure that the dependent variables referenced in the affected dependent object are properly connected to the model and that the two variables are not trivially related to each other. Look for any limiting logic that may cause the variables to stop at some limit during Jacobian generation.</td>
</tr>
<tr>
<td>Column Singularity</td>
<td>run-time</td>
<td>Review the model to ensure that the independent variable referenced by the solver term is correctly connected to the rest of the model. Verify that nothing in the model is setting the value of the independent variable (other than the solver). Can indicate</td>
</tr>
<tr>
<td>Type of Error</td>
<td>When Error Occurs</td>
<td>Remedial Action</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>that iteration has reached the edge of a table or map with extrapolation turned off so that the same table/map value is returned regardless of the map independent value set.</td>
</tr>
<tr>
<td>Row Linear-Dependent matrix</td>
<td>run-time</td>
<td>Substitute or define a different dependent object for one identified as the source of the problem. Make sure the error term for the new dependent is not a linear function of any other in the solver. If no such dependent object can be defined, consider eliminating one of the dependents AND one of the added independent objects from the Solver.</td>
</tr>
<tr>
<td>Column Linear-Dependent matrix</td>
<td>run-time</td>
<td>Substitute or define a different independent object for one of the those identified as the sources of the problem. Make sure that the new independent variable is not a linear function of any other in the solver and that it does not affect every error term in the solver in the same way as an existing independent. If no such independent object can be defined, consider eliminating one of the independents AND one of the dependent objects from the Solver.</td>
</tr>
</tbody>
</table>
| Solver fails to converge            | run-time          | * Easiest solution is to modify the initial guesses for the independent variables, or set the maxIterations attribute to a larger value.  
* If the convergence failed because it exceeds the maximum allowed number of Jacobian matrices, increase maxJacobians and/or maxBroydens attributes.  
* List errorConverged and updateConverged attributes to determine which terms failed to converge. Verify that none of these terms have excessively tight convergence tolerances.  
* Verify that none of the independent objects have excessively large values for dxLimit attributes.  
* If the user has set any attributes to non-default values, set these back to the default values. |

If the diagnostic and corrective actions described above do not solve the problems encountered in execution of the model, it may be necessary to examine the solver and model data for each iteration of the model in detail. A diagnostic output report format has been designed for the Solver. This report should be included in the run input file and the results analyzed by someone experienced with both the model and the solver algorithms. In most cases, the problem will eventually be traced to limitations in the model itself, rather than the Solver. In other cases, the problems may be due to inappropriate solver option or control parameter settings.

### 16.9 Solver Terms

The following terms or phrases are used in conjunction with the Solver.

- **absolute**  
  When referring to Solver variables, this term refers to a quantity that is not to be measured relative to some base value. When the toleranceType for a given Solver Dependent is ABSOLUTE, for example, the tolerance is the actual largest allowed difference between the two sides of the equality, and not the difference relative to the size of the right-hand side value. *(See fractional.)*

- **algebraic loop**  
  Refers to an instance where the input to a given model element is used before it is calculated by a second model element. For the model solution to be valid, all algebraic loops must be resolved by the Solver.
bias

Value added to a Solver Independent value in order to above floating point divide by zero exceptions and general convergence failures. A bias value should be assigned for any independent parameter which is expected to approach zero or change sign during a given run.

constraint

A limit placed on a model parameter; the solution to the model should not permit any Constraint from being violated.

convergence

Describes the state (or process of attaining the state) where all equations in the model are simultaneously satisfied (within some specified tolerance).

Dependent

Contains references to the right and left hand sides of an equation which must be satisfied in the model (energy balance, for example). The function of the Solver is to vary the model independent variables until all such equations are satisfied.

DiscreteStateVariable

Defines discontinuous operating regions for the model, or some portion of the model. The model should contain logic to set the DiscreteStateVariable value based on conditions in the system, and logic to use the DiscreteStateVariable value to control branching between different modes (behaviors). Using DiscreteStateVariables allows the solver to control switching between modes when the model gets caught in a loop between two model states.

DSV

See DiscreteStateVariable.

dxLimit

The maximum allowed change in each Solver Independent on a single convergence iteration (attempt).

error

The difference between the right and left hand sides of a Dependent equation. The Solver's function is to drive all errors to zero (within some tolerance).

fractional

When referring to Solver variables, this term refers to a quantity taken as a fraction of another. A fractional tolerance, for example, is the maximum allowed error, divided by some reference value (typically the right-hand-side value of a Dependent equation).

Independent

A model parameter which can be varied by the Solver to achieve a converged solution.

Integrator

Calculation or method which gives the area under a function curve over a specified period of time. The function in question is typically the time-derivative of a model state (solver independent) variable. Various methods exist for making this calculation; most assume a linear approximation to the function curve over a relatively small interval of time.

iteration

A single convergence attempt by the Solver. Each iteration involves the Solver setting the Independent values, running the model, and computing the new error values.

Jacobian

The matrix containing the partial derivatives of all Dependent errors with respect to changes in the Independents.

locked attribute

When the user sets an attribute in a specific Independent or Dependent, and if that attribute also has a default value set in the Solver, the local attribute value is locked to prevent further changes in the Solver default value from changing the value specified locally by the user. In order for the default value to be used in the future, that specific Independent or Dependent attribute must be unlocked by the user.

mapping

A non-linear relationship between the model independent value and the Solver
**perturbation**  
An incremental change to an *Independent* parameter value, used here to compute the *Jacobian* matrix.

**Solver**  
The object which controls the convergence process at each *steady-state* point or *transient* time-step. The Solver varies the *Independent* values to drive all *Dependent* errors to zero.

**steady-state**  
A converged state of the model in which all time-derivatives are zero (all flows and forces balance). The NPSS can find this condition **without** having to run a *transient* simulation until it settles out.

**tolerance**  
The maximum allowed value of each error value in order for the model to be considered converged.

**transient**  
A time-varying solution to the model. A transient run may consist of a large number of single-point model solutions, each representing the state of the system at a different value of time.
17 Transient Reference Guide

17.1 Transient (TransientExecutive class)

Time-dependent or transient simulation in the NPSS is performed by the TransientExecutive class. Only one TransientExecutive object with a name of “transient” is currently allowed in the simulation. It is automatically created at program start-up in the top-level assembly. The user cannot manually create a TransientExecutive in subassemblies.

The TransientExecutive has several user attributes and functions, as described briefly below.

A TransientExecutive is automatically created in the top-level Assembly. The minimum required user instructions in order to run a transient are as follows.

```plaintext
setOption("solutionMode", "TRANSIENT");

// set the stop time for simulation (in seconds)
transient.stopTime = desired_stop_time;

run(); // run the transient
```

A specific solution process will be followed to simulate the time variation of an NPSS model. The solution process makes assumptions about the state of the model; these are stated in Section 17.1.1 below. The initialization of the transient computation, which occurs only at the beginning of the transient, is described in Section 17.1.2. The time step computations (described in Section 17.1.3) handle the computation from the start time until the user’s end criteria are met.

17.1.1 Initial Model State Assumptions

Certain assumptions are made about the model state at the start of the transient solution process. It is intended, although not required, that the user will have run either a steady-state point or transient point prior to the commencement of a transient solution. The specific assumptions about the model state at the start of the transient process are:

1. The model is assumed to contain initial values for the integration states and iteration independents. The user typically sets these values by running a previously converged point.

2. If the solver setup has changed, a partial derivative matrix will be created by the solver.

3. The start time may be some value other than zero

17.1.2 Initialization of the Time Update Loop

The time history vectors and discrete elements of the system must be initialized. This initialization occurs at the start of the transient only. The initialization steps, in execution order, are described here.

1. Set the initial value of time and time increment, $dt_{base}$. The time value may be explicitly input by the user, or it may be inherited from a previously stopped transient simulation.

2. Compute time dependent inputs as functions of time.

3. Initialize the discrete elements of the system. The discrete elements of the system are allowed to carry out their specific initialization computations. The sensor histories of the NPSS discrete elements are initialized at this point.
4. Initialize the state integration histories

5. Initialize the solver independent histories. The solver independent histories are used to project the independent values for the next time step based on past history.

6. Set the convergence tolerance on all solver dependents to be larger (looser) by a multiplicative factor \((toleranceScaleFactor)\), which can be set by the user.

### 17.1.3 Time Update Loop

The time update loop is responsible for marching through time until the user’s finish criteria are achieved. Once the finish criteria are achieved, control returns to the calling object. A minimum of two passes through the model are required on each time step when adaptive time-stepping is ON. The first, the predictor pass, is used to modify the timeStep, if necessary, and the second is required to close the corrector errors. Time-discrete elements do not execute on predictor passes but should run on the first corrector pass, thus requiring at least two passes through the model. When adaptive time-stepping is not being used (user input timeStep), the predictor pass through the Model is not required, and it is possible that the time-step may succeed in a single pass.

The time-update process is described below and is summarized in Figure 6.

#### 17.1.3.1 Compute a Time Increment

The choice of time-increment \((dt)\) is made prior to the first convergence iteration (Solver) pass through the Model. This is done on each time step in the simulation.

1. Determine a required or forced time increment, \(dt_{\text{force}}\), by querying the time-discrete objects in the model for their next required times to be called (see Section 0 for a discussion of time-discrete objects)

2. Apply limits from the forced time increment, the \(stopTime\) for the simulation and the \(minTimeStep\):

   \[
   dt = \max\{dt_{\text{min}}, \min\{dt_{\text{base}}, dt_{\text{force}} \cdot (t_{\text{finish}} - t_{\text{current}})\}\}
   \]

   where \(dt_{\text{base}}\) is either input or computed by the adaptive time step logic on the previous time-step (see steps 10-11 below)

3. Update the current solution time:

   \[
   time = time + dt
   \]

4. Predict the iteration independents at the computed time, based on their values at previous time-points, and limited by \(dxTranLimit\) (see Section 17.1.4 for a description of predictor methods).

5. Compute time-dependent inputs as functions of time.

6. Steps 7-11 below describe the default adaptive time-stepping algorithm provided with the NPSS software. If a user-defined time-stepping function is to be used instead, execute the function and then proceed to Section 17.1.3.2 below. If no time-stepping function is to be used, proceed directly to Section 17.1.3.2 below.

7. Make a single predictor pass through the preSolve and Solve sequences.

8. Compute solver error terms including state integration errors

9. Compute projected solver independent values. The Model error terms, independents and solver derivative matrix are used to compute the independents needed to achieve convergence at the current
time. These independent values should be unlimited by the independent limiting logic since they will be used to evaluate the model state movement for the time step.

10. Compute a new time-increment to limit the change in model independents, as described in Section 17.1.5.1. This computed time increment may be larger or smaller than the current time increment. The computed time increment is assigned to $dt_{base}$ so it will be used on the next time step. (Note that it would be useful to have the limited time increment computed, even when fixed time steps are activated by the user.)

11. If another predictor pass is required with the time increment computed in Step 10, return to Step 2 above. Logic described in Section 17.1.5.1 is used to determine whether another pass is required. Otherwise continue to the next section.

17.1.3.2 Before the Convergence Iteration
The actual time step is known (either input directly or computed with the adaptive time step logic). The next part of the solution process focuses on calling the appropriate discrete elements to execute. At the end of these operations, the discretes will have “caught up” with the simulation time. Please refer to Section 0 for more details on the discrete objects of the system.

1. Call discrete Elements of the system to run to the value of time. The discretes will execute as many times as required to catch up with, but not exceed, the continuous model simulation time.

2. For each time the discrete needs to run, NPSS discrete elements will extrapolate from saved sensor histories and set the discrete’s “sensed” input to the extrapolated value.

3. Perform any explicit integrations assigned to the Transient Executive (not internal to the Model).

17.1.3.3 Convergence of the Model
The discrete objects of the system have run and provided their inputs to the continuously varying portion of the Model. The continuous portion of the model must be executed to convergence.

Run the model to convergence, closing on solver iteration errors and implicit state integrations (see Section 17.2.2).

17.1.3.4 Post convergence computations
1. Upon convergence (which might have required only a single model pass), an additional pass may be made through the inverse-Jacobian matrix to update the independents. This additional pass, although making only a small change in the independents, makes the predictor pass on the next time-step more accurate. This step improves the accuracy of the transient simulation, especially when the $toleranceScaleFactor$ has been set to a value greater than 1.0 (looser tolerances). This additional independent update is performed only when the $extraIndepUpdate$ option is set to TRUE.

2. Perform explicit integration of any state in the top-level Assembly. This refines the explicit integrations performed in Step 3 of Section 17.1.3.2 above. This can only be performed on state integrators in the top-level because integrators in lower-level assemblies may have already had their histories updated.

3. Perform an update pass through the model hierarchy. The update pass allows elements that keep time-histories to update them using converged (rather than predicted) inputs.

4. Call the postSolve execution sequence

5. Make updates to any “active” reports

6. Update state integration and iteration independent histories
7. Test for run termination conditions. These conditions may be user supplied.

8. Loop back to process the next \( dt \) time increment described in Step 1 of Section 17.1.3.1.

It should be noted that there is no update pass through the model after convergence at a given time step. Model elements with internal integration must use the value of time to determine when their internal updates should be performed. There will be no support in the Solver or Transient Executive for integrations performed internal to the model elements.

---

**Figure 2. Time March Executive Event Sequence**

---
Further explanation of steps in Figure 2:

Step 1: Determine a Computational Frame Time (TF) increment by querying all the objects in the system for their next required time to be called. The closest time requirement will set the value of TF. This will ensure that the computational time doesn’t pass any object’s required time. Note that the discretes will be queried at this time, and if they are operating in a synchronized mode of operation, they will force TF to match up with their execution frequency.

Step 2: Compute a provisional time, T, as the minimum of the last time plus the previous “dt” and TF. Predict the iteration independent values at proposed new time, TF. A prediction example is shown below of the independent value at T(n+1).

![Figure 3. Time Sequence - Step 2](image)

Step 3: Compute the time dependent user inputs at T(n+1). Call continuous elements at T(n+1). The single pass call for the continuous elements produces solver error terms at the T(n+1). The iteration independent update diagram might now look like the following:

![Figure 4. Time Sequence - Step 3](image)

Step 4: From the first pass computed error terms, updated values of the solver independents can be computed by multiplying the error terms by the solver partial derivative matrix. If one or more of the independents move more than the specified bandwidth limit (expressed as a percentage and absolute movement value), then the time increment is pulled back to the most limiting time as indicated by intercepts of the independent movement with the allowed bandwidth. This situation is shown below. Note that the diagram depicts a contraction in the time step in order to maintain accuracy, it is also possible to extend the time step (limited by TF of course) according to the same bandwidth argument. Since the time step is now known with certainty, TIME and time increment “dt” can now both be set.
Step 5: The Discrete Elements are all called after the time step has been computed from step 4. The discretes will execute as many times as required to catch up with, but not exceed simulation time, Time(n+1).

When a Discrete Executes, it first computes the sensor inputs based on the solution history of the continuous Elements. Three possible techniques exist for computing the sensor inputs to the discrete:

1) Use Time(n) values.
2) Use sensor history to extrapolate the sensor values to Time(n+1)
3) Interpolate between the Time(n) point and the single pass, Time(n+1) point that was made to compute first pass errors.

Compute time dependent user inputs at Time(n+1).
The Discretes return their outputs as functions of time so that the continuous elements will have them available.

Step 6: Update iteration independent values for next Solver pass.

Step 7: Inputs versus time are evaluated. The continuous Elements are executed. The state derivatives are passed to the integrator.

Step 8: The Integrator is called for each state. The integrator will return an error term for each integrand. The value of the error term will be one of two possible error terms:

1) Integrand demand error, \( (X(i) - X(i-1))/X(i) \).
2) Derivative error, \( \{(DXQDT(i) - \{(X(n+1)(i-1) - X(n) )/DT \}} \)

where “i” is referring to iteration pass, and “n” is referring to time steps.

An “equivalent time constant” is used as a means of identifying the relative response of a given state variable. It can be estimated on the basis of model parameters or can be provided as part of the integration specification.

Based on the relative values of the time constant and the DT, the appropriate mode is selected - large DT: differential - small DT: integral.

Note that in steady-state (DT -> very large), the second term of the derivative error is equal to zero.

Step 9: The Solver is called to determine whether convergence has been achieved. If not achieved, the independents are updated via step #6 and another pass through the continuous Elements is made. This step in addition to solving the continuity errors, will provide a correction pass for the integrator.

Step 10: After convergence, all history arrays are updated. Histories might be saved for a variety of purposes. The value of TIME is saved into TIME_OLD.
Step 11: Call Report Writer. The report writer writes data to reports if it is the correct time to do so. The report frequency may be controlled by the user in these ways:
- by time frequency (every 0.1 seconds)
- by number of time steps, (i.e. every 5 time steps, write output)
- by discrete time specification (at 1.1 seconds, 1.7 seconds, etc...)

Step 12: Determine if simulation time has attained the demanded frame time, TF. If so, continue, else loop back and compute a new proposed time step, “dt”.

Step 13: Determine if simulation time has attained the demanded final time of the simulation.

Further explanation of the terms used in Figure 5:

- **Discrete Element** – A Discrete Element is a special type of Element that performs integrations w.r.t. time, internally in the Element Logic. These element types may Only be Called once, at a given time. (has equations of the form: \( X = X + DX/DT \times DT \))
- **Synced Discrete** – A Synced Discrete refers to a Discrete for which the continuous submodel is forced to compute time accurate inputs. Predictions of actuation and sensors, at times when they haven’t been computed, are provided by default.
• **UnSynced Discrete** – An UnSynced Discrete refers to a Discrete Element for which the continuous submodel is NOT forced to compute time accurate inputs. UnSynced Discretes use out-of-date inputs (this speeds up execution). The maximum latency of Discrete sensor input data is controlled by the Maximum Continuous DT input.

• **Continuous Element** – A Continuous Element is one that assumes all Integrands with respect to time, X, are passed into the Element as input. These Elements generate derivatives of the integrand w.r.t. time (has equations of the form: DX/DT = F(X)). For Steady-state, the external Solver will drive X until DX/DT = 0. For Transient, the external integrator will force X to be the integrand of DX/DT, integrated with respect to time.

• **Maximum Continuous DT** – A maximum allowed time step for the continuous elements forces a time step to occur, even if the time management routines don’t think one is needed.

• **Minimum Continuous DT** – The Solver will error out if the time step reduces below the minimum time step without a solution.

### 17.1.4 Independent variable predictions

An important part of the transient solution process is the prediction of independent values of the model for future values of time based on known past values. Three predictor methods are provided with the NPSS: NONE, LINEAR, and LAGRANGE. The choice of method is selected by the user by setting the `predictorMethod` attribute. Each method is described in greater detail below. In each of these expressions, the subscripts represent the time-step, and the \( n \)th time step is the one for which predictions are being generated.

#### 17.1.4.1 No Predictor Extrapolation

When the `predictorMethod` is set to NONE, the predictor value is set to the independent value on the previous time-step.

\[
x_n = x_{n-1}
\]

#### 17.1.4.2 Linear Predictor Extrapolation

When the `predictorMethod` is set to LINEAR, the predictor value for each independent is set according to the expression

\[
x_n = x_{n-1} + \frac{x_{n-1} - x_{n-2}}{t_{n-1} - t_{n-2}} \cdot (t_n - t_{n-1})
\]

#### 17.1.4.3 Lagrange Predictor Extrapolation

When the `predictorMethod` is set to LAGRANGE, the independent predictions will be expressed in terms of a three point Lagrange polynomial technique. This is a standard technique for making extrapolations with arbitrarily spaced data observations. By saving a three point history of times and independent value observations within the predictor, good predictions can be made with the Lagrange formula.

The three point Lagrange extrapolation formula for a given independent, \( x \), is:

\[
x_n = x_{n-3} \cdot P_{n-3} + x_{n-2} \cdot P_{n-2} + x_{n-1} \cdot P_{n-1}
\]

Where \( x_n \) indicates independent’s value at the \( n \)th time step, and \( P \) refers to coefficients that are computed from the time history.

The formula for the three Lagrange coefficients: \( P_{n-3} , P_{n-2} , P_{n-1} \) are expressed in terms of the times at which the three \( x \) observations were made as well as the time at which \( x_n \) is desired.
\[ P_{n-3} = \frac{\left( t_n - t_{n-2} \right) \left( t_n - t_{n-1} \right)}{\left( t_{n-3} - t_{n-2} \right) \left( t_{n-3} - t_{n-1} \right)} \]  
\[ P_{n-2} = \frac{\left( t_n - t_{n-3} \right) \left( t_{n-1} - t_{n-1} \right)}{\left( t_{n-2} - t_{n-3} \right) \left( t_{n-2} - t_{n-1} \right)} \]  
\[ P_{n-1} = \frac{\left( t_n - t_{n-3} \right) \left( t_{n-1} - t_{n-2} \right)}{\left( t_{n-1} - t_{n-3} \right) \left( t_{n-1} - t_{n-2} \right)} \]  

(4a)  
(4b)  
(4c)

It is useful to note that for equally spaced time values (i.e. constant time increment), the three Lagrange polynomial values will be 1, -3 and 3 respectively.

**17.1.4.4 Limits on Predicted Values**

Setting the Independents to the predicted values and running the model entail some risk since no constraint checking is done on the predicted values before the model is executed. It is therefore possible that the predicted values are outside the range of valid values for the model, such as negative areas or pressures. The simulation may not be able to recover if it is run with the predicted values. To protect against this, attributes are available in the Independent objects that allow a minimum and maximum range to be defined. (See 16.2.2). If a predicted value is passed to the Independent, the Independent will limit the predicted value to the limit value.

**17.1.5 Time Step Computation**

There are three basic reasons for adaptive time stepping in a transient simulation:

1. Synchronize model execution with discrete time requirements.
2. Improve model transient accuracy.
3. Improves the overall computational performance of the simulation by allowing the time increment to increase when possible.

All three factors must be considered when selecting the next simulation time for the model. A time line is shown in Figure 5 illustrating the various factors in computing the time increment. The process of adaptive time-stepping and model execution is described in detail is section 17.1.3.1 above, and is also depicted in Figure 5.

**17.1.5.1 Adaptive Time Stepping.**

The following expression can be used to compute a new time increment, based on an allowed change in the solver independent values, as set by the user. This expression may increase (expand) or decrease (reduce) the time increment.

\[ dt_{\text{adapted}} = \frac{dt_{\text{current}}}{\max_i \left( \frac{dx_i_{\text{predicted}}}{dx_{\text{TranLimit}}_i} \right)} \]  

(5)

The following process is used to determine how \( dt_{\text{adapted}} \) is used.
1. If $dt_{\text{adapted}}$ is greater than current $dt_{\text{current}}$ (i.e. if the time increment is being expanded), it may not be necessary to make another predictor pass to test the expanded time increment. The value of $dt_{\text{current}}$ may be used on is step instead, if the following condition is satisfied.

$$dt_{\text{current}} \geq dt_{\text{ExpandFactor}} * dt_{\text{adapted}}$$  \hspace{1cm} (6)

The user can control this decision by setting the value of $dt_{\text{ExpandTolerance}}$. In this case, the $dt_{\text{adapted}}$ will be used to set the $dt_{\text{base}}$ in the next time-step.

$$dt_{\text{base}} = dt_{\text{adapted}} * dt_{\text{RelaxFactor}}$$  \hspace{1cm} (7)

The $dt_{\text{RelaxFactor}}$ is intended to help ensure that the $dt$ used in the next predictor pass will be small enough to avoid another time-step reduction.

2. If $dt_{\text{adapted}}$ is greater than current $dt_{\text{current}}$ (i.e. if the time increment is being expanded), but the condition in Equation 6 is not satisfied, $dt_{\text{current}}$ may still be used for the current time-step if $dt_{\text{adapted}}$ exceeds $dt_{\text{force}}$ or $dt_{\text{max}}$. In that case, the $dt_{\text{base}}$ for the next time-step will be set according to the appropriate limit.

If $dt_{\text{adapted}} > dt_{\text{force}}$, then

$$dt_{\text{base}} = dt_{\text{force}} - dt_{\text{current}}$$  \hspace{1cm} (8)

If $dt_{\text{adapted}} < dt_{\text{force}}$ but $dt_{\text{adapted}} > dt_{\text{max}}$, then

$$dt_{\text{base}} = dt_{\text{max}}$$  \hspace{1cm} (9)

The user may specify the value of $dt_{\text{max}}$ ($\text{maxTimeStep}$)

If $dt_{\text{adapted}} < dt_{\text{force}}$ and $dt_{\text{adapted}} < dt_{\text{max}}$, then another predictor iteration is required and control is passed to step 5 below.

3. If $dt_{\text{adapted}}$ is less than $dt_{\text{current}}$ (i.e. the time increment is being reduced), then $dt_{\text{min}}$ is used for the current time-step if the following condition is satisfied:

$$dt_{\text{current}} \leq \frac{dt_{\text{min}}}{dt_{\text{RelaxFactor}}}$$  \hspace{1cm} (10)

The value of $dt_{\text{base}}$ for the next time-step is also set equal to $dt_{\text{min}}$. The user may set a value for the $dt_{\text{min}}$ ($\text{minTimeStep}$).

4. If $dt_{\text{adapted}}$ is less than $dt_{\text{current}}$ (i.e. the time increment is being reduced), and Equation 10 is not satisfied, then another predictor iteration is required and control is passed to step 5 below.

5. The values of $dt$ (see section 17.1.3.1) and $dt_{\text{base}}$ are set according to Equation 11 and the process repeats at step 2 in section 17.1.3.1 above.

$$dt = dt_{\text{RelaxFactor}} * dt_{\text{adaptive}}$$

$$dt_{\text{base}} = dt_{\text{RelaxFactor}} * dt_{\text{adaptive}}$$  \hspace{1cm} (11)
17.1.5.2 Handling Synchronized Time-Discrete

An NPSS Model may include objects that must be executed at specific times. These are referred to here as \textit{synched time-discrete objects}. An Elements which contains digital control logic is one example. During the initialization of the Transient Executive, the synched time-discrete objects in the Model are identified. At each time-step, the Transient Executive must query the synched time-discrete objects to determine the next simulation time required (the earliest of all times indicated by the discrete elements). This defines the value of $dt_{force}$ (as used in section 17.1.3.1 above). If the simulation is about to step past a synched time-discrete request, the $dt$ is reduced so that the model will execute at the proper time.

17.1.5.3 Handling Unsynched Discrete Elements

\textit{Unsynched time-discrete objects} are those which perform time-dependent operations (such as integration) internally. These model objects use input values from the predictor pass of the top-level assembly to compute their own internal behaviors over the entire top-level time-step.

Before the unsynched discrete elements can be run, the proper time step for the model must first be determined as described in section 17.1.3.1 above. The time-discrete objects will not be execute during the predictor passes required to determine the proper time step. In the process of determining the time-step, the time-continuous model elements are executed and their output can be used to predict input values to the time-discrete objects. If an unsynched time-discrete object has a time increment smaller than that of the top-level assembly, it may execute several points for each top-level time-step. All required internal time-steps are executed on the first iteration of the corrector convergence process (for the top-level Assembly). These objects are executed only on this pass, and should be placed in the preExecutionSequence of the Model. It is desirable to have control elements run first, for example, so that their output values will be available to the system for the current time-step.

Since the discrete control elements generally require sensor inputs from the model, predictions of these parameter values are required at each discrete time where the elements must run. These sensor values may be predicted in three ways:

1. Use sensor values from the previous converged time-step (at $t_{n-1}$) for all internal sub-steps.
2. Extrapolate sensor values from a time-history of the last k converged solutions ($t_{n-k}$ through $t_{n-1}$).
3. Interpolate sensor values between the previous converged time-step solution (at $t_{n-1}$) and the predictor pass values (at $t_n$).

It is important to point out here, however, that the choice of sensor-prediction methods and the storage of time-histories for option 2 is the responsibility of the individual time-discrete elements. This function is not performed by the Solver or Transient Executive. The iteration over multiple discrete element cycles is done within the discrete elements themselves and do not require separate calls from the system or solver software.

17.1.6 Utility Functions

Certain optional utility functions have been identified that are not required in the Transient Executive source code, but which might be valuable to the users, for inclusion in their user-defined functions. Utility functions are built-in; they are compiled C++ rather than being interpreted by the NPSS. The functions are built in because they perform operations (such as sorting through arrays of pointers to other objects) which do not lend themselves to interpreted code.

The Transient Executive employs user-defined function for conditional report updates, conditional run termination, and adaptive time-stepping. The following utility functions have been identified as potentially useful.
17.1.6.1 Transient Run Quiescence Utility Function

This utility determines whether a transient run has come to a quiescent (steady) state. It does this by comparing the value of the expression

$$\left(\frac{dx}{dt} \ast \tau\right)_n x_n$$

with the user-defined threshold attribute (quiescenceTolerance). If this quantity is below the set threshold for all states on two consecutive time-steps, the quiescence conditions is considered TRUE.

17.1.7 TransientExecutive Functions

The following are Transient Executive functions:

```cpp
int quiescence ();
Returns TRUE (1) if the state integrators satisfy the quiescence criteria (see Section 18.1.6).

int testRunTermination ();
Returns TRUE if the transient terminateCondition is satisfied, returns FALSE otherwise.

real computeTimeStep ();
Computes the correct time increment for the next time-step, using whatever method the user has specified in transient timeStepMethod attribute.

void clear ();
Resets all solver dependent error tolerances to their steady-state solution values, puts all explicitly integrated states back in the solver, and clears the lists of pointers which the TransientExecutive keeps for the model.

void doExplicitInteg ();
Updates states associated with all explicit Integrators, based on the current time-step value.

void initializeHistory ();
Sets all independent predictor histories to the current independent values. Initializes histories for all state Integrators. As an alternative the attribute forceInitializeHistory has been added to the transient solver. Setting this flag to TRUE will cause histories to be initialized at the top of the first time step of the next transient run. As such it represents a replacement to calling this function between two transient runs to force initialization so that the history vectors don’t carry over.

void predictorPass(real predictTime);
Performs a predictor pass on all Independents in the model. This updates the currentValue in the associated histories kept by the TransientExecutive object.

void run ();
Runs a transient simulation. This function is not usually required, since the normal system run command typically invokes this function anyway.

int runTimeDiscretes(real prevTime, real nextTime);
Calls the runDiscreteCalcs function on the parent assembly, which sends that message to all objects contained in that assembly in turn. The previous and next (predicted) values of time are passed into this function through the argument list. This function should return TRUE if inputs to the model have been changed, FALSE otherwise.
```
Normally this function is called before the predictor pass on each time-step in a transient simulation. Predictor passes will, however, be run before this function is called if the adaptive time-stepping option is enabled.

```cpp
void setup();
```

Obtain all required configuration information from the Solver and set up the Transient internal data structures accordingly.

```cpp
void synchTimeDiscretes();
```

Query all Model objects for their next required execution time. Only time-discrete objects will return limiting values for the next time value. This function sets the dtForce attribute in the TransientExecutive.

All of the above messages must be sent to the TransientExecutive object, which has the name "transient" by default for any given assembly.

```cpp
nextTime = currentTime + transient.computeTimeStep();
transient.predictorPass(nextTime);
// etc.
```

### 17.1.7.1 Example

The following function illustrates how the above functions might be used to create an alternative transient simulation process.

```cpp
// the initializeHistory message propagates throughout model.
initializeHistory();

int terminate = FALSE;

// begin loop over time.
do {

    // increment time, using transient's time-step algorithm
    time = time + transient.computeTimeStep();

    // must run pre-solver sequence specifically (using function
    // defined previously) since solver does not run that sequence
    // automatically in TRANSIENT mode.
    runPreSolverPass();

    // run a single converged point for the current time.
    solver.run();

    // update the transient histories (propagates throughout model).
    // includes updates to DataViewers.
    updateHistory();
    terminate = transient.quiescence();
}
while ( (time < transient.stopTime) && (!terminate) );
```

### 17.1.8 Transient User Input Attributes

**baseTimeStep = 0.05;**

The default timeStep selected by the user (the value set by the user may be altered by the Transient Executive as required during the simulation).

**dtExpandTolerance = 0.30;**

The minimum fraction that the current dt may have, relative to the adapted dt (the new dtbase), in order to be used on the current time-step.
dtRelaxFactor = 0.90;
The fraction of the adapted dt to be used in the next adaptive iteration or for the next step's dtbase. Used to ensure that the next time step reduction will be sufficiently small, allowing for non-linearities.

dxTransLimit = 0.10;
Maximum allowed change in each Solver independent over a given time increment (for adaptive time stepping). This attribute is different from the Solver dxLimit, which is the maximum allowed independent change for any given Solver convergence iteration.

dxTransLimitType = "FRACTIONAL";
Option which determines whether dxTransLimit is taken as an absolute or fractional value.

explicitIntegOrder = "BEFORE";
This attribute still exists, but is no longer has any impact on the simulation.

extraIndepUpdate = FALSE;
When TRUE, an additional update will be made to the independents (though not set in the model). This makes the predictor for the next pass more accurate, especially when the toleranceScaleFactor is greater than 1.0.

forceInitializeHistory = FALSE;
Set by the User between two back-to-back transient runs when it is not desireable to carry over history from one transient run to the next. Setting this attribute represents an alternative to calling the initializeHistory() function between runs. Calling the initializeHistory function is still valid. After initializing the history vectors at the top of the first time step, the transient solver sets this flag back to FALSE. Thus it will have to be set to TRUE before each transient run where the history vectors are to be initialized. Note that this flag does not affect the required initializations that are automatically performed any time the solutionMode changes. Any time the solutionMode changes, all histories are initialized before starting a transient run.

frameRate = 50;
The maximum number of millisecs allowed for iterations that can be run on a transient point. If the max limit is hit, the solver will assume convergence and move on to the next point.

initialized = 0;
Integer flag that indicates whether or not the TransientExecutive has been initialized. If FALSE (0), the TransientExecutive and the Model will be re-initialized and the value will be set to TRUE(1). You can use the clear() function to reset this attribute to zero.

integErrorForm = "INTEGRAL";
The form of the integration error to be used by the Solver. Valid options are INTEGRAL, DIFFERENTIAL and VARIABLE.

integrationType = "GEAR_1st_ORDER";
The default integration method for all states in the system. The user may override this default by setting the integrationType attribute on each NCPStateIntegrator object individually. Valid options are GEAR_1st_ORDER, GEAR_2PNDP_ORDER, TRAPEZOIDAL, and EULER.

maxIterations = 5;
The maximum number of iterations that can be run on a transient point. If the max limit is hit, the solver will assume convergence and move on to the next point.

maxTimeStep = 1.000;
The maximum time step allowed in adaptive time stepping.

minTimeStep = 1.00e-04;
The minimum time step allowed in adaptive time-stepping.
predictorMethod = "LINEAR";
The method used to estimate the independent values on the predictor pass of each time-step. Options are NONE,
LINEAR, and three-point LAGRANGE.

quiescenceTolerance = 0.01;
The maximum allowed value of (dx/dt * time-constant) (expressed as a fraction of the state value) for each state to
be considered quiescent (steady with respect to time).

stopTime = 10.00;
The default finish time for a transient simulation (this value may not be reached if user-defined run-termination
conditions are met during the simulation). This value MUST be entered, or an exception will be thrown. The
default is zero (no run).

terminateCondition = "quiescence()";
User-defined integer expression used to determine whether the current transient run should terminate. If this
expression returns TRUE, the run will terminate. The default is blank (no conditional termination, runs to
stopTime).

time = 0.0000;
The current time for a transient simulation (the input value is used as the start time and is updated by the Transient
Executive during the simulation).

timeStepMethod = "CONSTANT_DT";
Desired method of computing the time-increment on each point of the transient run. Valid options are
CONSTANT_DT, ADAPTIVE, and USER_DEFINED. Users should note that setting this attribute to
ADAPTIVE will cause predictor passes to be performed BEFORE the runTimeDiscretes function is called. This
may be undesirable for the logic of some control elements that are predicated on being the very first calculations
to run on any given time-step.

toleranceScaleFactor = 1.0000;
The multiplicative factor by which the convergence tolerances are loosened for transient solutions (as compared to
steady-state solutions).

userTimeStepFunction = "mySteppingFunction()";
User-defined real expression (entered as a string) used to compute the time-increment for each point in the
transient run. Used only when timeStepMethod is set to USER_DEFINED. The default is blank (there IS no
default).

discreteCalcsOrder = "BEFORE";
Determines whether the time discrete calculations are run prior to running the continuous portion of the model or
after. The switch is set to BEFORE as a default. If the switch is set to AFTER then the associated functions
preTimeDiscretes(), runDiscreteCalc(), and postTimeDiscretes() will all be run after the final explicit integrations
but before updateHistory(). See the Developers Guide for a further description of these functions.

17.1.9 Transient Output (Calculated) Attributes

dtForce
The time-step calculated from time-discrete object requests in the Model.

initialized
A Boolean flag (1 or 0, TRUE or FALSE) which indicates whether the TransientExecutive and the objects it
controls have been initialized (including their transient histories). Setting this attribute to zero will have the same
effect as the initializeHistory function overall.
**stopFlag**
A Boolean flag (1 or 0, TRUE or FALSE) which is set to the results of `testRunTermination` from within the `run` function.

**timeStep**
Depending on the baseTimeStep set by the user, the adaptive time-stepping algorithm, and synchronization with time-discrete objects in the Model, the actual timeStep at any given point in the simulation is calculated.

### 17.2 Integrator (NCPStateIntegrator class)
The Integrator class is derived from the Dependent class; it has the same functions and attributes as the Dependent class, but behaves differently when running in transient mode.

#### 17.2.1 Creating an Integrator
The minimum required information to define an Integrator is shown below.

```c++
Integrator integrator_name {
    stateName = "state_variable_name_in_model";
    derivativeName = "state_derivative_name_in_model";
    eq_lhs = "expr_for_lhs_of_steady-state_equality";
    eq_rhs = "expr_for_rhs_of_steady-state_equality";
}
```

An example of an Integrator definition is as follows.

```c++
Integrator ShaftInertia {
    stateName = "HP_SHAFT.Nmech";
    derivativeName = "HP_SHAFT.dNqdt";
    eq_rhs = "HP_SHAFT.trqIn";
    eq_lhs = "HP_SHAFT.trqOut";
    integrationType = "TRAPEZOIDAL";
}
```

Integrators are added to the solver as Dependents (there is no addIntegrator function in the solver).

```c++
solver.addDependent ("Manifold Pressure"); // can use addDependent
// because it is a kind of
// dependent
// (etc.)
```

#### 17.2.2 Implicit Integration
**Implicit** integration refers to a predictor-corrector method. Some integration methods, such as Gear and Trapezoidal, depend on knowing the state derivatives at the new time point. This derivative may, itself, depend on the value of the state at the new time (for example, the time-derivative of shaft speed depends on the net torque exerted on the shaft, but the torques produced by the various components depend on the speed). To resolve this circular dependence, the state independent parameters are varied until their values are consistent with the values of the state derivatives, integrated over the current time-step. This iterative process is handled by the Solver in a manner similar to algebraic balances. The predictor passes discussed previously determine an estimate for the state independent variables at the new time ($t_n$). The **corrector errors** are defined (as described below) such that when they are zero, the states values will be consistent with the derivative values and integrator method used. The Solver controls the corrector iterations by which the corrector errors are driven to within a specified tolerance of zero.

In a transient simulation, the model computes the state derivative values, as well as a characteristic time-constant for the state. A generic expression for time-constant can be calculated using the expression
In this expression, $x$ is the state value, $y_1$ and $y_2$ are the values of the dependent parameters which would be used to compute steady-state error ($\text{torque}_{\text{in}}$ and $\text{torque}_{\text{out}}$ when $x$ is the shaft-speed state, for example).

Transient models may also be run in a steady-state mode (in order to set the initial conditions for a transient run, for example). The error terms for a steady-state run are not the same as for a transient simulation, as described above. In principle, it is possible to find the steady-state condition by driving the derivative values to zero (the derivative value itself would be the error in this case). In general, however, finding the steady-state solution by driving the derivatives to zero is discouraged. Instead, the component developer should define a dependent equality between two parameters which should be enforced at a steady-state condition. For example, instead of driving the derivative of the shaft speed to zero, the steady-state solver should drive the input torque and output torque from the shaft to be equal. Of course, this condition is physically equivalent to a zero shaft speed derivative, but it is generally better in terms of solver convergence to choose a dependent equality in which neither side is zero. The steady-state dependents must therefore be chosen on a case-by-case basis (there is not standard form as described for integrators above).

The implicit integration methods supported by the NPSS are described below. Each state variable can be independently assigned a different integrationType.

### 17.2.2.1 Trapezoidal Integration

When used as a corrector, it's also known as the Modified Euler formula. It assumes a linear variation in the state derivative with respect to time. Integration formulae that employ past state variable derivatives introduce a spurious lead effect into the response of the model which can be shown using the z-transform analysis. When the time increment employed is large compared to the response of the state variable, an oscillatory response is noted that is eliminated when either the Gear 1st- or 2nd-order formula is used. Trapezoidal integration uses the average of the state derivatives at the previous time ($t_{n-1}$) and current time ($t_n$).

$$x_{t+n} = x_t + \frac{1}{2} \left( \frac{dx}{dt}_{t+n} + \frac{dx}{dt}_t \right) * dt$$

### 17.2.2.2 Gear First-Order Integration

The 1st order Gear integration method, sometimes referred to as the “backwards Euler” method, is the best from a warranty standpoint. Although less accurate than the 2nd order method (next section), it does not share the stability properties of the higher order methods. The continuous and discrete stability domains are the same for this formula regardless of the size of the time increment. Gear’s first-order integration method uses only the new state derivatives (at $t_n$).

$$x_{t+n} = x_t + \frac{dx}{dt}_{t+n} * dt$$

### 17.2.2.3 Gear Second-Order Integration

The 2nd order Gear integration method is the best multi-point compromise, at least for implicit applications. Its weakness is that there is a small region near the imaginary “z” axis and the origin of the z-plane trajectory where "false" stability can occur. This means that for small (or negative) damping ratios and large time increments, a model can appear stable when the response of the real system would oscillate or diverge. At smaller time steps, the
correct (oscillatory or divergent) response is obtained. Therefore, when a new model is created, it is important to explore the potential for significant error by running a few test cases at time increments small enough to uncover unstable parasitic effects.

\[
x_{t+dt} = x_t + \left( \frac{2}{3} \frac{dx}{dt_t} + \frac{1}{3} \frac{dx}{dt_{t+dt}} \right) * dt
\]

(16)

17.2.3 Explicit Integration

Although implicit integration methods may improve the stability and accuracy of a transient simulation, closing the corrector error requires additional Solver iterations that may significantly impact the computational performance of the simulation overall. When the transient behavior is relative linear and well-behaved in time, explicit integration methods may offer an alternative which is less computationally intensive. Explicit integration methods do not include derivative values at the new time-step, and therefore do not require an iteration with the state values.

Explicit integration of the selected states is performed at the same point as the predictor passes.

The user may which select states (none, some, or all states in the Model) are to be explicitly integrated. Explicit integration does not require the Solver to close an error term between the state and its derivative. Thus when an Integrator is set to Euler Integration, a search is made to locate a matching Independent. When one is found, the Integrator and the Independent are removed from the list of Independents and Dependents used in the iterative solution.

The Euler explicit integration method is supported in the NPSS software. NPSS does limit which Integrators can be set to Euler Integration. To switch a given Integrator to use Euler, a state substitution variable must not be used (See Section 17.2.8 for more information). The practical impact is that there must be an Independent in the current solver setup with a varName attribute that points directly to the same model variable that is referenced by the stateName attribute of the Integrator being set to Euler. This is an indirect indication that the underlaying model calculations are structured to use the state variable value directly as input, rather than using another variable as input and then calculating the state value from the input value. The latter is the case when a state substitution variable is used as the Independent.

17.2.3.1 Euler Integration

Euler integration uses the state derivative value from the beginning of the time-step to predict the state value at the end of the time step.

\[
x_{t+dt} = x_t + \frac{dx}{dt_t} * dt
\]

(17)

17.2.4 Integrator Functions

The Integrator has the same functions as the Dependent class.

Note: When you add an integrator to the solver configuration, use the addDependent function. For example,

```java
    solver.addDependent("ShaftInertia");
```

Similarly, the removeDependent function is used to remove an integrator from the solver.
17.2.5 Integrator Input Attributes

In addition to those which it inherits from the Dependent class, the Integrator has the following attributes:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>derivativeName</td>
<td>string</td>
<td>Name of a variable holding the current value of the time derivative of the variable named by stateName.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>desiredDerivValue</td>
<td>real</td>
<td>The value of the derivative to be matched by the Solver when the local solutionMode is set to &quot;SET_DERIVATIVE.&quot;</td>
<td>0.0000</td>
</tr>
<tr>
<td>eq_RefTransient</td>
<td>string</td>
<td>Expression evaluated only during a transient run that yields a value of yRef. If an expression is not given, then the expression in eq_Ref is used during transient as well as steady-state operation. And if eq_Ref is not given, then eq_rhs is used to determine yRef, and finally if eq_rhs is zero or evaluates to zero, a yRef of one is used.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>integrationType</td>
<td>Option variable</td>
<td>Integration method for this integrator. Different integrators in the same system can use different integration methods. Allowable values are &quot;GEAR_1ST_ORDER&quot;, &quot;GEAR_2ND_ORDER&quot;, &quot;TRAPEZOIDAL&quot;, which are implicit methods, and &quot;EULER&quot; which is an explicit method. If unset, the default set in the TransientExecutive is used. It defaults to &quot;GEAR_1ST_ORDER&quot;.</td>
<td>Transient. IntegrationType</td>
</tr>
<tr>
<td>stateName</td>
<td>string</td>
<td>Name of the variable whose time-varying value is to be determined by integrating the derivative specified by derivativeName.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>timeConstantExpr</td>
<td>string</td>
<td>An expression used to calculate the state's time-constant (tau). If not set, the time-constant is computed internally (refer to the Transient E-Spec document for more information regarding the internally used expression).</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

17.2.6 Locking Attributes for Integrator

The SolverExecutive sets some attribute default values for each of its associated integrators. If you specifically set one of these attributes in an integrator, the default value will be ignored. In addition to those inherited from the Dependent, the Integrator has the following additional lock attributes.

When the corresponding attribute is set for a specific independent, the lock parameter is set to TRUE. When you want the default to be used again, you should set the corresponding lock attribute to FALSE again.

**Note:** The default value for all the "lock" variables below is FALSE.
### Table 65. Integrator Locking Attribute

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lockIntegrationType</td>
<td>An attribute that determines whether the TransientExecutive's default</td>
</tr>
<tr>
<td></td>
<td>integrationType will overwrite the current value. When lock is TRUE, the default</td>
</tr>
<tr>
<td></td>
<td>will NOT overwrite the current value.</td>
</tr>
</tbody>
</table>

### 17.2.7 Integrator Output (Calculated) Attributes

In addition to those which it inherits from the Dependent class, the Integrator has the following output attributes.

### Table 66. Integrator Output Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>derivative</td>
<td>The value of the state derivative computed by the Model.</td>
</tr>
<tr>
<td>state</td>
<td>The value of the state variable (set by Solver).</td>
</tr>
<tr>
<td>stateDemand</td>
<td>The value of the state variable calculated by the Integration equation.</td>
</tr>
<tr>
<td>timeConstant</td>
<td>The current value of the state's time-constant.</td>
</tr>
</tbody>
</table>

### 17.2.8 State Substitution Variables

In some cases, it is advantageous for the Solver to control a model independent variable other than the state variable itself. Consider a compressible gas volume, for example. The state variable for the gas is usually the density, whose time-derivative depends on the difference between the flows entering and leaving the volume. The flows, in turn, are dependent on the pressure of the volume, not the density. In this case, it would be easier to have the Solver control the volume pressure as the Independent, using this input to compute the volume density (state value) and the flows affecting the volume. In this case, therefore, the pressure is the Independent while the density is still the state whose derivative is referenced by the Integrator. By keeping the state Integrator and Independent separate, the NPSS makes this easy to setup.

```plaintext
Independent pressIndep {
    varName = "volume.Pt";
}

Integrator densInteg {
    stateName = "volume.density";
    derivativeName = "volume.dRHOdt";
    eq_lhs = "volume.Fl_I.W";
    eq_rhs = "volume.Fl_O.W";
    integrationType = "GEAR_2ND_ORDER";
}
```

The only potential problem that this sort of state substitution presents is when an explicit integration method is desired. In such cases, the state itself is typically updated. In the example above, there is no easy way to update the pressure based on integrating the density. This should be taken into account when deciding whether state-substitution will be used in a model.
17.3 Example Program

The following First Order Lag test program demonstrates the functions and attributes discussed in sections 17.1 and 17.2, above.

17.3.1 Model File

```plaintext
// First Order Lag Model
// for testing the NPSS Transient capability
//

class InterpLag extends Element {
    real x, dxdt, y, tau, localTime;

    void calculate() {
        dxdt = (y-x) / tau;
    }
}

// analytic recursion relation for a "clamped ramp"
// to be run in the ExecutionSequence and compared to the
// implicit solution.

class AnalyticFOL extends Element {
    real x, y, tau;

    void calculate() {
        x = y * (1.000 - (tau/timeStep)*exp(-time/tau)*(exp(timeStep/tau)-1.000));
    }
}

Element InterpLag myLag {   }
Element AnalyticFOL benchMark {
    real theError;

    void postexecute() {
        theError = myLag.x - x;
    }
}

OutFileStream myTransOut { filename = "cout"; append = TRUE; }

DataViewer VarDumpViewer dumper {
    variableList = {"myLag.*", "solver.*", "xInteg.*", "xIndep.*");
    outStreamHandle = "myTransOut";
}

DataViewer CaseRowViewer transOut {
    titleBody = "Transient Output for First-Order-Lag model";
    titleVars = {};
    pageWidth = 0;
    pageHeight = 0;

    if (getenv("NPSS_CONFIG") == "nt"){
        variableList = {"time", "solver.iterationCounter: ???? = iters",
                        "solver.numJacobians: ???? = Jacobians";
```
17.3.2 First Order Lag Test File

#include "FirstOrderLag.mdl"

// create a function that will reset the model to initial conditions.
void resetModel ()
{
    // reset the input and run once to reset the derivative (output)
    myTransOut << "\n\nRun a ONE_PASS to reset derivative in model\n";
    myLag.x = 0.0000;
    time = 0.0000;
    setOption("solutionMode", "ONE_PASS");
    run();
    transOut.reset();
    setOption("solutionMode", "TRANSIENT");
    initializeHistory();

    // force it start with a new matrix
    solver.forceNewJacobian = TRUE;
}

    // create an element for setting the input step function.
    class Input extends Element{
        real stepTime, stepVal, outputVal;

        void calculate() {
            if (time <= stepTime)
{  
  outputVal =  0.0000;  
}  
else  
{  
  outputVal =  stepVal;  
}

// create an instance of the input element  
Element Input yInput {  
  stepVal =  5.0;  
  stepTime =  0.00;  
  void postexecute() { myLag.y = outputVal; }  
}

// put the input element in the preExecutionSequence.  
solver.executionSequence.remove("yInput");  
solver.preExecutionSequence.append("yInput");

// set initial conditions  
myLag.x =  0.0000;  
myLag.tau =  0.5;  
benchMark.x =  0.0000;  
benchMark.tau =  0.5;  
benchMark.y =  yInput.stepVal;

// set the independent reference value (since x starts from zero).  
xIndep.indepRef = "1.00";

// set up the baseline transient run options  
setOption("solutionMode", "TRANSIENT");  
transient.stopTime =  2.5000;  
transient.baseTimeStep =  0.025;

// set the debug options so we can track the number of passes,  
// etc. in the transient run.  
solver.debugLevel = "NONE";

// optional settings to improve convergence  
//solver.toleranceBroydenScalar = 1.e-10;

// Test 1.1 Baseline run to quiescence  
myTransOut << "=======================================================
  TRANSIENT TESTS WITH IMPLICIT FIRST-ORDER LAG MODEL
=======================================================

Test 1.1 Baseline Case - run to quiescence
(dt = 0.025)
--------------------------------------------
";  
//cout << "Test 1.1
";

transient.terminateCondition = "quiescence()";
run();

//update transient output report;  
myTransOut << "\n"; transOut.display();

// Test 1.2 Run Steady-state point w/o resetting  
myTransOut << "\n\nTest 1.2 Run Steady-state point w/o resetting"
<< "\n--------------------------------------------------------\n";

352  18  Transient Reference Guide
cout << "Test 1.2\n";
setOption("solutionMode","STEADY_STATE");
run();
dumper.display();
transOut.reset();

// Test 1.3 Reset initial conditions and run to an x=.632 * 5.0
myTransOut << "\n\nTest 1.3 Reset initial conditions and run to an x=.632 * 
5.0" <<
        "\n-----------------------------------------------------------
---\n";
//cout << "Test 1.3\n";
resetModel();
transient.terminateCondition = "myLag.x > (0.632 * yInput.stepVal)";
myTransOut << "\n\n Now run transient ...
"
run();
myTransOut << "\n"; transOut.display();
transient.terminateCondition = "quiescence()";

// Test 1.4 Run Steady-state point w/o resetting
myTransOut << "\n\nTest 1.4 Run Steady-state point w/o resetting"
<<
        "\n-----------------------------------------------------------\n";
//cout << "Test 1.4\n";
setOption("solutionMode","STEADY_STATE");
run();
dumper.display();
transOut.reset();

// Test 3.1.3 - Run with different time-step
myTransOut << "\n\nTest 3.1.3 - Run baseline with dt = 0.05\n" <<
        "\n-----------------------------------------------------------\n";
//cout << "Test 3.1.3\n";
resetModel();
transient.baseTimeStep = 0.05;
run();
myTransOut << "\n"; transOut.display();

// Test 3.3 - run with adaptive time-stepping on
myTransOut << "\n\nTest 3.3 run 1.1 with timeStepMethod = ADAPTIVE\n" <<
        "\n-----------------------------------------------------------\n";
//cout << "Test 3.3\n";
resetModel();
transient.baseTimeStep = 0.025;
transient.timeStepMethod = "ADAPTIVE";
run();
myTransOut << "\n"; transOut.display();
// Test 3.4.1 - change min and max time steps
myTransOut << "\nTest 3.4.1 - set minTimeStep = 0.02; and maxTimeStep = 0.05\n"
" "------------------------------------------------------------
\n";  // cout << "Test 3.4.1\n"
resetModel();
transient.baseTimeStep = 0.025;
real oldMin = transient.minTimeStep;
real oldMax = transient.maxTimeStep;
transient.minTimeStep = .02;
transient.maxTimeStep = 0.05;
run();
myTransOut << "\n"; transOut.display();
transient.minTimeStep = oldMin;
transient.maxTimeStep = oldMax;

// Test 3.4.2 - change dxTransLimit and dxTransLimitType
myTransOut << "\nTest 3.4.2 - set dxTransLimit = 0.25 and dxTransLimitType = ABSOLUTE\n"
" "------------------------------------------------------------
\n";  // cout << "Test 3.4.2\n"
resetModel();
transient.baseTimeStep = 0.025;
real oldLimit = transient.dxTransLimit;
string oldType = transient.dxTransLimitType;
transient.dxTransLimit = 0.25;
transient.dxTransLimitType = "ABSOLUTE";
run();
myTransOut << "\n"; transOut.display();
transient.dxTransLimit = oldLimit;
transient.dxTransLimitType = oldType;

// Test 3.4.3 - change dtExpandTolerance
myTransOut << "\nTest 3.4.3 - increase dtExpandTolerance by factor of three.\n"
" "------------------------------------------------------------
\n";  // cout << "Test 3.4.3\n"
resetModel();
transient.baseTimeStep = 0.025;
real oldTol = transient.dtExpandTolerance;
transient.dtExpandTolerance = 3.0 * oldTol;
myTransOut << "    (new dtExpandTolerance = " << transient.dtExpandTolerance
" " << ");
run();
myTransOut << "\n"; transOut.display();

transient.dtExpandTolerance = oldTol;

// Test 3.4.4 - change dtRelaxFactor
myTransOut << "\nTest 3.4.4 - set dtRelaxFactor = 1.0\n" << "------------------------------------------------------------\n";
//cout << "Test 3.4.4\n";
resetModel();
transient.baseTimeStep = 0.025;
real oldRelax = transient.dtRelaxFactor;
transient.dtRelaxFactor = 1.0000;
run();
myTransOut << "\n"; transOut.display();

transient.dtRelaxFactor = oldRelax;

transient.timeStepMethod = "CONSTANT_DT";
transient.baseTimeStep = 0.025;

// run to steady-state as setup for next point.
setOption("solutionMode","STEADY_STATE");
run();
// clear viewers
dumper.reset();
transOut.reset();

// Test 4.3 Run to a SET_DERIVATIVE (for x) of 1.0
myTransOut << "\n\nTest 4.3 Run x to a SET_DERIVATIVE = 1.0" << "\n-----------------------------------------------\n";
//cout << "Test 4.3\n";
setOption("solutionMode", "STEADY_STATE");
// override on xInteg alone
xInteg.solutionMode = "SET_DERIVATIVE";
xInteg.desiredDerivValue = 1.0;
run();
dumper.display();
transOut.reset();

myTransOut << "\n\nContinue run from there to quiescence" << "\n-----------------------------------------------\n";

setOption("solutionMode", "TRANSIENT");
initializeHistory();
transient.stopTime = 10.;
transient.terminateCondition = "quiescence()";
run();
myTransOut << "\n"; transOut.display();

// Test 6.2.1 - Run baseline with integrationType = "GEAR_2ND_ORDER"
myTransOut << "\n\nTest 6.2.1 Run baseline case with integrationType" << " = GEAR_2ND_ORDER"
Test 6.2.1

Test 6.2.2 Run baseline case with integrationType = "GEAR_1ST_ORDER"

Test 6.2.3 Run baseline case with integrationType = "TRAPEZOIDAL"

Test 6.2.4.1 Run baseline case to stopTime = 10.0 with integrationType = "EULER"

Test 6.2.5.1 Run baseline with integrationType = "GEAR_2ND_ORDER" and baseTimeStep = 2.5
myTransOut << "\n\nTest 6.2.5.1 Run baseline case with integrationType" << " = GEAR_2ND_ORDER\nand baseTimeStep = 2.5"
<< "\n------------------------------------------\n";  //cout << "Test 6.2.5.1\n";
resetModel();

transient.baseTimeStep = 2.5; // must set timeStep in TransientExec after
// initializeHistory is done. Otherwise, the baseTimeStep overwrites it.
transient.maxTimeStep = 3.0; // need to increase max too.
transient.terminateCondition = "";
transient.stopTime = 10.0; // run out to 10 sec.
transient.integrationType = "GEAR_2ND_ORDER";
run();
myTransOut << "\n"; transOut.display();

// Test 6.2.5.2 - Run baseline with integrationType = "TRAPEZOIDAL"
myTransOut << "\n\nTest 6.2.5.2 Run baseline case with integrationType" << " = TRAPEZOIDAL\nand baseTimeStep = 2.5"
<< "\n------------------------------------------\n";  //cout << "Test 6.2.5.2\n";
resetModel();

transient.integrationType = "TRAPEZOIDAL";
run();
myTransOut << "\n"; transOut.display();

// put things back to original condition
transient.baseTimeStep = .025;
transient.maxTimeStep = 1.0;
transient.integrationType = "GEAR_1ST_ORDER";

// Test 6.3.2 - Run baseline with extraIndepUpdate = TRUE and
// toleranceScaleFactor = 1.0
myTransOut << "\n\nTest 6.3.2 Run baseline case to quiescence" << " with extraIndepUpdate = TRUE and toleranceScaleFactor = 1.0"
<< "\n------------------------------------------\n";  //cout << "Test 6.3.2\n";
resetModel();
transient.extraIndepUpdate = TRUE;
transient.terminateCondition = "quiescence()";
run();
myTransOut << "\n"; transOut.display();

// Test 6.3.3 - Run baseline with extraIndepUpdate = FALSE and
// toleranceScaleFactor = 10.0
myTransOut << "\n\nTest 6.3.3 Run baseline case to quiescence" << " with extraIndepUpdate = FALSE and toleranceScaleFactor = 10.0"
<< "\n------------------------------------------\n";  //cout << "Test 6.3.3\n";
resetModel();
transient.extraIndepUpdate = FALSE;
transient.toleranceScaleFactor = 10.0;

run();

myTransOut << "\n"; transOut.display();

// Test 6.3.4 - Run baseline with extraIndepUpdate = TRUE and
// toleranceScaleFactor = 10.0
myTransOut << "\n\nTest 6.3.4 Run baseline case to quiescence"
<< "\n\nwith extraIndepUpdate = TRUE and toleranceScaleFactor = 10.0"
<< "\n\n----------------------------------------------------------------\n";
//cout << "Test 6.3.4\n";
resetModel();
transient.extraIndepUpdate = TRUE;

run();

myTransOut << "\n"; transOut.display();

// Test 6.4.2 - Run baseline case to quiescence with predictorType = LAGRANGE
myTransOut << "\n\nTest 6.4.2 Run baseline case to quiescence"
<< "\n\nwith predictorType = LAGRANGE"
<< "\n\n----------------------------------------------------------------\n";
//cout << "Test 6.4.2\n";
resetModel();
transient.extraIndepUpdate = FALSE;
transient.toleranceScaleFactor = 1.0;
transient.predictorMethod = "LAGRANGE";

run();

myTransOut << "\n"; transOut.display();

// Test 6.4.3 - Run baseline case to quiescence with predictorType = NONE
myTransOut << "\n\nTest 6.4.3 Run baseline case to quiescence"
<< "\n\nwith predictorType = NONE"
<< "\n\n----------------------------------------------------------------\n";
//cout << "Test 6.4.3\n";
resetModel();
transient.predictorMethod = "NONE";

run();

myTransOut << "\n"; transOut.display();
17.3.3 Test Output File
Because of its length, only selected portions of the test output file have been included below.

**TRANSIENT TESTS WITH IMPLICIT FIRST-ORDER LAG MODEL**

Test 1.1 Baseline Case - run to quiescence (dt = 0.025)

<table>
<thead>
<tr>
<th>Case</th>
<th>time</th>
<th>iters</th>
<th>Jacobians</th>
<th>Broydens</th>
<th>myLag.y</th>
<th>myLag.x</th>
<th>benchMark.x</th>
<th>benchMark.error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.12294</td>
<td>2.50E-02</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>5.00000</td>
<td>0.23810</td>
<td>9.52381</td>
<td></td>
</tr>
<tr>
<td>0.36080</td>
<td>5.00E-02</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.46485</td>
<td>9.07029</td>
<td></td>
</tr>
<tr>
<td>0.58706</td>
<td>7.50E-02</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.68081</td>
<td>8.63838</td>
<td></td>
</tr>
<tr>
<td>0.80228</td>
<td>1.00E-01</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.88649</td>
<td>8.22702</td>
<td></td>
</tr>
<tr>
<td>1.00700</td>
<td>1.25E-01</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.08237</td>
<td>7.83526</td>
<td></td>
</tr>
<tr>
<td>1.2174</td>
<td>1.50E-01</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.26892</td>
<td>7.46215</td>
<td></td>
</tr>
<tr>
<td>1.38699</td>
<td>1.75E-01</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.44659</td>
<td>7.10681</td>
<td></td>
</tr>
<tr>
<td>1.56320</td>
<td>2.00E-01</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.61580</td>
<td>6.76839</td>
<td></td>
</tr>
</tbody>
</table>

Note worthy differences between the next two runs are that: In Test 6.2.3 (integrationType = TRAPEZOIDAL) the benchmark error is much lower than in Test 6.2.4.1 (integrationType = EULER). However, in test 6.2.4.1 the number of iterations = 0 at each point.

Test 6.2.3 Run baseline case with integrationType = TRAPEZOIDAL

Run a ONE_PASS to reset derivative in model

<table>
<thead>
<tr>
<th>Case</th>
<th>time</th>
<th>iters</th>
<th>Jacobians</th>
<th>Broydens</th>
<th>myLag.y</th>
<th>myLag.x</th>
<th>benchMark.x</th>
<th>benchMark.error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.12294</td>
<td>2.50E-02</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5.00000</td>
<td>0.12195</td>
<td>9.75610</td>
<td></td>
</tr>
<tr>
<td>0.36080</td>
<td>5.00E-02</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.35990</td>
<td>9.28019</td>
<td></td>
</tr>
<tr>
<td>0.58706</td>
<td>7.50E-02</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.58625</td>
<td>8.82750</td>
<td></td>
</tr>
<tr>
<td>0.80228</td>
<td>1.00E-01</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.80156</td>
<td>8.39689</td>
<td></td>
</tr>
<tr>
<td>1.00700</td>
<td>1.25E-01</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.00636</td>
<td>7.98728</td>
<td></td>
</tr>
<tr>
<td>1.20174</td>
<td>1.50E-01</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.20117</td>
<td>7.59766</td>
<td></td>
</tr>
</tbody>
</table>

18 Transient Reference Guide 359
Test 6.2.4.1 Run baseline case to stopTime = 10.
with integrationType = EULER

Run a ONE_PASS to reset derivative in model

Transient Output for First-Order-Lag model

<table>
<thead>
<tr>
<th>Case</th>
<th>time</th>
<th>iters</th>
<th>Jacobians</th>
<th>Broydens</th>
<th>myLag.y</th>
<th>myLag.x</th>
<th>benchMark.x</th>
<th>benchMark.error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.50E-02</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.00000</td>
<td>10.00000</td>
<td>-1.2294E-01</td>
</tr>
<tr>
<td>0.12294</td>
<td>5.00E-02</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.25000</td>
<td>9.50000</td>
<td>-1.1080E-01</td>
</tr>
<tr>
<td>0.36080</td>
<td>7.50E-02</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.47500</td>
<td>9.02500</td>
<td>-9.9556E-02</td>
</tr>
<tr>
<td>0.58706</td>
<td>1.00E-01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.71313</td>
<td>8.57375</td>
<td>-8.9153E-02</td>
</tr>
<tr>
<td>0.80228</td>
<td>1.25E-01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>0.90747</td>
<td>8.45065</td>
<td>-7.9534E-02</td>
</tr>
<tr>
<td>1.00700</td>
<td>1.50E-01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.13110</td>
<td>7.37381</td>
<td>-7.0648E-02</td>
</tr>
<tr>
<td>1.20174</td>
<td>1.75E-01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.32454</td>
<td>7.35092</td>
<td>-6.2446E-02</td>
</tr>
<tr>
<td>1.38699</td>
<td>2.00E-01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.50831</td>
<td>6.98337</td>
<td>-5.4882E-02</td>
</tr>
<tr>
<td>1.56320</td>
<td>2.25E-01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.68290</td>
<td>6.63420</td>
<td>-4.7913E-02</td>
</tr>
<tr>
<td>1.73081</td>
<td>2.50E-01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.00000</td>
<td>1.84875</td>
<td>6.30249</td>
<td>-4.1498E-02</td>
</tr>
</tbody>
</table>

17.4 Functions for Time-Discrete Model Objects

The functions below help manage simulation time and perform internal time-dependent operations for NPSS model objects. For more information regarding these functions, refer to the Developer's Guide.

void initializeHistory();
Sets the initial values of time-dependent history variables and vectors used to perform internal calculations.

real nextExecTime(real proposedTime);
Informs the TransientExecutive when the model must execute again in order to meet the objects functional requirements during a transient simulation.
int runDiscreteCalcs(real startTime, real endTime);
Performs calculations which must be performed only once per time step, regardless of which model sequencer the
element belongs to (preExecutionSequence, executionSequence, or postExecutionSequence).

void updateHistory();
Updates internal variables and time-dependent histories after the model has converged at the current time-step.

void postConvergence()
This function is called immediately after convergence in the NPSSSteadyStateSolver::run() routine. If the user
does not define this function, no additional functionality is invoked.

void updateDiscretes()
The user/modeler can define this function in a similar fashion to the way they can define runDiscreteCalcs().
This function has no arguments and does not return anything. As with runDiscreteCalcs() and similar
functions, it is acceptable for the user/modeler not to provide a function called updateDiscretes(). This is
strictly an optional function that the user could define. This function is called at the end of the time step but prior to
calling postTimeStep() described below.

void preTimeStep()
This function is called in the NPSSTransientSolver::run() routine, immediately following the point where the new
time and time step values are set. If the user does not define this function, no additional functionality is invoked.

void preTimeDiscretes()
This function is called immediately prior to the call to runTimeDiscretes in the NPSSTransientSolver::run()
routine. If the user does not define this function, no additional functionality is invoked.

void postTimeDiscretes()
This function is called immediately after the call to runTimeDiscretes in the NPSSTransientSolver::run()
routine. If the user does not define this function, no additional functionality is invoked.

void postPredictors()
The location of this function would be after predictTransIndepUpdates and before any integrations in the
NPSSTransientSolver::run() routine. If the user does not define this function, no additional functionality is invoked.

void postTimeStep()
The location of this function would be immediately prior to exiting the NPSSTransientSolver::run() routine. If the
user does not define this function, no additional functionality is invoked.

17.4.1 Time Discrete Example Test File
The following example demonstrates the functions described above. The program performs runs at various
frequencies above and below the baseTimeStep.

class Clock extends Element {
    real nextTime, prevTime, dtClock, signal, x, y, sigFreq;
    void nextExecTime(real currExecTime) {
        if (time < 0.4999) {
            dtClock = 0.025;
        } else if ((time > 0.4999) && (time < 0.9999)) {
            dtClock = 0.020;
        } else if ((time > 0.9999) && (time < 1.9999)) {
            dtClock = 0.040;
        } else if ((time > 1.9999) && (time < 2.4999))
```cpp
{ dtClock = 0.050; }
if (time > 2.4999)
{ dtClock = 0.025; }

nextTime = prevTime + dtClock;
cout << "\nnext time to run discretes = " << nextTime << "\n";
return nextTime;
}

int runDiscreteCalcs(real start, real stop) {
    if (abs(time - nextTime) > 0.000001) { return FALSE; }
    prevTime = time; // update the prevTime for next cycle.
    signal = 1.0001 + sin(sigFreq * time);
cout << "discrete calcs run at time = " << time << " with input = "
    << signal << "\n";
    return TRUE;
}

void updateHistory() { cout << "\n    continuous model runs at time = "
        << time << " ... x = "
        << x << " updated\n"; }

void initializeHistory() { cout << "initialized\n"; }

void calculate() {
    y = x**3.0;
}

MODELNAME = "testTimeDiscrete";

Element Clock myClock { sigFreq = 6.28; }; 

Independent xIndep {
    varName = "myClock.x";
    indepRef = "1.0";
}

Dependent yDep { eq_lhs = "myClock.y"; eq_rhs = "myClock.signal"; }

solver.addIndependent("xIndep"); 
solver.addDependent("yDep"); 
solver.defaultDxLimit = 0.25;
setOption("solutionMode", "TRANSIENT");

transient.baseTimeStep = 0.025;
transient.stopTime = 3.0;
transient.predictorMethod = "NONE";

cout << "====================================================
    TEST OF TIME DISCRETE HANDLING\n----------------------------------------------------
    The time-discrete element sets several different 
execution cycle periods, both smaller and larger 
than the baseTimeStep. \n" << "
"; 
```
run();

17.4.2 Time Discrete Test Output File

Portions of the output from this test program is shown below. For time < 0.5 seconds, the execution cycle period is 0.025 seconds.

===================================================================
<table>
<thead>
<tr>
<th>TEST OF TIME DISCRETE HANDLING</th>
</tr>
</thead>
<tbody>
<tr>
<td>The time-discrete element sets several different execution cycle periods, both smaller and larger than the baseTimeStep.</td>
</tr>
<tr>
<td>------------------------------</td>
</tr>
</tbody>
</table>
===================================================================

initialized

next time to run discretes = 0.025
discrete calcs run at time = 0.025 with input = 1.15646
    continuous model runs at time = 0.025 ... x = 1.04963 updated
next time to run discretes = 0.05
discrete calcs run at time = 0.05 with input = 1.30897
    continuous model runs at time = 0.05 ... x = 1.0939 updated
next time to run discretes = 0.075
discrete calcs run at time = 0.075 with input = 1.45388
    continuous model runs at time = 0.075 ... x = 1.13286 updated
next time to run discretes = 0.1

For 0.5 < time < 1.0 seconds, the execution cycle period is 0.020 seconds.

discrete calcs run at time = 0.5 with input = 1.00169
    continuous model runs at time = 0.5 ... x = 1.00056 updated
next time to run discretes = 0.52
discrete calcs run at time = 0.52 with input = 0.87641
    continuous model runs at time = 0.52 ... x = 0.956979 updated
next time to run discretes = 0.54
discrete calcs run at time = 0.54 with input = 0.753077
    continuous model runs at time = 0.54 ... x = 0.909801 updated
next time to run discretes = 0.56
discrete calcs run at time = 0.56 with input = 0.633635
    continuous model runs at time = 0.56 ... x = 0.858908 updated
next time to run discretes = 0.58

For 1.0 < time < 2.0 seconds, the execution cycle period is 0.040 seconds, and so on.
The following terms are relevant to the model. For terms relating specifically to the solver, see Section 16.9.

**API**
Application. Program Interface

**Assembly**
A collection of elements that is encapsulated and managed within NPSS. An assembly is treated by the model as though it were a single element.

**Block**
A group of statements that are grouped together as a unit. The statements that instantiate an object must be grouped together in a block.

**CMF**
Creation Method Facilities

**Comment**
A segment of an input file that is ignored by NPSS. Any text following a pair of slashes (//) to the end of the line is treated as a comment. Comments may also be enclosed between these delimiters: /* ... */.

**conditional branch**
A construct in which a test is made and a different set of statements executed depending on the outcome of the test. NPSS provides two keywords for constructing conditional branches: if and else.

**CORBA**
Common Object Request Broker Architecture. Users can write special code that encloses existing component code and then use CORBA to register it for access at run time.

**DataPorts**
Transfer a single numerical value, such as an engine measurement.

**data reduction**
Refers to the process of incorporating measured test data into an engine model, adjusting the model to match those measurements, and using the adjusted model to predict results at interesting conditions.

**DataViewer**
An object that allows users to display formatted and unformatted output. There are four kinds of data viewers: CaseColumnViewer, CaseRowViewer, Page Viewer, and VarDumpViewer. These viewers retrieve variable values and display them in the format specified in the input file.

**ESO**
An ESO (Engineering Status Object) is an NPSS model object class that holds a complete set of information about error/warning conditions. It includes attributes such as general description, an instance description, and context information. One of the ESO attributes is an 8-digit ESI (Engineering Status Indicator).

**Element**
The main building block of an engine model in NPSS. It is generally defined along component lines (e.g., compressor, turbine, burner, duct, and nozzle). Component elements are preprogrammed objects designed to model the thermodynamic/physical process they represent.

**expressions**
Formed by applying operators to variables or constants. An expression evaluates to a single value and can appear in a statement anywhere a variable or constant can appear. See Section 2.2.5.2 on expressions.

**Fluid Ports**
Contains a reference to fluid station, which actually stores the data. Data flows through the port in only one direction, from output to input port. Fluid ports transfer primary-gas flow properties between elements (e.g., flow rate, total temperature and pressure,
molecular constituents). Special fluid ports are available for bleeds and leakages.

**Fluid Station**

Holds all the fluid information being shared between two fluid ports.

**Fuel Ports**

Are used to transfer fuel properties from a FuelStart element into a Burner. See Section 4.3 on Ports.

**functions**

An action that can be performed at the request of other objects. An NPSS function accepts zero or more arguments, whose types must be specified, and returns a value, whose type must be also be specified. A function is called by supplying its name and its arguments (or empty parentheses if no argument). If the function returns a value, the function call itself is treated just like the value it returns and can be used in expressions or other statements.

**global functions**

A function that applies to NPSS as a whole and is not associated with any particular object. (The Function Summary contains some information on global functions.

**initialization statement**

Generally used to set a loop counter to an initial value. This statement is executed only once.

**instantiation**

Creating an example or instance of an object. Whenever an object is instantiated, it is given a unique name by which it is referenced thereafter.

**interactive mode**

Allows users to enter commands while running within the NPSS framework. Users can trigger this mode by entering a command-line option when NPSS is started or by entering a keyword in the input file.

**Linear Model Generator (LMG)**

An optional object created at run-time that generates a linear representation of the full non-linear model about a converged point of the non-linear model. A linear model relates changes in selected state derivative and model output variables to changes to the corresponding state and selected input variables. See Section 8 on Linear Models.

**link**

Connects two ports so data can flow between them

**loop**

A construct in which one or more statements are executed repeatedly until some terminating condition is met. Each repetition is called an iteration.

**Model**

The object in NPSS that contains all of the configuration information for a particular simulation.

**NCP**

National Cycle Program. This was the name of the early versions of NPSS; hence, the code contains references to NCP.

**NPSS**

Numerical Propulsion System Simulation. A computer simulation that realistically models the physical interactions that take place throughout an engine. The system is being designed to reduce 1) development time and 2) the need for expensive full-scale tests and experiments. Eventually, it will be able to implement one-, two-, and three-dimensional models of the engine flow field and structure and reconcile them with zero-dimensional component-based models.

**object-oriented program**

A program whose structure revolves around objects of various kinds, each of which has a set of attributes accessible by other objects, and certain functions—a set of actions particular to the object—that can be performed at the request of other objects.

**option variable**

A scalar text string variable with a finite number of possible values. For example, many NPSS objects classed as Elements contain an option variable named switchDes that has
two allowable values: "DESIGN" and "OFFDESIGN".

<table>
<thead>
<tr>
<th><strong>parent</strong></th>
<th>An object that contains another object.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port</strong></td>
<td>An object that connects elements, allowing them to communicate with each other. There are fluid, fuel, shaft (mechanical), data, and file ports. A port can be created only within an element, not a subelement.</td>
</tr>
<tr>
<td><strong>scope</strong></td>
<td>That part of the input where the variable can be referenced by name only, without a path.</td>
</tr>
<tr>
<td><strong>Shaft Ports</strong></td>
<td>Transfer mechanical properties such as torque and rotational speed between rotating components.</td>
</tr>
<tr>
<td><strong>Socket</strong></td>
<td>An object created within an element that acts as a placeholder at specific points within the element's execute() function. Users must specify where in the element execute function the socket will be called. If a socket containing an object is called, it will execute the object. Each socket is a distinctive type, and each object must meet the requirements of the socket type to &quot;fit&quot; into the socket. By using the socket attribute, socketType for compatibility checking, NPSS can detect when incompatible objects are called from an element.</td>
</tr>
<tr>
<td><strong>Solver</strong></td>
<td>The object within NPSS software that drives any system model to a converged state.</td>
</tr>
<tr>
<td><strong>Station</strong></td>
<td>Contains the state of a fluid, heat flux, or mechanical energy flux for a given point in space and time. A station also provides accessibility for reading the values of the parameters contained in the station.</td>
</tr>
<tr>
<td><strong>statement</strong></td>
<td>Syntactical unit that instructs NPSS to define or instantiate an object or access an existing object's attributes or capabilities.</td>
</tr>
<tr>
<td><strong>statement terminator</strong></td>
<td>In NPSS, the semi-colon (;) that typically ends a statement.</td>
</tr>
<tr>
<td><strong>Subelement</strong></td>
<td>Objects that are used to define the behavior of certain elements. A subelement supplies element performance parameters (efficiencies, pressure ratios, etc.) to elements or other subelements. In the NPSS hierarchy, a subelement is always the child of an element.</td>
</tr>
<tr>
<td><strong>Table</strong></td>
<td>An organized set of numeric data that is used to relate a desired output (dependent) to one or more input variables (independents). Tables may be of arbitrary dimensionality, from 1 to N.</td>
</tr>
<tr>
<td><strong>variable</strong></td>
<td>Basic object within NPSS that holds data values (numbers or text) and is accessible through the VariableContainer API. Variables may be integers, real numbers, or text strings, and they may be either scalars or arrays.</td>
</tr>
</tbody>
</table>
Index

A
A matrix .................................................. 201, 204, 205
absolute... .................................................. 327
action argument ........................................ 58
activateBreak () .......................................... 59
activateSecureMode() ................................ 256
active
ESOPRegistry attribute .......................... 119
adaptive time stepping .......................... 338
add(string stationName) ........................... 238
addConstraint () ....................................... 305, 309
addConstraint() ......................................... 177
addConstraintGroup () ............................ 305
addDependent () ....................................... 290
addDSV () ................................................ 290
addEq(" .............................................. 246
addGutter
GroupBlock variable ............................ 286
addIndependent () .................................. 290
addInput() ............................................. 202, 205
addInterface (string socketType) .............. 236
addNonEq" ........................................... 246
addOutput() ........................................... 205
addState() ........................................... 202, 205
addUnitBase() ......................................... 218
addUnitPref() ......................................... 220
addUnitRelation() .................................... 219
advanced solver functions ....................... 317
algebraic loop .................................... 327
aliases .................................................. 280
allFuel ................................................. 65
allowed state values .............................. 311
allowedInts .......................................... 27
allowedStrings ....................................... 27
allowedValues ....................................... 27, 315
DSV input attribute ................................ 315
anchor
GroupBlock variable ............................ 286
viewer attribute .................................... 282
anchor text block .................................... 106
append
output stream attribute .......................... 289
append() .............................................. 23
append(type value) ................................. 234
applyIndepMods() .................................. 249
array types ........................................... 16, 22
arrays .................................................. 22

functions ............................................ 234
one-dimensional .................................... 22
three-dimensional .................................. 23
two-dimensional .................................... 23
assembly .............................................. 5, 61, 80, 92, 364
attributes ............................................. 217
evaluation order in .................................. 81
attributes and functions
external components ............................... 237
attributes, of variables ........................... 16
AutoDoc ............................................. 277
AutoDocViewer .................................... 277, 287
autoSetup ............................................. 166, 168, 301, 306, 315
autoSolverSetup() ................................. 88, 92, 166, 171, 187, 236

B
B matrix ................................................. 201
baselineValue ........................................ 303
baseTimeStep ........................................ 192, 196, 342
batch mode ........................................... 6
bias ...................................................... 328
biasModel .............................................. 166
binary data storage ............................... ix, 211
bitwise operators .................................. 35
bldPumpPwr ......................................... 258
Bleed element ........................................ 85, 133
bleed extraction
Compressor element ............................... 84
bleed macro .......................................... 258
bleed ports ........................................... 84, 138, 310
linking ............................................... 85
bleed sinks ........................................... 84, 86
bleeds .................................................. 84, 138
creating .............................................. 85
modeling .............................................. 133
block .................................................... 11, 38, 39, 40, 41, 364
BlockDG() ........................................... 259
body
simple block attribute ............................ 283
body statements .................................... 38, 40
bottom
text block attribute ............................... 282
bounce
DSV output attribute .............................. 312, 317
break statement
in a loop .............................................. 41
breakpoint number ................................ 59
breakPoint() .......................................... 57
breakpoints .......................................... 57
for internal solver functions .................... 57
breaks
in files ................................................ 58
in interpreted functions ........................ 57
broydenLimit ......................................... 172, 292
burn() .................................................. 238
Burner element ...................................... 134

367
C
C matrix .......................................................... 201
C++ language .................................................. 9
calcDxLim() ..................................................... 318
calcmethod ...................................................... 208
calculate() ................................................... 115, 116
canCreate() .................................................... 254
case input .................................................... 140
case sensitive .................................................. 10
CaseColumnViewer........................................... 98, 101, 236, 280, 281
caseHeader ..................................................... 281
caseHeaderBody ............................................. 281
caseHeaderVars ............................................. 281
CaseRowViewer............................................. 98, 101, 235, 280, 281
CaseViewer .................................................. 101, 188
category
ESORegistry attribute ....................................... 117
CEA
running ....................................................... 65
cchar operator [] ............................................. 270
child
of an object .................................................. 10
clear() .......................................................... 290
clear() ....................................................... 91, 122, 206, 341
clearBreak () .................................................. 59
close() .......................................................... 95, 246, 247, 248
CMF ............................................................ 63, 364
command line options ..................................... 7
comments ..................................................... 12, 364
compAttr ..................................................... 104, 283
components
acquiring ..................................................... 61
instantiating ............................................... 129
Compressor element ........................................ 131
compressor map
modeling ..................................................... 131
compType ..................................................... 104, 283
compTypeFormat ............................................ 284
Computational Frame Time ................................ 334
computeTimeStep () ......................................... 341
condition argument ....................................... 58
conditional branches .................................... 34, 38, 364
conditional expression .................................. 38
in a loop ..................................................... 40
conditional preprocessing commands ................. 14
constraint dependents ................................. 176
constraint groups ........................................ 179
constraint slopes conflicts ......................... 181
constraintGroup ........................................... 315
ConstraintGroup ........................................... 179, 180
input attributes .......................................... 309
input commands ........................................... 309
constraintHandlingOpt .................................... 176, 180, 292
constraintNameList ........................................ 175, 179, 306, 309
constraintNames ........................................... 292
constraintPriorities ....................................... 175, 179, 306, 309
constraints ................................................... 5, 174, 328
disabling ..................................................... 181
removing ..................................................... 180
constraintsActive ......................................... 176, 297
constraintSlopes .......................................... 175, 179, 306, 309
constructs ..................................................... 34
contains(type value) ...................................... 234
containsConstraint () ...................................... 309
content
simple block attribute .................................. 283
continue
interactive debugging command ...................... 60
continue statement
in a loop .................................................... 41
continuousConverged .................................... 297
Control element .......................................... 137
Control Structures ........................................ 20
controlBounce ............................................. 312, 315
convergeContinuousModel () ......................... 318
converged .................................................. 162, 174, 297
convergence ................................................ 5, 328
convergenceLimit ......................................... 172, 292
convergenceRatio ......................................... 174, 297
copyFlow() .................................................. 238
copyFlowRemote() .......................................... 239
copyFlowStatic() .......................................... 239
copyFlowStaticRemote() .................................. 239
Corba .......................................................... 8
CORBA ...................................................... 1, 4, 6, 364
coverPage .................................................... 280
Creation Method Facilities (CMFs) .................... 61
curly braces
to enclose a block ....................................... 11
D
D matrix ........................................................ 201, 204, 205
-D varName=value ........................................... 7
data matching and analysis ......................... 124
data reduction ............................................. 364
DataPorts .................................................... 5, 73
dataset ......................................................... 211
DataViewers .................................................. 2, 6, 82, 87, 98, 235, 280, 364
dataSet ......................................................... 211
dataSet ......................................................... 211
dataSet ......................................................... 211
dataSet ......................................................... 211
functions .................................................... 235
 updating ..................................................... 99
DCLOD ......................................................... 8, 61, 63, 255
DCLOD_PATH .................................................. 110
DCLUDPATHS .................................................... 32
DCoITBlock ................................................ 104
 -debug ...................................................... 8
debug mode .................................................. 8
debugLevel ................................................... 172, 292
debugLevel = "ITERATION DETAILS" .................. 320
debugLevel = "ITERATION_FAILURES" ............... 320
debugLevel = "MATRIX DETAILS" ....................... 320
debugLevel = "NONE" .................................... 320
debugLevel = "RUN SUMMARY" .......................... 320
debugLevel = "SETUP DETAILS" ......................... 320
defaultDxLimit ............................................ 172, 292
defaultDxLimitType ....................................... 172, 292
defaultPerturbation ...................................... 173, 208, 292
defaultPerturbationType ............................... 173, 208, 292
defaultTolerance ........................................ 173, 293
defaultToleranceType ..................................... 173, 293
defaultXLimReport ......................................... 293
defIntFormat ............................................... 281, 284
defRealFormat ............................................. 281, 284
### H

- **-h, command line option** ........................................ 9
  - hasInterface() .................................................. 77, 227
  - haveStatic() .................................................. 239
  - HDF ................................................................. ix, 211
  - HDF file
    - creating .................................................. 211
    - HDF user-accessible functions ............................. 214
    - headers ..................................................... 104
    - in DataViewers ........................................... 102
  - height
    - empty text block variable .................................. 286
    - hidden .......................................................... 228
    - hide ......................................................... 227, 228
  - Hierarchical Data Format .................................... 211
  - hierarchical objects ........................................ 10

- **i**
  -<dimname> .......................................................... 8
  - **-i, command line option** .................................... 8
  - icl.postBreak() ................................................ 257
  - icl.preBreak() ................................................ 257
  - icl.preexecute() { } .......................................... 257
  - icl.start() .................................................... 257
  - ICLOD ............................................................ 63, 255
  - ICLOD (Interpreted Component Load on Demand) ......... 8
  - ICLOD_PATH ................................................... 62, 110
  - -iclofirst ..................................................... 8, 64
  - ICLODPATHS .................................................... 32
  - id
    - ESORegistry attribute ...................................... 118
  - iDescription
    - DSV input attribute ........................................ 316
    - variable attribute .......................................... 17, 75
    - variable attribute .......................................... 67
  - if construct ................................................... 38
  - implicit integration ........................................ 345
  - include path ................................................ 8
  - INCLUDEPATHS .................................................. 32
  - increment statement
    - in a loop .................................................. 40
  - independent variables ..................................... 5, 88
  - independentNames .......................................... 294
  - independents ................................................ 90, 140, 162, 328
  - functions .................................................... 246
  - input attributes ............................................ 166, 301
  - input/commands ............................................. 300
  - locking attributes ......................................... 167, 200, 302
  - output attributes .......................................... 303
  - independentValues .......................................... 294
  - indepRef ..................................................... 167, 301
  - index
    - for an array ................................................ 23, 40
    - specifies where exception occurred ...................... 123, 235
  - index number
    - of an array ................................................ 115
    - of an array ................................................ 22
  - index() .......................................................... 269, 273, 274
  - index(type value) ........................................... 234
  - indirect membership
    - of an object .............................................. 45
    - indirect membership operator ......................... 45
    - indirect wrappers ........................................ 260
    - initialization statement ................................ 365
      - in a loop ............................................... 40
    - initialization steps
      - of time update loop .................................... 330
    - initialize() ............................................... 318
    - initialized ................................................ 198
      - transient input attribute ............................ 343
      - transient output attribute ......................... 344
    - initializeHistory() ...................................... 190, 341, 360
    - intXFunction ................................................ 301
    - Inlet element .............................................. 130
    - input file .................................................. 1, 9
    - input ports ................................................. 68, 70, 73, 84, 136, 138
    - input stream ............................................... 93, 94
    - inputFileList ............................................... 32
    - inputVars .................................................. 202, 208, 209
    - insertAt() .................................................. 234
    - insertEscapes() ............................................. 279
    - instantiation ............................................. 270
    - instantiation block ...................................... 10
    - InStream functions ....................................... 247, 248
    - integer associations .................................... 27, 314
    - integErrorForm ............................................ 343
    - integration method ....................................... 189
    - integrationType .......................................... 200, 343, 348
    - integrators ................................................. 187, 328, 345
    - input attributes ........................................... 200, 348
    - locking attributes ....................................... 348
    - output attributes ........................................ 200, 349
    - interactive debug
      - output functions ....................................... 322
    - interactive mode ......................................... 6, 55, 365
    - debugging commands ....................................... 59
    - functions .................................................. 256
    - how to enter .............................................. 56
    - Interoperable Object Reference (IOR) .................. 8
    - interpolation ............................................. 51
    - interpreted Component Load on Demand (ICLOD) ....... 9, 61, 62, 63
    - intValue ................................................... 27, 315
    - invertConstraint () .................................... 305, 309
    - IOstatus ................................................... 17, 18
    - IOstatus values .......................................... 18
    - isa() .......................................................... 228
    - isActive ................................................... 99, 280
    - isEmpty() .................................................. 249
    - isNaN() ...................................................... 253
    - iteration ................................................... 40, 185, 328, 365
    - Iteration
      - before convergence .................................... 332
    - iteration pass ............................................. 323
    - iterationCounter ......................................... 174, 298
    - iterations ................................................ 5

### J

- jacobian .......................................................... 295
provisional warnings..............................55
ProxySpec object
attributes of........................................266
purpose
ESORegistry attribute..........................118
putFile().............................................237

Q
quiescence ........................................187
quiescence ().....................................341
quiescenceTolerance............................197, 344
quit
interactive debugging command..............60
quit()................................................62, 254

R
randomInt([int lowerBound,] int upperBound)......253
randomReal([real lowerBound, real upperBound, int closed])
................................................253
randomSeed([int seed])..........................253
range of values
assigned to array...................................24
regenNewJacobian................................296
remove()..........................................234
removeConstraint ().........................309
removeConstraint ().........................305
removeDependent ()..........................291
removeDependent ()..........................91, 347
removeDerivative().............................206
removeDSV ().....................................291
removeEscapes().................................270
removeFile()......................................238
removeIndependent ().......................291
removeIndependent().........................91
removeInput()...................................203, 206
removeOutput().................................203, 206
removeState()...................................203, 206
repeatabilityCheck.............................208
repeatTolerance..................................208
reportFileName................................202, 204, 208
resetConstraints()............................291
resetConvTable()................................218
resolveBounce...................................316
resolveMinMaxConflict.......................176, 296
restoreIndependents()......................259
return type......................................42
revertToDefaults()............................206, 236, 237, 238, 246, 291, 305
right
  text block attribute..........................282
right-hand side
  of independent variable’s equation.........162
run().............................................236, 291, 318, 341
runDiscreteCalcs()...........................361
runInitialPass()...............................209
runSequence macro............................259
runTimeDiscretes()...........................341

S
saveIndependents()..............................259

scalar variable types.............................16
scalars ............................................21
scaleFactor .......................................299
scientific
  output stream attribute......................289
  scope...........................................366
  scope of object
  and indirect membership .....................45
second-order interpolation ..................52
semi-colon
  as statement terminator .....................11
  use within a block.........................11
send()..........................................238
sendFile().......................................237, 238
sequence
  solver sequence where exception occurred....123
  solver sequence where exception occurred....235
sequential
  DSV input attribute........................316
ServerSpec object
attributes of....................................265
setAccess ()....................................256
setAlias().......................................254
setBias().........................................206
setBleedStation()............................235
setDefaultAccess ()..........................256
setDryAir()......................................239
setExecutive().................................235
setFunction
  attribute of FunctVariable...................28
  setLimitType ()................................305
  setMember()....................................234
  setOption().................................28, 45, 141, 190, 228
  setPerturb().................................202, 207
  setPerturbType()............................207
  setPriority ().................................305
  setSlope ().....................................305
  setStatic_u_rho()............................240
  setThermoPackage()..........................65
  setTotal_hP().................................240
  setTotal_hS().................................241
  setTotal_u_rho()..............................246
  setTotalSP()..................................242
  setTotalStaticTP()...........................244
  setTotalTP()...................................244
  setup()........................................342
  setupExecutive().........................220, 235
  setVal()........................................254
  setValueInParent()..........................251
severity
  ESORegistry attribute.......................118
Shaft Elements ................................136
  shaft ports...................................5, 138, 366
show
  interactive debugging command............60
  showColHeader().............................284
  showErrors..................................281
  showHeaders..................................281
  showHidden...................................227
  showMarks.................................282
  showpoint
output stream attribute ........................................... 289
show ProvError ..................................................... 121
show ProvWarnings ................................................ 121
Simple Data Types ................................................ 20
SimpleBlock ....................................................... 109, 283
simultaneous DSV solution ...................................... 311
single pass .......................................................... 320
singularityTest ...................................................... 296
singularRowColTolerance ........................................ 297
sink part of bleed macro .......................................... 258
sockets ............................................................... 2, 4, 75, 77
attributes ................................................................ 76
definition of .......................................................... 366
functions ................................................................... 249
solutionMode ......................................................... 33, 142, 173, 189, 197, 297, 301
solver ................................................................. 2, 5, 9, 55, 67, 80
definition of .......................................................... 329, 366
attributes affecting constraints .................................. 176
commands/member functions .................................... 290
creating within assembly ......................................... 290
definition of .......................................................... 329, 366
input attributes ....................................................... 172, 292
key terms ................................................................... 327
output attributes ..................................................... 173, 297
searching diagnostic output ....................................... 321
setup ....................................................................... 87
structure .................................................................... 165
troubleshooting ....................................................... 326
solver diagnostic reference ....................................... 326
solver macro .......................................................... 259
solver setup automatic .............................................. 90
SolverExecutive class ............................................. 290
solverSequence ...................................................... 33, 81, 217
sourceStr = sourceString: ........................................ 275
Splitter element ..................................................... 130
string variables ....................................................... 25
ssh/scp support functions ........................................ 264
state ................................................................. 200, 311
state value ............................................................. 311
state variables ....................................................... 349
stateDemand ......................................................... 315, 316
stateInitialValue .................................................... 311, 315
statement terminator .............................................. 11, 366
statements ............................................................ 11, 366
stateName ............................................................. 200, 348
stateVars ............................................................. 202, 209
stations ............................................................... 70
definition of .......................................................... 366
steady state112, 126, 136, 142, 173, 183, 185, 187, 188, 329
step interactive debugging command ......................... 60
stepOut interactive debugging command ..................... 60
stepOver interactive debugging command .................... 60
stopFlag .............................................................. 198, 345
stopTime .............................................................. 194, 197, 344
stopTime attribute .................................................. 190
streams .............................................................. 93, 288
common attributes .................................................. 288

input........................................................................ 93
output ...................................................................... 93, 95
string getP1Name() .................................................. 247
string getP2Name() .................................................. 248
string[] getLinkedPortName ...................................... 249
string[] getLinkName() ........................................... 249
study executive file .............................................. 127
subelements .......................................................... 2, 4, 7, 10, 28, 61, 65, 74, 75
definition of .......................................................... 366
inserting into a socket ............................................. 77
substr() ............................................................... 270, 271
subtable ................................................................... 47
subtractEq " .......................................................... 246
subtractNonEq ......................................................... 246
suspendBreak () ..................................................... 59
switchDes ............................................................. 33, 217
synched time-discrete objects .................................... 340
synchTimeDiscretes() ............................................. 342
definition of .......................................................... 366
sys.dirSeparator ..................................................... 260
definition of .......................................................... 121
directories and functions ......................................... 261
definition of .......................................................... 121
sys.etStr(string t1, string t2) .................................... 261
definition of .......................................................... 121
definition of .......................................................... 261
sys.pathSeparator .................................................. 260
definition of .......................................................... 121
definition of .......................................................... 261
definition of .......................................................... 121
sys.pushErrorResponse() ........................................ 263
definition of .......................................................... 121
definition of .......................................................... 261
definition of .......................................................... 121
system(string shellCommand) ................................ 253
definition of .......................................................... 121
System.fnc file attributes and functions in ................. 267
definition of .......................................................... 121
definition of .......................................................... 121
definition of .......................................................... 121
definition of .......................................................... 121
T

table ...................................................................... 249, 250
table extrapolation
warning attributes .................................................. 55
tables ................................................................. 2, 4, 46, 48
definition of .......................................................... 366
definition of .......................................................... 366
inserting into sockets ............................................. 78
definition of .......................................................... 366
interpolation & extrapolation attributes ................. 51
definition of .......................................................... 366
inverse lookup ....................................................... 49
minimum definition ............................................... 48
one dimensional .................................................... 46
scaling independent values ..................................... 49
scaling return values ............................................. 49
stacked ............................................................... 50
three-dimensional ................................................ 47
target dependents .................................................. 175, 176, 178, 304
terminateCondition .............................................. 197, 344
testRealValid ....................................................... 307
testRunTermination () ............................................. 341
testXConvergence ............................................... 297
Text block ........................................................... 104
TextBlock ............................................................ 282
Therm ............................................................... 65
thermodynamics package ..................................... 2, 14, 34, 62, 65, 127
time transient input attribute .................................. 344
time discrete model objects functions ...................... 360
time step ............................................................. 186, 190
time stepping adaptive .......................................... 187, 192, 193
user defined .......................................................... 193
updateConverged .................................................. 299
time update loop.......................................................... 331
timeConstantExpr ...................................................... 348
time-discrete objects .......................................................... 190
time-increment computing .................................................. 331
time-sequence steps .......................................................... 334
timeStep .......................................................... 345
timeStepCounter .......................................................... 198
timeStepMethod ...................................................... 192, 193, 197, 344
title
dynamic text block variable ........................................ 283
viewer attribute .......................................................... 282
titleBody .......................................................... 282, 283
titleVars .......................................................... 282, 283
toInt() ........................................................................ 271
Tokenizer Class .......................................................... 274
tolerance .......................................................... 162, 169, 307, 329
absolute .......................................................... 162
fractional .......................................................... 162
toleranceRMS .......................................................... 299
toleranceScaleFactor ...................................................... 344
toleranceType .......................................................... 169, 307
toReal() ........................................................................ 272
toStr() ........................................................................ 272
-trace ........................................................................ 7
trace [on/off] .......................................................... 60
trace output .......................................................... 7
traceExecution(int onOff) .................................................. 254
transient .......................................................... 142, 330
attribute of timeStep ...................................................... 345
calculating conditions ...................................................... 184
definition of .......................................................... 329
driving input attributes ...................................................... 342
output attributes .......................................................... 344
running terminating ...................................................... 190, 194
Transient Executive utility functions .................................................. 340
transient simulations ...................................................... 183
TransientExecutive ...................................................... 185, 186
attributes .......................................................... 196
functions .......................................................... 341
-tree() ........................................................................ 236
trigger .......................................................... 17, 19
Turbine element .......................................................... 135
updateDSVs() ............................................................. 318
updateErrors() ............................................................. 318
updateHistory() ............................................................. 361
updateIndependents() ...................................................... 318
useConstraints ...................................................... 176, 307
userReport() ............................................................. 323, 325
UserTextBlock .......................................................... 107, 286
userTimeStepFunction ...................................................... 197, 344
useSolvSeqOrder .......................................................... 284
-v, command line option ...................................................... 7
value
variable attribute .......................................................... 17
VarDumpViewer ...................................................... 98, 99, 236, 280, 286
variable scope .......................................................... 20
variable types .......................................................... 15
variableList .......................................................... 99, 102, 282, 286
variables .......................................................... 4, 15
definition of .......................................................... 366
naming .......................................................... 19
varName .......................................................... 167, 301
varNamesActiveIndep () .................................................. 291
varNamesActiveIndep() .................................................. 229
vars
simple block attribute ...................................................... 283
verify() ........................................................................ 229
version number .......................................................... 67, 75
void debug(int flag) ...................................................... 256
void insertByName() ...................................................... 234
void replaceByName() ...................................................... 234
void revertToDefaults() .................................................. 300
warnStreamName .......................................................... 121
where
interactive debugging command ........................................ 60
while loop .......................................................... 41
width
empty text block variable ...................................................... 286
output stream attribute ...................................................... 289
X
xFirst .......................................................... 303
xLimitCheck .......................................................... 301
xLimitExceeded .......................................................... 301
xLimitReport .......................................................... 301
XMappingFunction ...................................................... 302
xMaxLimit .......................................................... 302
xMaxLimitExp .......................................................... 302
xMaxLimitFail .......................................................... 302
xMaxLimitWarn .......................................................... 302
xMinLimit .......................................................... 302
xMinLimitFail .......................................................... 302
xMode .......................................................... 302
xModelFirst .......................................................... 303
xPrevious .......................................................... 303
xRef .......................................................... 303
xTolerance .......................................................... 302
xToleranceType ................................................................. 302

yRef .............................................................. 171, 309