National Aeronautics and Space Administration

NPSS Developer’s Guide

Software Release: NPSS_1.6.5

Doc. #: NPSS–Developer
Doc Revision: R
Revision Date: March 12, 2008

Numerical Propulsion System Simulation
NASA Industry Cooperative Effort
(NPSS/NICE)

NASA John H. Glenn Research Center at Lewis Field
2100 Brookpark Rd.
Cleveland, Ohio 44135-3191

Copyright 1997, 2001, 2002. The United States Government, as Represented by the Administrator of the National Aeronautics and Space Administration (NASA). All rights reserved.

Controlled Distribution.
Further distribution requires written approval of the NASA Glenn Research Center
Cleveland, Ohio.
# Table of Contents

PREFACE .......................................................................................................................... VI

1 INTRODUCTION ........................................................................................................ 1-1

1.1 Components within the NPSS ............................................................................. 1-1

1.2 Four Mechanisms for Adding Components ...................................................... 1-1

2 INTERPRETED COMPONENTS ........................................................................ 2-1

2.1 Reasons for Using an Interpreted Component .................................................. 2-1

2.2 Objects to Aid Development of Interpreted Components .................................. 2-1

2.2.1 Variable Types ............................................................................................... 2-1

2.2.2 Port Objects .................................................................................................... 2-2

2.2.3 Station Objects .............................................................................................. 2-3

2.2.4 Solver Objects ............................................................................................... 2-3

2.2.5 Sockets ............................................................................................................ 2-6

2.2.6 Subelements ................................................................................................... 2-6

2.2.7 Tables ............................................................................................................. 2-6

2.2.8 Reference Stations ....................................................................................... 2-7

2.2.9 Transient Histories ...................................................................................... 2-7

2.2.10 Interpreted User Defined Class Types ....................................................... 2-8

2.3 Interpreted Component Functions ..................................................................... 2-9

2.3.1 Required Functions ....................................................................................... 2-9

2.3.2 Optional Functions ....................................................................................... 2-9

2.4 Error Handling ..................................................................................................... 2-11

2.4.1 Provisional Errors and Warnings .................................................................. 2-11

2.4.2 Exceptions ...................................................................................................... 2-11

2.4.3 Errors Found During verify() ....................................................................... 2-11

2.5 Units ..................................................................................................................... 2-11

2.6 How to Add an Interpreted Element to NPSS ................................................... 2-11

2.6.1 Example of Adding an Interpreted Duct Element ......................................... 2-12

2.6.2 How to Inherit From an Existing Element Type ........................................... 2-12

2.7 How to Add an Interpreted Subelement to NPSS ............................................. 2-13

2.7.1 Example of an Interpreted Aerodynamic Pressure Loss Subelement ............ 2-13

2.8 Wrapping External Codes with Interpreted Elements ...................................... 2-14

2.8.1 Existing Wrapper Classes ............................................................................ 2-14

2.8.2 Classes for Batch Jobs ................................................................................ 2-16

2.8.3 Batch Job Example ..................................................................................... 2-20

2.8.4 Namelist ........................................................................................................ 2-21

2.8.5 Map Generation Classes ............................................................................... 2-23
2.8.6 Map Generation Example.................................................................2-25

3 INTERNAL AND DLM COMPONENTS...........................................3-1

3.1 Pros and Cons .....................................................................................3-1

3.2 Objects to Aid Development of Compiled Components..................3-1

3.2.1 Generic NPSS Values ......................................................................3-1

3.2.2 Variable Types..................................................................................3-2

3.2.3 Port Objects ....................................................................................3-3

3.2.4 Station Objects ................................................................................3-3

3.2.5 Transient Histories .........................................................................3-3

3.2.6 Solver Objects ................................................................................3-5

3.2.7 Sockets ............................................................................................3-9

3.2.8 Function Objects ............................................................................3-10

3.2.9 Tables .............................................................................................3-11

3.3 Component Functions .......................................................................3-12

3.3.1 Required Functions .........................................................................3-12

3.3.2 Required Macros.............................................................................3-12

3.3.3 Optional Functions ..........................................................................3-13

3.3.4 Convenience Functions ....................................................................3-13

3.4 Error Handling ..................................................................................3-14

3.4.1 Provisional Errors and Warnings .....................................................3-14

3.4.2 Exceptions ......................................................................................3-14

3.4.3 Errors Found During verify() ........................................................3-14

3.5 How to Add a Compiled Element to NPSS ........................................3-14

3.5.1 The Makefile System (UNIX) ..........................................................3-15

3.5.2 The Makefile System (Windows) .....................................................3-17

3.5.3 Example of Adding a Compiled Duct Element ................................3-17

3.6 How to Add a Compiled Subelement to NPSS ...............................3-20

3.6.1 Example of Adding a Compiled Aerodynamic Pressure Loss Subelement ....3-20

3.7 Inheritance of Internal Elements and Subelements ..........................3-24

4 DLM DEVELOPMENT KIT.................................................................4-1

4.1 Overview of the DLM Development Kit ..........................................4-1

4.1.1 What the DLM Dev Kit Can Do ......................................................4-1

4.1.2 Installing the DLM Development Kit ............................................4-2

4.2 Creating and Running the Provided Example Builds ......................4-5

4.2.1 Component Source Tree and Building Overview .........................4-5

4.2.2 Compiling a Component Source Tree as a DLM (UNIX) ...............4-7

4.2.3 Compiling a Component Source Tree as a DLM (Windows) ..........4-8

4.2.4 Examining the Provided Examples ...............................................4-9
Preface

Who Will Use this Manual

The Numerical Propulsion System Simulation is a framework for executing systems of components. Out of the box, NPSS comes with a set of built-in components designed to model aeropropulsion systems. The NPSS framework was designed so that adding new types of components is easy. After reading this guide, you will be able to add your own custom components to NPSS (interpreted, internal or DLM) and use them to model specialized engine components or other types of dynamic components unrelated to aeropropulsion system simulation. Or, you can use the Dev Kit information to build an NPSS executable, a DLM for use with NPSS, or a standalone executable to use with specific models (customer decks).

This document assumes that you are already familiar with how NPSS works. You should understand the basic objects in an NPSS model and how they interact. For example, you should know that assemblies contain elements and that elements contain subelements. (In general elements can be defined along component lines.) You should also know that ports connect elements. If these concepts are not familiar to you, please consult the User Guide and Reference before proceeding further.

If you plan to add compiled components, you must be familiar with the C++ programming language.

Other Applicable Documents

- For information on using the CCDK (CORBA Component Development Kit) to create external elements, please refer to the CORBA Component Developer’s Guide. That guide is available to users who are permitted access to the CCDK.Base tar file.

- For users who are permitted access to the CCDK.HiFi tar file, a CCDK High Fidelity Supplement is available.

- For users who are permitted access to the CCDK.RocketsITAR tar file, a short supplement is available. (It currently only includes information on the PUMPA code included in the tar file.)

Whenever a formal or incremental NPSS package is released, one of the above relevant documents is included in the appropriate tar file.

Overview of Sections

The Preface contains general information on who will use the manual, what information is contained in the document, style conventions, and getting help.

Chapter 1, “Introduction,” briefly introduces the methods for adding components.

Chapter 2, “Interpreted Components,” explains how to add an interpreted component to NPSS using the input syntax and discusses reasons why you might want to use this method.

Chapter 3, “Internal and DLM Components,” discusses objects to aid development of compiled components and discusses how to add a compiled element or subelement to NPSS. It also includes sections on component functions and error handling.
Chapter 4, “The DLM Development Kit” provides an overview of the DLM Dev Kit, and includes information on running example builds and the layout of the source tree.

Chapter 5, “NPSS Model Delivery Development Kit,” discusses the security model and building a customer deck using the Native NPSS API or the FORTRAN API.

Chapter 6, “Testing an Internal Component,” contains general information on how to test components.

Chapter 7, “Utilities,” has information on the NPSS to C++ Converter, the Vcopy Utility, and the DLMgenUtility (for UNIX).

Font Conventions/Symbols

As you read this manual, please be aware of how typefaces are used. See the following table:

<table>
<thead>
<tr>
<th>Font</th>
<th>Used For</th>
</tr>
</thead>
<tbody>
<tr>
<td>Times Roman</td>
<td>Normal text</td>
</tr>
<tr>
<td>Courier New</td>
<td>Code examples, variable names, string values, path names</td>
</tr>
<tr>
<td>Bold</td>
<td>Literal values that users enter</td>
</tr>
<tr>
<td></td>
<td>Text headings</td>
</tr>
<tr>
<td></td>
<td>Emphasis</td>
</tr>
<tr>
<td>Italic</td>
<td>Titles of documents</td>
</tr>
<tr>
<td></td>
<td>Emphasis</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.
- hx 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:

```python
solver_name.addIndependent("independent_name");
```

In real use, the user would supply specific names for the items in italics, such as:

```python
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 ClearTrack 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 NASA. You may also receive help via email. Please see the following section.

Getting Help
If you have a problem or question that requires assistance, you can send an e-mail to the following address:

ncphelp@grc.nasa.gov

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.

**Document Revisions**

Please note that there is a Revision Page at the end of this document, just before the Index. The revision page lists those CRs (change requests) that went into a particular release (increment or formal build) IF the CR resulted in a change to this document. The build (e.g., Rev: Z) and date the document changes were made are also noted. If significant changes were made to a section, section numbers are provided. When many minor changes (one or two words) are made throughout the document, these are generally not listed. For a list of all CRs that went into a particular NPSS release, please see the Release Notes for that package.
1 Introduction

1.1 Components within the NPSS

Components are the computational building blocks of an NPSS model. The three primary types of NPSS components that you should be aware of are:

**Element**
A basic computational component with ports that can connect to other Elements.

**Assembly**
A special kind of Element that can contain many Elements and a Solver to iterate over those Elements.

**Subelement**
A basic computational component that provides some detailed computation inside of its parent Element or Subelement.

This manual will show you how to create a new type of Element or Subelement and add it to NPSS. Other users will then be able to use your new type to create components that are connected to other NPSS components to form an NPSS model.

1.2 Four Mechanisms for Adding Components

The NPSS framework provides four mechanisms to add new components: *internal components, DLM components, interpreted components*, and *external components*. When a user attempts to create an instance of a component, NPSS searches for that component type in the same order as listed above.

To develop *internal components*, you write C++ code and compile it into the NPSS executable. These components are typically only added by NPSS infrastructure developers.

The simplest way to add a component is to write it using NPSS input syntax. Component types defined this way are used to create *interpreted components*. Interpreted components are roughly twice as slow as their internal or DLM counterparts.

To add a *DLM component*, you can either write your component directly in C++, or you can take an *interpreted component* and run it through the NPSS component converter. DLM components, like *internal components*, are compiled, but they are dynamically loaded by the NPSS executable at run time. *DLM components* based on either interpreted code or C++ can be built using the DLMdevkit.

You can add a component type in a fourth way: by writing special code that encloses your existing component code and then using the Common Object Request Broker Architecture (CORBA) to register it for access at run time. If the system cannot find a component type match from the internal, DLM, or interpreted sets, NPSS checks for a local Object Request Broker (ORB) to see if a CORBA registered component type exists outside of the NPSS. If the component type does exist, NPSS creates a new instance of the component. A component added in this way is an *external component*. For coding external components, NPSS provides a CORBA Component Development Kit (CCDK).
2 Interpreted Components

2.1 Reasons for Using an Interpreted Component

Interpreted components can be added to NPSS quickly and easily. If you have a simple element you want to add, or you want to test an idea (such as a new performance calculation) before creating a compiled element, this is the easiest way to do it. The only real problem with using an interpreted component vs. using an internal or DLM component is that the interpreted version will run more slowly.

2.2 Objects to Aid Development of Interpreted Components

The object types described in the following sections may be declared as members of an interpreted component and may be used inside of component functions.

2.2.1 Variable Types

Interpreted components support all of the following NPSS variable types.

- `any a;`  
  A variable that can be assigned to a variable of any other type (real, real[], int, string, etc.).

- `real x;`  
  A variable with a single floating point value.

- `real x[];`  
  A variable with an array of floating point values.

- `real y[][];`  
  A variable with a 2-dimensional array of floating point values.

- `real z[][][];`  
  A variable with a 3-dimensional array of floating point values.

- `realScalarAdder`  
  The class `realScalarAdder` is an extension of the type `real`. This class contains the attributes `scalar`, `adder`, `base`, and a Boolean that determines what value to return. All attributes can be displayed. This class inherits from `VariableBase`. Any value assigned to a variable of this type gets the scalar multiplied to base value; then the adder is applied. When the Boolean is set to TRUE or nonzero, the class returns the value with the scalar and adder applied. When the Boolean is set to FALSE or zero, the class returns just the base value. When assigning one `realScalarAdder` variable to another, only the value is copied, not the scalar, adder, base value, and Boolean. A warning message will be issued when the scalar or adder is set and the Boolean is FALSE or nonzero.

  **Attribute names:**

  - `s`  
    scalar value, Type NCPReal, Default value =1.0
  - `a`  
    adder value, Type NCPReal, Default value =0.0
  - `apply`  
    Boolean used to determine what value to return, Type NCPInteger; Default value = TRUE or nonzero
  - `b`  
    value without the scalar and adder applied, Type NCPReal (get only); Default value =0.0

- `int i;`  
  A variable with a single integer value.
int i[];
A variable with an array of integer values.

int j[][];
A variable with a 2-dimensional array of integer values.

string s;
A string variable.

string s[];
A variable with an array of string values.

Option
A variable with only a finite number of allowed string values. Options are declared as follows:

```
Option myOption {
    allowedValues = { "myValue1", "myValue2" }
    value = "myValue2";
    trigger = FALSE; // this Option will not call variableChanged()
}
```

2.2.1.1 A Note On Variable Triggers
All variable types in NPSS have a trigger attribute. If this attribute is TRUE, they will call variableChanged() on their parent object whenever their value is set. In Option variables, the trigger attribute defaults to TRUE, while in all other types it defaults to FALSE.

2.2.2 Port Objects
Note that ports should only be created within an element, not a subelement. To declare a port within your interpreted element, use the following syntax:

```
<PortType> <portName> { ... }
```

For example,

```
FluidInputPort Fl_I {
    // set initial values here . . .
}
```

The following port types are available to the interpreted element developer. For a list of attributes for each port type, see the reference section of the User Guide and Reference.

FluidInputPort
FluidOutputPort
NewStreamPort
FuelInputPort
FuelOutputPort
ShaftInputPort
ShaftOutputPort
ThermalInputPort
ThermalOutputPort
DataInputPort
DataOutputPort
FileInputPort
2.2.3 Station Objects

FlowStation
If you choose to use the built-in NPSS thermodynamics routines in your component calculations, you will need to create one or more FlowStation objects or fluid ports. FlowStation objects have member functions that perform thermodynamic calculations. The available flow station functions depend on the property package used and are defined in the Reference Sheets document under the appropriate flow station type.

2.2.4 Solver Objects

2.2.4.1 Using the System Solver
For interpreted components, solver objects are defined, added and removed in exactly the same manner as they are in a Model (refer to the NPSS User Guide and Reference for more information about the various solver object types). As in the Model, solver terms must be defined first and then added to the Solver before they can be used.

When a solver term is defined inside an Element, it is placed on an internal list. Items on this list can, in turn, be added to the Solver when the user calls the autoSolverSetup function (as described in the NPSS User Guide). Each object has an attribute called autoSetup which determines whether the autoSolverSetup function will add it to the solver. If autoSetup is set to TRUE, the object will be added. If autoSetup is set to FALSE, the object will not be added. The default value for this attribute is FALSE. The user must set this attribute to TRUE before calling autoSolverSetup in order for the object to be added to the solver.

```java
Class Volume extends Element {
    real pressure, density, dRH0dt;
    real flowIn, flowOut;
    real volSize;
    //etc.

    Independent pressIndep {
        varName = "pressure";
        autoSetup = TRUE;
    }

    Integrator densInteg {
        stateName = "density";
        derivativeName = "dRH0dt";
        eq_rhs = "flowOut";
        eq_lhs = "flowIn";
        autoSetup = TRUE;
    }

    void calculate() {
        // connect internal flow variables to ports
        // . . .
        // compute derivative
        dRH0dt = (flowIn - flowOut)/volSize;
    }
}

Element Volume myVol { volSize = 100.; }
```
list("Independent", 1);  // list the independents defined by the
// Model, including those added to Elements.
List("Dependent", 1);  // list the dependents in Model ...

autoSolverSetup();  // query the Model to get all Solver terms added
// there, and add them to the Solver.

Solver.list("Independent",0);  // list will match that above.
Solver.list("Dependent",0);  // ditto.

The solver object types which can be defined within an Element are:

Dependent
DiscreteStateVariable
Independent
Integrator

Solver terms may be mode-specific. Different solver setups may be required for DESIGN and OFFDESIGN modes for example. It is therefore possible to specify which solver objects are added to the solver by the autoSolverSetup function, depending on changes in the mode (or other variables). This is accomplished through setting the object's autoSetup attribute to TRUE and calling autoSolverSetup. Similarly, setting this attribute to FALSE and calling autoSolverSetup will remove those objects from the solver, but not from the Element.

The following will add four objects to the solver:

<dependent_name>.autoSetup = TRUE;
<DSV_name>.autoSetup = TRUE;
<independent_name>.autoSetup = TRUE;
<integrator_name>.autoSetup = TRUE;
autoSolverSetup();

The following will remove four objects from the solver:

<dependent_name>.autoSetup = FALSE;
<DSV_name>.autoSetup = FALSE;
<independent_name>.autoSetup = FALSE;
<integrator_name>.autoSetup = FALSE;
autoSolverSetup();

In the case described above, for example, the density integrator can be added to and removed from the solver by calling the functions as follows.

densInteg.autoSetup = TRUE;  //sets the autoSetup attribute
autoSolverSetup();  //adds all objects with autoSetup == TRUE to solver

densInteg.autoSetup = FALSE;  //sets the autoSetup attribute
autoSolverSetup();  //adds all objects with autoSetup == TRUE to solver
//effectively removes densInteg from solver

Note that when objects are created, the default value of the autoSetup attribute is “FALSE.” In some cases, it may be desirable to remove all solver objects and add back the ones required for a specific mode. This can be accomplished using the clearSolverTerms function.

clearSolverTerms();  //sets autoSetup flags to FALSE
autoSolverSetup();  // adds back objects with autoSetup set to TRUE
// effectively removes all solver terms
NOTE: Calling `clearSolverTerms` merely sets the `autoSetup` flag to `FALSE` for all objects in the solver. The objects will not actually be removed from the Solver until the `autoSolverSetup` function is called.

### 2.2.4.2 Using Component-Internal Solvers

In addition to creating Solver terms that will be controlled by the system solver, the NPSS includes a simple, one-parameter solver that can be embedded within components. This is called a secant solver, (see Secant Solver E-Spec for more information). This object is not suitable for use at the Model level and is therefore not described in the NPSS User Guide and Reference.

The SecantSolver is not a part of the NPSS architecture, but is an object that can be used within Elements and Subelements. The execution logic for the component must specifically define the iteration loop, and use methods on the SecantSolver to predict new independent values for each iteration, test for convergence, etc. Each SecantSolver object handles a single independent and dependent-error. Unlike the system Solver, the SecantSolver attributes are all absolute, not fractional (refer to the Solver E-Spec document for an explanation of absolute and fractional parameters). Loops can be nested in the component logic to make use of multiple SecantSolver objects, if desired.

The necessary steps for using the SecantSolver object are as follows:

1. Create the SecantSolver object.
2. Set the desired attributes such as `maxIters`, `perturbation`, etc. (functions and attribute listed below)
3. Initialize the SecantSolver object.
4. Run component calculations and compute the error value.
5. Set the new value of the independent parameter to the results of the SecantSolver object’s `iterate` function.
6. Repeat steps 4 and 5 while the SecantSolver object’s `isConverged` and `errorFound` functions both return `FALSE`.

In the following example, a SecantSolver is used in an iteration loop to find the area required to produce a certain fluid pressure drop in a pipe.

```java
Class SelfSolvingDuct extends Element {
    real area, dpNorm, dpNormDes, Wflow, Cf, rhoIn;

    // Step 1: Create the PsSecant object
    SecantSolver solveArea {
        // Step 2: Set attributes
        solveArea.setMaxIters(20);
        solveArea.setPerturb(0.005);
        // use defaults otherwise);
    }

    void calculate()
    {
        // Step 3: Initialize PsSecant object
        // set the independent parameters initial (guess) value
        area = 0.001;
        solveArea.initialize(area, this);

        // start iteration loop
        do {
            // Step 4: Run component calculations and compute error value.
            NCPReal error;
            dPqP = pow((Wflow/Cf/area), 2.0)/rhoIn;
            error = (dPqP - dPqPdes)/dPqPdes;

            // Step 5: Set area (independent) to results of iterate
```
// function.
Area = solveArea.iterate(error);

// Step 6: iterate until converged or found error.
} while (!(solveArea.isConverged()) &&
!(solveArea.errorType));

} // end of calculate function.
} // end of self-solving duct element.

A list of the attributes and functions for the SecantSolver object is given in the table contained in section 0 of this guide.

2.2.5 Sockets
Sockets are placeholders for subelements or functions. Each Socket has a list of variable names of variables that its child subelement is allowed to set in the Socket’s parent. Sockets are declared as follows:

Socket S_mySubE {
  allowedValues = { “x”, “y”, “z” }
}

When a new object is added to its parent, if that parent contains a Socket having the same name as the new object, the new object will be automatically inserted into the Socket and renamed to child. Objects can also be placed in a Socket by using the move() function.

2.2.6 Subelements
Normally, Subelements are declared inside of an instance of an NPSS component rather than in the class declaration itself, but sometimes it is useful to do this so that every instance of that component type will contain that subelement. To make a subelement a member of every instance of your new component, simply declare it within the scope of the class declaration, for example:

class MyElement extends Element {
  Subelement dPqP S_mySubE { ... }
}

Now, every instance of type MyElement will contain a dPqP subelement called mySubE.

2.2.7 Tables
Tables are added to interpreted component types or to instances of interpreted components in the same way that they are added to instances of built-in components. The table is simply declared within the scope of the desired component or component type. Once declared, the table may be evaluated by calling the table like a function. For example:

class MyElement extends Element {
  real a, b, c;

  // In this case, declare the Table as part of the new type, so that every
  // instance of the type will contain the table
  Table TB_mytab(real x) {
    x = { 1.0, 2.0, 3.0 }
    y = { 2.0, 4.0, 6.0 }
  }

  // now write my calculate function so that I use the Table mytab
}

Interpreted Components
Often you won’t declare a Table as part of a class, but instead will depend on the model builder to add a Table to an instance of your component type that the person is using in his model. In this case, you may want to check to see if the Table exists before you call it in your calculate() function. You could rewrite the calculate() function above to do this as follows:

```cpp
void calculate() {
    if (exists("TB_mytab")) {
        a = TB_mytab(b);
    } else {
        // perform some other calculation
    }
}
```

You don’t need to add code to generate an error message if the table doesn’t exist, since this will be generated automatically if NPSS can’t find the table you’re attempting to evaluate. Using the exists() function is only necessary when you want to perform some alternate calculation when the table or function doesn’t exist.

### 2.2.8 Reference Stations

String inputs on Elements and Subelements can now be used to reference flowstations in other parts of the model. The string is declared in the input section of the element/subelement. Then, in the calculate function, a copyFlow or copyFlowStatic command is done to grab the information from the port and place it in a temporary station location for use by the element. When this is done, the user can input a string to an element/subelement that references a station elsewhere in the engine and use its value in the calculate function.

```cpp
Class Example extends Element{
    string temp1;
    FlowStation Fl_It;

    void calculate() {
        // make sure temp1 is pointing to something!
        Temp1 = "otherElement.Fl_I";
        Fl_It.copyFlow(temp1);
    }
}
```

### 2.2.9 Transient Histories

The TransHistory class allows you to store and retrieve time-dependent history data for variables in your component. This is often useful if your component performs any internal integrations or other such calculations which depend on past parameter values. To create an automatic link between a component variable and a history object, you must set the varName attribute. This can be done, for example, as part of the TransHistory object definition

```cpp
TransHistory tempHistory {
    varName = "Fl_I.Ts";
    size = 10; // keep 10 past data points
}
```

It is also possible to define your own internal time variable (different from the system Model time) and connect it to the history by setting the localTimeName.

```cpp
tempHistory.localTimeName = "timeLocal";
```
This may be useful, for example, if you wish to perform several internal time integrations for each time-step at the system level.

The TransHistory object also provides a wide range of functions for accessing historical data, which are described in section 3.2.5 of this document. (Although TransHistory is a complex data type, it is not listed in the NPSS User Guide because the basic user does not encounter it.)

### 2.2.10 Interpreted User Defined Class Types

Instances of User Defined Classes may be created and used within interpreted code subject to the following restrictions.

Class instance must be declared PRIOR to the user class functions, for example:

```c
class USING_Class extends Subelement {
  ...
  UserClass_X  UC_01 { UC_data1 = 5; UC_data2 = 20.0; } // w optional data IC
  ...
  void calculate() {
    ...
    UserClass_X  UC_02  // This declaration will be ignored.
    ...
  }
}
```

Attempts to create instances inside the `calculate()` or other functions will cause the converter to completely ignore the class declaration attempt.

2) Single class data element initializations work fine in the functions as do User Class function executions, for example:

```c
void calculate() {
  ...
  UC_01.UC_data1 = data_load_variable;
  UC_01.initialize();
  UC_01.anotherUserClass_X_function();
  ...
}
```

3) The converter can resolve a `UserClass_X` type which is at least one level extended from a new `UserBaseClass_X`.

   (Note that the base class must be converted first if the User classes are interpreted.)

   ```c
   class UserClass_X extends UserBaseClass_X
   {  // definition of UserClass_X would follow
   }
   ```

4) Attempted Compound class Data Element initializations in functions, for example,

```c
void calculate() {
  ...
  UC_01 { UC_data1=0; UC_data2=1.e-6; maxIters=25.; } 
  ...
```

will ignore the class name and erroneously initialize 3 new variables NOT connected to the class reference.
2.3 Interpreted Component Functions

2.3.1 Required Functions

```cpp
void calculate()
```
The `calculate()` function should retrieve data from input ports, execute subelements, perform any necessary calculations, and write data to output ports.

2.3.2 Optional Functions

```cpp
real nextExecTime(real proposedTime)
```
If your component is a time discrete transient component that must run at specific times, you can override this function to return the next time that you want the entire Model to execute.

```cpp
void initializeHistory()
```
If your component performs special initialization functions at the beginning of transient runs, you can override this function. This function is automatically called the first time a transient run is executed in a batch file (not at the beginning of every transient run in the batch). It can also be called by the user at the top level and will propagate throughout the Model. If this function is not overloaded, the default action when called is to send the initializeHistory function to all TransHistory objects and Subelements in the component. If you override the function for any reason, you must send the initializeHistory signal to these objects in your override (it is no longer automatic). An example of how this is done is shown below (see section 2.2.9 for more information about TransHistory objects).

```cpp
void initializeHistory() {
    // perform any special calculations or operations required
    // by the object before transient run starts.
    ...
    ...
    // send the initializeHistory message to internal objects
    // which may keep their own time-histories.
    initChildHistories();
}
```

```cpp
void postcreate (string name)
```
This function is called when an object is created in a component; `name` is the name of the newly created object. This function may be useful when a component wants to keep track of certain items created inside of it, as in the following example, where the Fluid Input and Output port type components are tracked.

```cpp
void postcreate( string name )
{
    // If an input port was created, add it to the list
    if (name->hasInterface("FluidInputPort")) {
        InputPorts.append( name );
    }
    // If an output port was created, add it to the list
    else if ( name ->hasInterface("FluidOutputPort" )) {
        OutputPorts.append ( name );
    }
}
```

```cpp
void postconvergence()
```

2.3 Interpreted Component Functions

2.3.1 Required Functions

```cpp
void calculate()
```
The `calculate()` function should retrieve data from input ports, execute subelements, perform any necessary calculations, and write data to output ports.

2.3.2 Optional Functions

```cpp
real nextExecTime(real proposedTime)
```
If your component is a time discrete transient component that must run at specific times, you can override this function to return the next time that you want the entire Model to execute.

```cpp
void initializeHistory()
```
If your component performs special initialization functions at the beginning of transient runs, you can override this function. This function is automatically called the first time a transient run is executed in a batch file (not at the beginning of every transient run in the batch). It can also be called by the user at the top level and will propagate throughout the Model. If this function is not overloaded, the default action when called is to send the initializeHistory function to all TransHistory objects and Subelements in the component. If you override the function for any reason, you must send the initializeHistory signal to these objects in your override (it is no longer automatic). An example of how this is done is shown below (see section 2.2.9 for more information about TransHistory objects).

```cpp
void initializeHistory() {
    // perform any special calculations or operations required
    // by the object before transient run starts.
    ...
    ...
    // send the initializeHistory message to internal objects
    // which may keep their own time-histories.
    initChildHistories();
}
```

```cpp
void postcreate (string name)
```
This function is called when an object is created in a component; `name` is the name of the newly created object. This function may be useful when a component wants to keep track of certain items created inside of it, as in the following example, where the Fluid Input and Output port type components are tracked.

```cpp
void postcreate( string name )
{
    // If an input port was created, add it to the list
    if (name->hasInterface("FluidInputPort")) {
        InputPorts.append( name );
    }
    // If an output port was created, add it to the list
    else if ( name ->hasInterface("FluidOutputPort" )) {
        OutputPorts.append ( name );
    }
}
```

```cpp
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.

```cpp
int runDiscreteCalcs(real startTime, real endTime)
```

If your component is a time-discrete transient component that must run only once per time-step, you can override this function. It will be called only during transient simulations and will be executed before the predictor step (unless adaptive time-stepping is enabled). The start and end times for the current time-step are passed into the function (the startTime is the previous time the Model executed and the endTime is the time at which the Model will execute next). This function should return TRUE if any model inputs have been changed, FALSE otherwise.

```cpp
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.

```cpp
void updateHistory()
```

If your component performs special operations at the end of converged time-step (in a transient simulation), you can override this function. It is called after the solver converges at each time-step of a transient run. If this function is not overloaded, the default action when called is to send the updateHistory function to all TransHistory objects and Subelements in the component. If you override the function for any reason, you must send the updateChildHistories() signal to these objects in your override (see the example for initializeHistory above).

```cpp
void variableChanged(string varName, any oldValue)
```

This is called when an Option variable, or any other variable with its trigger attribute set to TRUE, is set.

```cpp
void VCinit()
```

This function is called upon instantiation of your component. It can be used in a similar manner as a C++ constructor: to initialize variables or execute initialization code. Define this function if you have any one-time initialization needs.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.
2.4 Error Handling
There are a number of functions available for reporting error conditions. The preferred ones are discussed in the following sections. It is not necessary to include the name of the object where the error occurred in the error message, because the full pathname of the object will automatically be prepended for you.

2.4.1 Provisional Errors and Warnings
Provisional errors are errors that occur during the solver iteration process before the model has converged to a solution. Provisional errors are collected during the current iteration pass. If that iteration pass results in a converged solution, then the provisional errors become “real” errors and are reported to the user in the form of exceptions. Provisional warnings are similar except that they result in warning messages instead of exceptions. Provisional errors should be reported using the provisionalError(string msg) function, and provisional warnings should be reported using the provisionalWarning(string msg) function.

2.4.2 Exceptions
To generate an exception, call the throwError(string msg, string functName) function. The second argument to throwError() is optional.

2.4.3 Errors Found During verify()
Errors spotted in the verify() function should be displayed using the error(string) function. This will print the error message to the error stream but it won’t throw an exception. The warning(string) function may also be called from within verify() if the problem is not serious enough to be an error.

2.5 Units
Real variables have a string attribute called units that can be set to allow NPSS to perform automatic unit conversions. For example:

```plaintext
real len {
    units = “M”;
}
```

The variable len above has units of “M” (meters). In order to assign len to a value in some other unit system, for example, feet, we could do the following:

```plaintext
len = 3.21 “FT”;
```

NPSS would parse the assignment above and convert the value 3.21 to meters before assigning it to the variable len.

2.6 How to Add an Interpreted Element to NPSS
Using your favorite editor, create a file containing the definition of your new element type. For an example of what this looks like, see the interpreted Duct example that follows. When the definition is complete, save the file as your_element_name.int. Using this naming convention for your element definition file will allow NPSS to locate and load it automatically using ICLOD whenever your element is needed in a model. You can then create instances of your new element inside of your model. You can also add the definition of your new element type directly to your model file, but putting it in a separate file will make it reusable, i.e., you can use it in any model where you need that type of element. If the file is going to become part of the standard NPSS package, the new file would be placed in the /NPSS/dev/AirBreathing/InterpComponents directory where it can be located by ICLOD.
2.6.1 Example of Adding an Interpreted Duct Element

The following example shows you how to add an Element called MyDuct with a subelement called S_dPqP using the input syntax.

```c++
#include <InterpIncludes.ncp>

class MyDuct extends Element
{
    // define the element variables
    real     a_dPqPaud;
    real     dPqP;
    real     dPqPbase;
    real     s_dPqPaud=1;

    // define a fluid input port and a fluid output port
    FluidInputPort Fl_I;
    FluidOutputPort Fl_O;

    // create the socket called S_dPqP
    Socket S_dPqP;

    // define the execute function
    void calculate() {
        if(!S_dPqP.isEmpty()) {
            S_dPqP.run();
        }

        // Apply scalar and adder to the bare socket value
        dPqP = s_dPqPaud * dPqPbase + a_dPqPaud;

        // pass the flow information from the inlet to the outlet
        Fl_O.copyFlow("Fl_I");
        Fl_O.setTotal_hP(Fl_I.ht, (1 - dPqP) * Fl_I.Pt);

        // pass the swirl through
        Fl_O.swirl = Fl_I.swirl;
    }
}
```

2.6.2 How to Inherit From an Existing Element Type

In the previous example, Element was chosen as the base class for our MyDuct class; but suppose there already existed an element that did most of what you needed. You could inherit from that element and create a new class that could use all of the attributes and functions of the base class, adding additional ones as necessary. For example, suppose you wanted a modified version of the existing Duct class. You could create a new class as follows:

```c++
class NewDuct extends Duct // notice we extend Duct, not Element
{
    // add any new attributes we need here...

    void calculate() {
        // add new calculations here...
        Duct::calculate(); // call the base class version of calculate()
        // or here...
    }
}
```
The base class can be either another interpreted class or a built-in or DLM class. However, if both the base class and the new class are interpreted and are to be converted to C++, they must be set up in the make system such that the base class is converted first.

### 2.7 How to Add an Interpreted Subelement to NPSS

Adding an interpreted subelement to NPSS is done in exactly the same way as adding an interpreted element. Inheritance is also handled in the same way. See the aerodynamic pressure loss subelement code in the next section for an example of an interpreted subelement definition.

#### 2.7.1 Example of an Interpreted Aerodynamic Pressure Loss Subelement

The following is the definition of an interpreted subelement of type `MyDPnorm`.

```c++
#include <InterpIncludes.ncp>

class MyDPnorm extends Subelement {
    real a_dPqP;
    real s_dPqP=1.0;
    real a_MN;
    real Closs=1.0;
    real dPqPdes;
    real dPqPmap=1.0;
    real MNmap;
    real RNI=1.0;
    real s_MN=1.0;
    real s_RNI=1.0;

    s_RNI.trigger=TRUE; // call Subelement’s variableChanged when 
                          // this var changes

    Option switchDes {
        allowedValues = { "DESIGN", "OFFDESIGN" }
        value = "DESIGN";
        trigger = FALSE;
    }

    Option switchMatch {
        allowedValues = { "SCALAR", "ADDER" }
        value = "SCALAR";
        trigger = FALSE;
    }

    void calculate() {
        real gams = F1_I.gams;
        real Ps = F1_I.Ps;
        real Pt = F1_I.Pt;
        real Tt = F1_I.Tt;
        real MN = F1_I.MN;

        real Vhead = .5 * gams * Ps * MN * MN;

        real powTt = Tt ** 1.1;

        // Check to see if it is design mode.
        // If so, determine appropriate scalars.
        if ( switchDes == "DESIGN" ) {
```
// Set all adders to zero and scalars to one.
a_dPqP = 0.0;
s_dPqP = 1.0;

// Set the RNI scalar to give no Reynolds.
// scaling at design
s_RNI = powTt / Pt;
RNI = Pt / powTt * s_RNI;
MNmap = s_MN * MN + a_MN;

// Check to see if there is a table to be read.
Closs = TB_Closs ( MNmap, RNI );
if ( switchMatch == "SCALAR" ) { }
int calculateStatus
Returned status code from the command(s) executed by calculate().

string directory
Directory to run in. Often used with sys.pushd() and sys.popd(). By using distinct directories for different instances of a wrapped code, the files used by the multiple instances won't potentially interfere with each other.

string logicals[]
Names of variables created by createLogical().

string preexecCommands[]
Commands to execute before stdCommands. These are intended to be configured by the user.

string stdCommands[]
Standard set of commands to be run. These are intended to be specific to the external code to be run.

string postexecCommands[]
Commands to execute after stdCommands. These are intended to be configured by the user.

string stdin
File for standard input.

string stdout
File for standard output.

string stderr
File for error output. If set to “stdout,” error output is merged with standard output.

int traceLevel
Function trace level, default 0 => off.

void createLogical(string name, string desc)
Creates an Option variable with the given name and description, with allowedValues of “F=0”, “T=1”. This can be useful for reading and writing Fortran logical variables.

void createPlotTable(string name, string xArray1D, string yArray1D)
Creates a table from the given x and y data arrays. This can be useful for plotting results using the VBS table plotting facility.

void fatalError(string msg)
Convenience routine which calls sys.fatalError(msg).

int fileReplace(string filename, string str1, string str2)
Replace all instances of str1 with str2 in filename. Returns TRUE if there were no read or write errors.

real grabRealValue(string line, int index)
Grab real value from line, scan starts at index.

int isDigit(string c)
Determine if the given character is a digit (0-9).

int isLeadingNumeric(string c)
Determine if the given character could be the start of a numeric value (0-9, ., +, -).
int isNumeric(string c)
Determine if the given character could be part of a numeric value (0-9, ., +, -, E, e).

string realStr(real val)
Convert real to string, ensure it looks like a real (contains decimal point or exponent).

string realStrFmt(real val, string fmt)
Convert real to string using fmt, ensure it looks like a real (contains decimal point or exponent).

real sendFile(string localName, string remoteElement, string remoteName)
Send a file from this element to another. Handles the case where remoteElement is an external element (requiring a putFile() operation) or another element in the local model (implying a file copy on the local system).

real recvFile(string remoteElement, string remoteName, string localName)
Receive a file from another element to this one. Handles the case where remoteElement is an external element (requiring a getFile() operation) or another element in the local model (implying a file copy on the local system).

string this()
Convenience routine to return the name of the object.

string toNN(int i)
Format integer as 2 digits, with leading zeros if necessary. Typically used for generating filenames, etc. Superceeded by toStr(int i, string fmt), but available for backwards compatibility.

### 2.8.1.2 SimpleWrapper

Class SimpleWrapper extends WrapperBase with a calculate() method and an attribute to return the status of executing the method:

```cpp
void calculate()
Checks for directory, creates it if necessary. It then runs in sequence preexecCommands, stdCommands, and postexecCommands. If any command returns a non-zero status, that is saved in calculateStatus and calculate() immediately returns.
```

### 2.8.2 Classes for Batch Jobs

The system provides support for running external codes as batch jobs via Element and Subelement classes defined in files residing in the InterpIncludes directory.

#### 2.8.2.1 BatchJob

Class BatchJob extends WrapperBase, providing attributes relevant for batch job scheduling, and a socket to be filled with a BatchJobExec derivative appropriate for the batch system in use. BatchJob::calculate() sets the current directory to the directory attribute, validates the other attributes, and then calls the socket’s calculate() to actually run the batch job. Afterwards the current directory is restored to what it was before calculate() was called.

```cpp
string shell
Shell to interpret commands, either /bin/sh, /bin/csh, or COMSPEC. The default on UNIX hosts is /bin/sh, while the default on Windows is COMSPEC
```

```cpp
string environment[]
Environment variables, in the form of "varName value".
```

```cpp
int cpuCount
```
Number of CPUs to allocate, default 1.

```cpp
int hostCount
```
Number of hosts to spread cpuCount processes across, default 0 => #hosts == #CPUs. Not all batch systems support this attribute.

```cpp
int maxTime
```
Maximum time required (minutes), default 0 => unspecified.

```cpp
int maxMemory
```
Maximum memory required (MB), default 0 => unspecified.

```cpp
string jobName
```
Name for queued job, default null => job name based on element name.

```cpp
string jobDirectory
```
Directory where the job should be run.

```cpp
string queue
```
Name of queue to submit to, default null => unspecified.

```cpp
string project
```
Name of project (for accounting, etc.), default null => unspecified.

```cpp
string stdInputs[]
```
Input files for stdCommands to be copied to job.

```cpp
string stdOutputs[]
```
Output files for stdCommands to be copied from job.

```cpp
inputs[]
```
Input files to be copied to job. Intended for user configuration.

```cpp
string outputs[]
```
Output files to be copied from job. Intended for user configuration.

**Socket S_jobExec**
Fill with a BatchJobExec derivative to perform batch job submission.

```cpp
string scriptBase()
```
Convenience routine for generating script filenames.

```cpp
void use(string jobExecName)
```
Convenience routine to move() a BatchJobExec subelement into the S_jobExec socket. Any previously use()d subelement will be restored to its original name.

### 2.8.2.2 BatchJobExec

Class BatchJobExec extends Subelement, and provides a base class for the various batch-system specific classes described below.

**LocalOutFileStream batchJobOut**
This is the output stream used for generating the script to be submitted.

```cpp
int useRelativePath
```
Some batch systems require an absolute path to be specified for the execution directory. If `useRelativePath` is **TRUE** (the default), then this absolute path is formed by prefixing the `directory` attribute with the current directory’s path relative to $HOME, if the current directory is indeed ‘under’ $HOME.

```java
void fatalError(string msg)
Convenience routine which calls sys.fatalError(msg).
```

```java
void generateScript(string scriptName, int needDirectory)
Generates the “generic” part of the batch script. Classes extending `BatchJobExec` call this after first opening `batchJobOut` as `scriptName` and writing the system-specific header information. After the generic part of the script is written, the stream is closed and the the file `scriptName` is made executable. If `needDirectory` is **TRUE**, then code is generated to move to the appropriate directory.
```

```java
void stageInputs()
Copies input files to the wrapper’s directory.
```

```java
void stageOutputs(string tmp)
Copies stdout (or `tmp` if `stdout` is null and `tmp` is not null) as well as other output files from the wrapper’s directory to the current directory.
```

```java
string this()
Convenience routine to return the name of the object.
```

### 2.8.2.3 GlobusJobExec

Class `GlobusJobExec` extends `BatchJobExec` to use Globus job submission facilities. Note that a Globus job typically will be run on a host remote from the host running NPSS, so various attributes regarding the remote system are required.

```java
string user
User ID on remote Globus host. If null, the local user ID is used.
```

```java
string host
The name of the remote Globus host.
```

```java
int port
Resource manager port on `host`.
```

```java
string service
Resource manager service name; can be used to specify non-default job managers.
```

```java
string subject
Resource manager subject name. Rarely used.
```

```java
int dryrun
Submit, but do not execute the job.
```

```java
Option jobType
The type of job that Globus is submitting (i.e, single, multiple); default is single.
```

```java
int cleanRemoteEnvironment
Remove the remote directory if created by `forceRemoteDir`; default is **FALSE**.
```

```java
int debugGRAM
Displays GRAM job manager log.
```
**string execDir**  
Remote directory for execution.

**int forceRemoteDir**  
Force creation of a remote directory; default is **FALSE**.

**int fullRemoteCopy**  
Copies the contents of the remote directory to the current directory. This can be more efficient than specifying many individual output files to be retrieved. Default is **FALSE**.

### 2.8.2.4 LSFJobExec  
Class **LSFJobExec** extends **BatchJobExec** with support for Platform Computing’s Load Sharing Facility

**string bsubArgs**  
Arguments to be supplied on the `bsub` command line. These should not include arguments related to defined attributes.

### 2.8.2.5 PBSJobExec  
Class **PBSJobExec** extends **BatchJobExec** with support for the Portable Batch System.

**int pollInterval**  
Job status polling interval (seconds), default 5. This is used to detect job completion.

**string qsubArgs**  
Arguments to be supplied on the `qsub` command line. These should not include arguments related to defined attributes.

### 2.8.2.6 RemoteJobExec  
Class **RemoteJobExec** extends **BatchJobExec** to support running jobs on remote machines, much like **GlobusJobExec**, but without requiring Globus. Instead the job is run via `ssh`.

**string user**  
User ID on remote host. If null, the local user ID is used.

**string host**  
The name of the remote host.

**int cleanRemoteEnvironment**  
Remove the remote directory if created by `forceRemoteDir`; default is **FALSE**.

**string execDir**  
Remote directory for execution.

**int forceRemoteDir**  
Force creation of a remote directory; default is **FALSE**.

**int fullRemoteCopy**  
Copies the contents of the remote directory to the current directory. This can be more efficient than specifying many individual output files to be retrieved. Default is **FALSE**.
2.8.2.7 ShellJobExec

Class ShellJobExec extends BatchJobExec to support running the job under a Bourne sub-shell of the NPSS process. This is sometimes useful for debugging before using a real batch system.

`int directOutput`

If `TRUE`, output is routed to the controlling terminal rather than a file. This is useful for debugging, but it does not mimic the behaviour of the above subelements, and output data will not be propagated to any remote client (such as a higher-level NPSS model or VBS).

2.8.3 Batch Job Example

An example of extending BatchJob for a particular external code is shown below. This is one way of wrapping the NCC code, a 3D combustion code using MPI.

```c
class NCCJob extends BatchJob {
    string nccPath {
        description = "Path to NCC executable";
        value = "ncc_solve.x";
    }

    string mpirunPath {
        description = "Path to mpirun executable";
        value = "mpirun";
    }

    void calculate() {
        // Create job description.
        stdCommands = {;
            stdCommands.append(mpirunPath + " -np " + toStr(cpuCount) + " + " + nccPath);

        // Run job.
        BatchJob::calculate();
    }
}
```

The following is an example use of the above element. It runs the batch job using the Globus system. As part of the job, `a.out` is used to generate an input profile. Next NCC is run using MPI. If the NCC run is successful, various outputs are copied to another directory and `profile.x` is run to generate an exit profile. Note that if they had been defined, `preexec()` and `postexec()` methods would be run as part of the NPSS model execution, not as part of the batch job.

```c
NCCJob comb {
    GlobusJobExec S_jobExec {
        rmContact = "turing.nas.nasa.gov";
    }

    queue = "@chapman";
    maxTime = 2*60;
    maxMemory = 2000;
    cpuCount = 256;
    directory = "GE90_240";
    environment = "PATH $PATH:";

    preexecCommands = {
        "cd ../Pre_GE90",
        "/a.out",
        "cp ncc_output ../GE90_240/ncc_radial_profile.1",
        "cd ../GE90_240"
    }
}
```
Interpreted Components

2.8.4 Namelist

The Namelist class supports reading and writing Fortran "namelist” data. Both the standard terminator “/” and the earlier non-standard terminator “&END” are recognized as terminating a namelist group. If a user wants to subdivide variables in a namelist, they can be put in one or more NamelistVargroup objects.

At this time this is only a partial implementation. Namelist features not yet supported include:
- Array data that starts with an integer, and subsequently has real data. A workaround is to use a trailing decimal point on the "integer" data.
- Arrays of more dimensions than supported by the corresponding NPSS type
- Fortran90 derived types

Despite these limitations, the Namelist class is useful for many typical applications using namelist formatted data. The following are the user visible attributes and methods:

**int nmlDynamic**
If TRUE, unknown variables are created during read(). Otherwise unknown variables generate an error. Default TRUE.

**VariableContainer nmlDefaults**
Used to hold default values for namelist variables. See setDefaults() and writeDefaults.

**string nmlGroup**
If not null, the name of the namelist group. Default namelist group name is the Namelist instance name.

**string nmlLogicals[]**
Variables of Fortran type "logical".

**int nmlModified[]**
Variable in nmlVars which have been modified. By default only modified variables are written. This array is updated upon reading a namelist or setting a namelist variable.

**int nmlTrace**
Displays operations during read().

**string nmlTerminator**
String used to terminate namelist, default “/”. Some older Fortran implementations want “&END.”

**string nmlVargroups[]**
Names of NamelistVargroup container objects which are contained within this object.
string nmlVars[]
Names of variables read/written.

int nmlWriteDefaults
If TRUE, write() will write all variables in the namelist. If FALSE (the default), write() will only write those variables not explicitly set whose values are different than the value of the corresponding variable in nmlDefaults, if it exists. Variables which have no corresponding nmlDefaults variable are always written. See setDefaults().

int nmlWriteElements
If non-zero, 2D and 3D arrays are written one element per line. Needed by at least one incarnation of HP Fortran.

int nmlWriteFullArrays
If non-zero, all array values are written, even if they match the default.

void clear()
Clear variable definitions and values to those specified by nmlDefaults.

int read(string stream)
Read namelist group matching this object’s instance name on stream. Returns TRUE if successfully read.

int readConditional(string stream)
Read namelist group matching this object’s instance name on stream. Returns 1 if successfully read, 0 if the group was not found, and -1 on error.

void setDefaults()
Copies the current variable definitions and values to nmlDefaults. These nmlDefaults variables will affect subsequent clear() and write() operations.

int write(string stream)
Write namelist group matching this object’s instance name on stream. Returns TRUE if successfully written.

For example, given the following instance definition:

```c
Namelist test1 {
    int   i = 42;
    int   l = TRUE;
    nmlLogicals.append("l");
    real  r = 3.14159;
    string s = "Hello World!";

    int   iarray[] = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    int   larray[] = { TRUE, FALSE, TRUE };
    nmlLogicals.append("larray");
    real  rarray[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };
    string sarray[] = {
        "testing", "one", "two", "three", "...
    };
}
```

Invoking test1.write() results in the following output written to the specified stream:

```
&TEST1
    i = 42
    l = .TRUE.
    r = 3.14159
```
s = 'Hello World!'  
intarray = 9, 8, 7, 6, 5, 4, 3, 2, 1  
boolarray = .TRUE., .FALSE., .TRUE.  
realarray = 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9  
strarray = 'testing', 'one', 'two', 'three', '...'  
/ 

## 2.8.5 Map Generation Classes

Some applications use multiple executions of one code to provide a map for another. These map generation classes support such applications by running a model at operating points specified by all combinations of particular input variables, collecting results, and outputting the results in the form of an NPSS map file.

### 2.8.5.1 MapGenerator

A MapGenerator object will generate a map based on data obtained by running the model. It refers to MGindependent and MGtable objects which determine at what points the model is run, what results are saved, and what tables make up the resulting map.

**string assembly**

If `assembly` is non-null, then that assembly is run at each operating point. If `assembly` is null (the default), then the top-level assembly is run.

**string validRunPtr**

If non-null, the name of a variable which is set TRUE if the run has valid results. The default value is null, indicating that all runs are assumed to have valid results.

**string filename**

Name of the file to write the generated map to.

**string comments[]**

Strings to write at the top of the map file.

**string independents[]**

Names of MGindependent objects which determine at what points the model is run.

**string tables[]**

Names of MGtable objects which determine what model results are collected and the tables to be output.

**int nRuns**

Number of model runs.

**real totalTime**

Total time for all model runs (seconds).

**real avgTime**

Average time for a model run (seconds).

**void addIndependent(string name)**

Add an MGindependent object to this MapGenerator. MGindependents which are created within the scope of a MapGenerator object are automatically added.
void removeIndependent(string name)
Remove the named MGindependent object from this MapGenerator.

void addTable(string name)
Add an MGtable object to this MapGenerator. MGtables which are created within the scope of a MapGenerator object are automatically added.

void removeTable(string name)
Remove the named MGtable object from this MapGenerator.

void run()
Run the assembly at each combination of independent values and write out table definitions.

2.8.5.2 MGdependent
An MGdependent object defines what model variables are collected after each run.

string variable
Name of the variable.

real values[]
Values of the variable collected after each model run.

2.8.5.3 MGindependent
An MGindependent object defines the operating points at which a MapGenerator runs the model. In some situations, an independent of a generated table is calculated during the map generation process. In this case, attribute calculated is set TRUE, and the values for the independent are retrieved from the model.

string variable
Name of the variable.

int calculated
If FALSE (the default) then the values attribute contains points at which the model is run. Otherwise the values attribute will be filled from data retrieved from the model.

real values[]
Values written to, or read from, variable.

Option interp
Value for the interp attribute of the generated table’s independent, default “linear.”

Option extrap
Value for the extrap attribute of the generated table’s independent, default “linear.”

2.8.5.4 MGtable
An MGtable object defines a table to be generated by a MapGenerator. The generated table will have the same name as the MGtable object.

string independents[]
Names of MGindependent objects. These are also the names of the arguments to the generated table.

string dependent
Name of MGdependent object. An MGdependent object created within the scope of an MGtable object will automatically set this attribute.
real s_rtn
Value for the s_rtn attribute of the generated table, default 1.0.

real a_rtn
Value for the a_rtn attribute of the generated table, default 0.0.

int printExtrap
Value for the printExtrap attribute of the generated table, default FALSE.

int extrapIsError
Value for the extrapIsError attribute of the generated table, default FALSE.

int valid[]
Flags indicating whether the corresponding dependent value is valid

void addIndependent(string name)
Add an MGindependent object to this MGtable. MGindependents which are created within the scope of an MGtable object are automatically added.

void removeIndependent(string name)
Remove the named MGindependent object from this MGtable.

void clear()
Clear data for new map generation run.

void update(int validRun)
Grab dependent and calculated independent values.

int write(string stream)
Write the table definition to the given stream. Values flagged as invalid are skipped.

2.8.6 Map Generation Example
The model below is an example of how a map generation process can be configured. This is not a real model, but is based on an actual application. Note that the resulting map uses PR as an independent, but the model calculates modelPR. Thus, the model’s MGindependent object PR is configured with calculated=TRUE.

```c++
// BETA is related to geometry.
real modelBETA;

// The RPM used in the map is %designRPM.
real mapRPM;
real designRPM = 2000;
real actualRPM;

FunctVariable modelRPM {
    setFunction = "_setRPM";
    getFunction = "_getRPM";
}
void _setRPM(real rpm) {
    actualRPM = designRPM * (rpm/100.0);
    mapRPM = rpm;
}
real _getRPM() {
    return mapRPM;
}
```
// The map PR is generated by runs at various %designW points.
real mapW;
real designW = 300;
real actualW;

FunctVariable modelW {
    setFunction = "_setW";
    getFunction = "_getW";
}

void _setW(real w) {
    actualW = designW * (w/100.0);
    mapW = w;
}
real _getW() {
    return mapW;
}

// PR is actually a calculated quantity, not something to be set.
real modelPR;
real modelWc;
real modelEFF;

MapGenerator mapGen {
    filename = "mini.map";
    comments.append("designRPM "+toStr(designRPM));
    comments.append("designW  "+toStr(designW));

    MGindependent BETA {
        variable = "modelBETA";
        values = (51.0, 60.0);
        interp = "linear";
        extrap = "none";
    }

    MGindependent RPM {
        variable = "modelRPM";
        values = (95.0, 100.0, 102.0);
        interp = "lagrange2";
        extrap = "none";
    }

    MGindependent massFlow {
        variable = "modelMW";
        values = (98.0, 99.0, 100.0, 101.0, 102.0);
    }

    MGindependent PR {
        variable = "modelPR";
        calculated = TRUE;
        interp = "lagrange3";
        extrap = "linear";
    }

    MGtable TB_Wc {
        MGdependent Wc {
            variable = "modelWc";
        }

        addIndependent("BETA");
        addIndependent("RPM");
    }
addIndependent("PR");
}

MGtable TB_eff {
  MGdependent EFF {
    variable = "modelEFF"
  }
  addIndependent("BETA");
  addIndependent("RPM");
  addIndependent("PR");
}

void calculate() {
  // Just some bogus data filling...
  modelPR = modelW - 97;
  modelWc = modelBETA*100 + modelRPM*10 + modelPR;
  modelEFF = modelBETA*0.0001 + modelRPM*0.01 + modelPR;
}

// Generate the map.
mapGen.run();

Running the above map generation model results in the following map file, named mini.map:

// designRPM 2000
// designW 300

Table TB_Wc (real BETA, real RPM, real PR) {
  BETA = 51.0000 {
    RPM = 95.0000 {
      PR = ( 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 )
      Wc = ( 6051.00, 6052.00, 6053.00, 6054.00, 6055.00 )
    }
    RPM = 100.000 {
      PR = ( 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 )
      Wc = ( 6101.00, 6102.00, 6103.00, 6104.00, 6105.00 )
    }
    RPM = 102.000 {
      PR = ( 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 )
      Wc = ( 6121.00, 6122.00, 6123.00, 6124.00, 6125.00 )
    }
  }
  BETA = 60.0000 {
    RPM = 95.0000 {
      PR = ( 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 )
      Wc = ( 6951.00, 6952.00, 6953.00, 6954.00, 6955.00 )
    }
    RPM = 100.000 {
      PR = ( 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 )
      Wc = ( 7001.00, 7002.00, 7003.00, 7004.00, 7005.00 )
    }
    RPM = 102.000 {
      PR = ( 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 )
      Wc = ( 7021.00, 7022.00, 7023.00, 7024.00, 7025.00 )
    }
  }

BETA.interp = "linear";
BETA.extrap = "none";
Interpreted Components

```c
RPM.interp = "lagrange2";
RPM.extrap = "none";

PR.interp = "lagrange3";
PR.extrap = "linear";

s_rtn = 1;
a_rtn = 0;

printExtrap = 0;
extrapIsError = 0;
```

Table TB_eff (real BETA, real RPM, real PR) {

BETA = 51.0000 {
    RPM = 95.0000 {
        PR = { 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 }
        EFF = { 1.95510, 2.95510, 3.95510, 4.95510, 5.95510 }
    }
    RPM = 100.000 {
        PR = { 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 }
        EFF = { 2.00510, 3.00510, 4.00510, 5.00510, 6.00510 }
    }
    RPM = 102.000 {
        PR = { 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 }
        EFF = { 2.02510, 3.02510, 4.02510, 5.02510, 6.02510 }
    }
}

BETA = 60.0000 {
    RPM = 95.0000 {
        PR = { 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 }
        EFF = { 1.95600, 2.95600, 3.95600, 4.95600, 5.95600 }
    }
    RPM = 100.000 {
        PR = { 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 }
        EFF = { 2.00600, 3.00600, 4.00600, 5.00600, 6.00600 }
    }
    RPM = 102.000 {
        PR = { 1.00000, 2.00000, 3.00000, 4.00000, 5.00000 }
        EFF = { 2.02600, 3.02600, 4.02600, 5.02600, 6.02600 }
    }
}

BETA.interp = "linear";
BETA.extrap = "none";

RPM.interp = "lagrange2";
RPM.extrap = "none";

PR.interp = "lagrange3";
PR.extrap = "linear";

s_rtn = 1;
a_rtn = 0;

printExtrap = 0;
extrapIsError = 0;
```
3 Internal and DLM Components

3.1 Pros and Cons

Both internal and DLM components run significantly faster than corresponding interpreted components, since both DLMs and internal components are compiled. Internal components have the disadvantage of requiring that the NPSS executable be rebuilt whenever their source code changes. If DLM source code changes, only the DLM must be rebuilt. DLMs also have the advantage of only being loaded when needed by a model, so they tend to save memory as well. There is really no reason to create internal components unless they are key infrastructure classes. The good news is that you can turn your component source into either an internal component or a DLM simply by passing different arguments into the NPSS make system. To build into a DLM,

```
make shared=single    // UNIX
nmake /f Makefile.win32 // Windows
```

To build into a library that can be linked into the NPSS executable,

```
make shared=          // UNIX
nmake /f Makefile.win32 nodll=1 // Windows
```

3.2 Objects to Aid Development of Compiled Components

3.2.1 Generic NPSS Values

**NCPRealScalarAdder**

NCPRealScalarAdder class is an extension of the type real. This class contains the attributes scalar, adder, base, and a Boolean that determines what value to return. All attributes can be displayed. This class inherits from VariableBase. Any value assigned to a variable of this type gets the scalar multiplied to base value; then the adder is applied. When the Boolean is set to TRUE or nonzero, the class returns the value with the scalar and adder applied. When the Boolean is set to FALSE or zero, the class returns just the base value. When assigning one NCPRealScalarAdder variable to another, only the value is copied, not the scalar, adder, base value, and Boolean. A warning message will be issued when the scalar or adder is set and the Boolean is FALSE or zero.

**Attribute names:**

- `s` scalar value, Type NCPReal, Default value = 1.0
- `a` adder value, Type NCPReal, Default value = 0.0
- `apply` Boolean used to determine what value to return, Type NCPInteger; Default value = TRUE or nonzero
- `b` value without the scalar and adder applied, Type NCPReal (get only); Default value = 0.0

**NCPVal**

Most objects in an NPSS model are inherited from VCIInterface, so they all support a generic get/set API. The API uses a class called NCPVal to represent a generic value. This generic value can be any of the types in the list shown below. The NCPVal class is a union of all of these types.

- NCPAny
- NCPString
- NCPStringArray1D
ValSequence
A ValSequence is simply an array of NCPVals that is used to pass generic arguments into NCPfunction objects.

3.2.2 Variable Types

The following variable objects are used by the component developer to give the user access to component attributes through the input file. All NPSS variable types are inherited from VInterface. In an abstract sense, an NPSS variable is an object with a name and a value. Variable objects also have other attributes such as I0status and description.

NCPFunctVariable<T, R>
NCPFunctVariables provide an efficient way to add side effects to an attribute. An NCPFunctVariable does not point to an attribute like an NCPRefVariable does. Instead it points to a “get” function and optionally a “set” function. This allows you to add side effects not only to setting the variable, but also to getting the variable. “T” is the class of the object containing the member function(s) that you want to be called. Typically it will be your component class. “R” is the return type of your “get” function and the argument type of your “set” function. Note that your “get” function return type and your “set” function argument type must be identical. The following is an example of how to create an NCPFunctVariable. It was taken from the file GasTblFlowStation.C.

```cpp
new NCPFunctVariable<GasTblFlowStation, NCPReal>("MN", this,
        &GasTblFlowStation::getMach,
        &GasTblFlowStation::setMachDesign,
        this);
```

NCPOptionVariable
A variable will have a finite number of allowed string values. Otherwise it acts like an NCPTTriggerVariable. If set to a value that is not allowed, it will generate an exception.

RefVarA     // any
RefVarD     // double
RefVarF     // float
RefVarI     // int
RefVarI1D   // NCPIntegerArray1D
RefVarI2D   // NCPIntegerArray2D
RefVarL     // long
RefVarS     // NCPString
RefVarS1D   // NCPStringArray1D
RefVarR     // NCPReal
RefVarR1D   // NCPRealArray1D
RefVarR2D   // NCPRealArray2D
RefVarR3D   // NCPRealArray3D

If you have variables that you use in internal component calculations but you want the user to be able to access them, use a RefVar. It provides a means to hook your internal variable into the NPSS infrastructure without slowing down your internal calculations. RefVar classes are available for all basic NPSS types.
NCPVariable
Contains an NCPVal. Using these in internal calculations involves more overhead than using an NCPRefVariable.

3.2.3 Port Objects

NOTE: Ports can be created only within an element, not a subelement. To create a port within your element, use the following function:

    PortPtr addPort(const NCPString& portType, const NCPString& portName);

For example,

    myFluidPortPtr = addPort("FluidInputPort", "Fl_I");

The following port types are available to the element developer.

FileInPort
FileOutPort
FluidInputPort
FluidOutputPort
FuelInputPort
FuelOutputPort
ShaftInputPort
ShaftOutputPort
ThermalInputPort
ThermalOutputPort

3.2.4 Station Objects

FlowStation
FlowStations represent conditions in a fluid stream at a single point. The available flow station functions depend on the property package used and are defined in the Reference Sheets document under the appropriate flow station type.

FuelStation
FuelStations represent conditions in a fuel stream at a single point. The available fuel station functions depend on the property package used and are defined in the Reference Sheets document under the appropriate flow station type.

3.2.5 Transient Histories

TransHistory objects are represented by the NCPTransHistory internal C++ class. These objects are created and must be registered just like any other object derived from VariableContainer. There is no special add function for NCPTransHistory objects.

Below is a list of the functions that can be called on a TransHistory object.

void clear()
Clears all data out of the history.

real extrapolate()
Based on the stored history, extrapolates a value to the current time.

real extrapolateTo(real newTime)
Based on the stored history, extrapolates to the desired value of time.
real getCurrentTime()
Returns the current time (either system time or local time)

real getCurrentValue()
Returns the current value of the parameter being tracked.

real getDerivative()
Returns the time derivative of the parameter at the current time.

real getDerivAt(int index)
Returns the time-derivative value at the indexed point.

real getDt()
Returns the current time-increment

real getDtAt(int index)
Returns the time-increment at the indexed point

real getDvalue()
Returns the difference between the current value and the previous value stored in the history.

real getDvalueAt(int index)
Returns the value difference for the indexed point

real getPastTime(int index)
Returns the value of time at the indexed point.

real getPrevTime()
Returns the previous value of time (system or local) stored in the history.

real getPrevValue()
Returns the previous parameter value stored in the history.

real getValue(int index)
Returns the parameter value at the indexed point.

void initChildHistories()
If your component overrides the initializeHistory() function (see below). This function should be called within the body of the override. This function sends the initializeHistory signal to all TransHistory objects and Subelements in the component.

void initializeHistory()
Sets the size of the history to default of 3, load the current parameter’s value into all points in the history, starting with the current time (system or local) and working backward in time, each point being separated by the current system time-step.

void initHistoryFull(int size, real dtime, real time, real value)
Sets the number of points in the history to size, and loads value into all points, starting at time and working backwards by increments dtime.

real interpolate(real oldTime)
Based on data in the history, performs a linear interpolation of the parameter value to the desired time.
void updateChildHistories()
If your component overrides the updateHistory() function (see below). This function should be called within the body of the override. This function sends the updateHistory signal to all TransHistory objects and Subelements in the component.

void updateHistory()
Updates the history by making the current value and time (system or local) the last point in the history and removing the oldest data point (size of history is fixed).

In cases where the function takes an index, the values should start at -1 for the previous point and decrease for older points (-2, -3, etc.). Derivative values are based on a simple backward difference formula. The extrapolation functions can be linear or three-point lagrange, and are set using the extrapolType attribute. The default is linear extrapolation.

extrapType = "LINEAR"; // other options are "LAGRANGE" and "NONE".

Note that the initializeHistory and updateHistory functions are called automatically during transient simulations.

### 3.2.6 Solver Objects

The syntax for defining solver objects within compiled components is considerably different than for Interpreted Components. The types of solver objects which can be included in compiled components are the same as those previously described for interpreted components. In addition to these objects, which are designed to work with the NPSS system solver, the developer can also utilize a secant solver, intended for use within components, which can solve simple iterative problems without the aid of the system solver.

#### 3.2.6.1 Using the System Solver

In compiled components, solver objects are created as follows.

```c
NCPDependent dep_ptr = addDependent(dep_name);
DSVPtr  dsv_ptr = addDSV(dsv_name);
NCPIndependentPtr indep_ptr = addIndependent(indep_name);
NCPSolverPtr integrator_ptr = addIntegrator(integ_name);
```

The functions noted above create an object of the appropriate type and register it with the component, with the name given in the argument list (which should be a quoted string or NCPSolver variable). Each function returns a pointer to the new object. The programmer can then address the object by name (using the set function as shown below) or using the pointer.

These new objects will be created with their autoSetup attribute set to "FALSE". For example, the following statement will create an NCPDependent named `dep_errWc`.

```c
NCPDependentPtr dep_errWcPtr = addDependent("dep_errWc");
```

Attributes can then be set in the `errWc` Dependent object as follows.

```c
set("dep_errWc.eq_lhs", "errWc"); OR
dep_errWcPtr->set("eq_lhs", "errWc");
set("dep_errWc.eq_rhs", 0.0000); OR
dep_errWcPtr->set("eq_rhs", 0.0000);
set("dep_errWc.tolerance", 0.0005); OR
dep_errWcPtr->set("tolerance" "0.0000");
set("dep_errWc.autoSetup", TRUE); OR
dep_errWcPtr->set("autoSetup", TRUE);
```
Certain Dependents can be used to constrain the operation of other, target Dependents. To do this, the developer should define the target and constraint Dependents first, as described above. Then the constraint Dependents can be added to the target Dependent using the `addConstraint` function.

```csharp
target_dep_name.addConstraint(constraint_dep_name, limitType, priority, slope_multiplier);
```

where the `limitType` is a string with values of “MIN” or “MAX”, priority is an integer greater than 0, and the `slope_multiplier` is 1 or -1.

Solver terms created for a given Element are not automatically added to the solver. Only solver objects which have their `autoSetup` attribute set to TRUE when the `autoSolverSetup` command is received will be added to the Solver. It is important that constraint Dependents are not added to the solver directly. They should always have their `autoSetup` attributes set to FALSE by the developer; constraints will be added to the solver when their target Dependents are added (targets may have their `autoSetup` set to TRUE). To remove a solver term, the user must set the `autoSetup` attribute to FALSE and call `autoSolverSetup` again as follows.

```csharp
set("dep_name.autoSetup", FALSE);
set("dsv_name.autoSetup", FALSE);
set("indep_name.autoSetup", FALSE);
set("integrator_name.autoSetup", FALSE);
autoSolverSetup();
```

When an existing solver object is to be added, the `autoSetup` attribute is set to TRUE and `autoSolverSetup` is called again.

The `autoSetup` attributes are often set in the `variableChanged` function of a Component, triggered by changes in the `switchDes` variable. For example, the code below will automatically configure a different solver setup depending on whether the `switchDes` is DESIGN or OFFDESIGN. The dependent shown here is only added to the solver in the OFFDESIGN mode. It is important to note, however, that the changes will not affect the Solver until the `autoSolverSetup` function is called again.

```csharp
if (switchDes == "OFFDESIGN")
    { set("dep_errWc.autoSetup", TRUE); }
else
    { set("dep_errWc.autoSetup", FALSE); }
```

In some cases, it may be desirable to remove all solver terms and start from a clean slate, rather than remove each term separately (especially when the objects which need to be added to the solver may depend on what the previous mode setting was, for instance). This can be accomplished using the `clearSolverTerms` function.

```csharp
clearSolverTerms(); // removes all solver objects from solver.
```

In some cases, it may be desirable to remove all solver objects and add back the ones required for a specific mode. This can be accomplished using the `clearSolverTerms` function.

```csharp
clearSolverTerms(); //sets autoSetup flags to FALSE
autoSolverSetup(); //adds back only objects having autoSetup
// set to “TRUE,” effectively removes all
// solver terms
```

**NOTE:** Calling `clearSolverTerms` merely sets the `autoSetup` flag to FALSE for all objects in the solver. The objects will not be removed from the Solver until the `autoSolverSetup` function is called again.
3.2.6.2 Using Component-Internal Solvers

In addition to creating Solver terms that will be controlled by the system solver, the NPSS includes a simple, one parameter solver that can be imbedded within components. This is called a secant solver, (see NPSS Secant Solver E-Spec for more information). This object is not suitable for use at the Model level and is therefore not described in the NPSS User Guide.

The SecantSolver is not a part of the NPSS architecture, but is an object that can be used internal to C++ routines. The execution logic for the component must specifically define the iteration loop, and uses methods on the SecantSolver to predict new independent values for each iteration, test for convergence, etc. Each PsSecant object handles a single independent and dependent-error. Unlike the system Solver, the SecantSolver attributes are all absolute, not fractional (refer to the NPSS Solver E-Spec document for an explanation of absolute and fractional parameters). Loops can be nested in the component logic to make use of multiple PsSecant objects, if desired.

The necessary steps for using the PsSecant object are as follows:

1. Create the PsSecant object.
2. Set the desired attributes such as maxIters, perturbation, etc. (functions and attribute listed below)
3. Initialize the PsSecant object.
4. Run component calculations and compute the error value
5. Set the new value of the independent parameter to the results of the PsSecant object’s iterate function.

Repeat steps 4 and 5 while the SecantSolver object’s Converged()and errorFound() functions both returns FALSE.

In the following example, a SecantSolver is used in an iteration loop to find the area required to produce a certain fluid pressure drop in a pipe.

```c++
void calculate() {
    // Step 1: Create the PsSecant object
    PsSecant solveArea();

    // Step 2: Set attributes
    solveArea.setMaxIters(20);
    solveArea.setPerturb(0.005);
    // use defaults otherwise);

    // Step 3: Initialize PsSecant object
    // set the independent parameters initial (guess) value
    area = 0.001;
    solveArea.initialize(area, this);

    // start iteration loop
    do {
        // Step 4 : Run component calculations and compute error
        // value.(W, Cf, rhoIn and dPqPdes are assumed inputs
        // here)
        NCPReal error;
        dPqP = pow((W/Cf/area), 2.0)/rhoIn;
        error = (dPqP - dPqPdes)/dPqPdes;

        // Step 5: Set area (independent) to results of iterate
        // function.
        area = solveArea.iterate(error);

        // Step 6: iterate until converged or found error.
        } while (!(solveArea.Converged()) &&
            !(solveArea.errforFound()));
    }
```

In the following example, a SecantSolver is used in an iteration loop to find the area required to produce a certain fluid pressure drop in a pipe.
PsSecant Functions and Attributes

Table 3-1. PsSecant Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>converged</td>
<td>integer flag (0=FALSE, 1=TRUE) indicating if error is within tolerance</td>
<td>0</td>
</tr>
<tr>
<td>errorType</td>
<td>integer flag indicating the type of solver failure encountered (0 = NOERROR, 1 = SINGULAR, 2=MAXITERS)</td>
<td>0</td>
</tr>
<tr>
<td>maxIters</td>
<td>maximum allowed number of iterations to reach convergence</td>
<td>50</td>
</tr>
<tr>
<td>maxDx</td>
<td>maximum allowed change in x (independent input) on a single iteration.</td>
<td>1.0e+10 (no limit)</td>
</tr>
<tr>
<td>numIter</td>
<td>current number of iterations executed.</td>
<td>0</td>
</tr>
<tr>
<td>perturbSize</td>
<td>the amount by which the independent input is varied to compute the effects on error.</td>
<td>1.00</td>
</tr>
<tr>
<td>tolerance</td>
<td>maximum allowed value of error for convergence to be attained</td>
<td>1.0e-03</td>
</tr>
<tr>
<td>switchThrowExceptions</td>
<td>NCPString (“TRUE”, “FALSE”) indicating if PsSecant will throw exceptions: ‘81013999 - Reached Max number of iterations’ and ‘81023999 - Slope between two points went off to infinity’, when those conditions occur. If set to “FALSE” exceptions will not be thrown.</td>
<td>“TRUE”</td>
</tr>
</tbody>
</table>

PsSecant Functions

PsSecant(const NCPString& name);
Basic constructor sets attributes to their default values.

PsSecant(const NCPString& name, NCPReal perturbSize, NCPReal tolerance, NCPReal maxDx, int maxIters);
Constructor which sets attributes shown to input values.

int errorFound();
Returns the value of errorType attribute.

int errorFlag();
Returns the value of errorType attribute.

int getCurrentIter();
Returns the value of the numIter attribute.

double getMaxDx();
Returns the value of maxDx attribute.

int getMaxIters();
Returns the value of maxIters attribute.
double getPerturb();
Returns the value of perturbSize attribute.

double getTolerance();
Returns the value of tolerance attribute.

void initialize(double initialX, VCInterface* this);
Initializes the independent (X) to the input value. The "this" pointer allows the error handler to point to the correct parent object.

int isConverged();
Returns the value of the converged attribute.

double iterate(double error);
Returns the independent value (X) that will drive error (Y) to zero, given the previous value of X and the corresponding input error.

void setMaxDx(const double& newVal);
Sets the value of the maxDx attribute to the input value.

void setMaxIters(const int& newVal);
Sets the value of the maxIters attribute to the input value.

void setPerturb(const double& newVal);
Sets the value of the perturbSize attribute to the input value.

void setTolerance(const double& newVal);
Sets the value of the tolerance attribute to the input value.

void setSwitchThrowExceptions(NCPString switchSetting);
Sets the value of the NCPOptionVariable switchThrowExceptions. If set to “TRUE” (default), PsSecant will throw exceptions: “81013999 - Reached Max number of iterations” and “81023999 - Slope between two points went off to infinity”, when those conditions occur. If set to “FALSE” exceptions will not be thrown.

NCPString getSwitchThrowExceptions();
Returns the value of the NCPOption variable switchThrowExceptions.

3.2.7 Sockets

Sockets provide a placeholder within your component for a subelement. You can specify the type of the socket, which forces any object inserted into the socket to be that type or inherited from that type. You can also specify a list of variable names that the subelement is permitted to set within your component. To add a socket to your component, use the following function:

    SocketPtr addSocket(const NCPString& sockName, const NCPString& sockType,
                        const NCPStringArray1D& varNameList)

Socket names should follow the naming convention of S_name.

For example,

    NCPStringArray1D allowedNames;
    allowedNames.append("x");
allowedNames.append("y");
allowedNames.append("z");
SocketPtr mySocket = addSocket("S_mySocket", "MySockType", allowedNames);

3.2.8 Function Objects

MemberFunction<T>

MemberFunction objects make it easy to add user-callable functions to your component. When you create a MemberFunction object, you specify the name of the user-callable function and the member function that should be called whenever the user-callable function is invoked. Your member function must have the following prototype:

NCPVal VAL_myFunct(const ValSequence& args);

If an existing member function that you want to call does not have this prototype, you can create a wrapper function with the correct prototype that just calls your existing function. In general there is no need for you to perform argument type checking on the incoming function arguments. This is taken care of for you by the MemberFunction class. The following example shows how a MemberFunction object is used to provide user-callable functions to a new element class called MyElement.

class MyElement : public Element {
    ...
    NCPVal VAL_myFunct(const ValSequence& args);
};

// create a function local to the MyElement.C file that adds the
// function(s) to the MyElement ClassObject.  Note that this
// function is NOT a class member function.  The name of the function
// must have the following prototype and naming convention:

void MyElement_setupClassObject(ClassObject* co)
{
    ValSequence args; //a list of dummy arguments used for
    // type checking
    args.reserve(1);
    args.append(NCPVal(NCPVal::REAL));  // add a real arg
    co->addClassMember(new MemberFunction<MyElement>("myFunct",
               NCPVal::REAL, args, &MyElement::VAL_myFunct));
}

// after declaring MyElement_setupClassObject, add a call to the
// following macro:

NPSS_INIT_CLASOBJ(MyElement, Element);

// the first arg of NPSS_INIT_CLASOBJ is the name of your class and
// the second is the name of its base class.

// at the beginning of all MyElement constructors except the copy
// constructor, call the NPSS_SETCO macro:

MyElement::MyElement(const NCPString& name) : Element(name)
{
    NPSS_SETCO(MyElement); // required by framework
    ...
}
// In your function, extract the function arguments and perform
// your calculation.
// Type checking of the arguments is not necessary; it’s taken care
// of by MemberFunction.

NCPVal MyElement::myFunct(const ValSequence& args)
{
    return 1.2 * cos(args[0]); // args[0] is a real argument supplied
    // by the user
}

3.2.9 Tables

Tables are implemented in the NPSS using the TableFunction class, which is inherited from NCPFunction, so TableFunctions can be evaluated using the same interface as NCPFunction. The prototype is as follows:

NCPVal eval(const ValSequence& args)

For convenience, a pair of functions, called evaluate() and evalReal, have been provided to easily evaluate function objects taking between zero and six arguments.

NCPVal FunctionRef::evaluate()
NCPVal FunctionRef::evaluate(NCPVal a1)
NCPVal FunctionRef::evaluate(NCPVal a1, NCPVal a2)
NCPVal FunctionRef::evaluate(NCPVal a1, NCPVal a2, NCPVal a3)
NCPVal FunctionRef::evaluate(NCPVal a1, NCPVal a2, NCPVal a3, NCPVal a4)
NCPVal FunctionRef::evaluate(NCPVal a1, NCPVal a2, NCPVal a3, NCPVal a4, NCPVal a5)
NCPVal FunctionRef::evaluate(NCPVal a1, NCPVal a2, NCPVal a3, NCPVal a4, NCPVal a5, NCPVal a6)
NCPReal FunctionRef::evalReal()
NCPReal FunctionRef::evalReal(NCPReal a1)
NCPReal FunctionRef::evalReal(NCPReal a1, NCPReal a2)
NCPReal FunctionRef::evalReal(NCPReal a1, NCPReal a2, NCPReal a3)
NCPReal FunctionRef::evalReal(NCPReal a1, NCPReal a2, NCPReal a3, NCPReal a4)
NCPReal FunctionRef::evalReal(NCPReal a1, NCPReal a2, NCPReal a3, NCPReal a4, NCPReal a5)
NCPReal FunctionRef::evalReal(NCPReal a1, NCPReal a2, NCPReal a3, NCPReal a4, NCPReal a5, NCPReal a6)

To access a table from C++ that has been declared by the user in the input file, the table or function pointer must first be located using the getFunction() call.

MyTablePtr = getFunction("TB_myTable");

If a table or function with the given name has not been declared, getFunction() returns NULL. The call to getFunction can be performed at the beginning of the calculate() function if desired, but it is more efficient to call it in the verify() function.

After the table/function pointer has been obtained, eval() or evaluate() can be called. For example:

If(MyTablePtr) {
    x = MyTablePtr->evaluate(y, z);
}
3.3 Component Functions

3.3.1 Required Functions

The following functions must be defined for your component. Otherwise it will not function correctly in the NPSS framework.

MyElement::MyElement(const MyElement& original)
This must call the base class copy constructor, for example if MyElement inherits from Element:

    MyElement::MyElement(const MyElement& original) : Element(original)
    {
        // copy internal objects here
    }

MyElement::MyElement(const NCPString& name)
This must call the appropriate base class constructor, for example:

    MyElement::MyElement(const NCPString& name) : Element(name)
    {
        NPSS_SETCO(MyElement); // required by framework
        // create ports, sockets, flowstations here
    }

virtual VCInterfacePtr clone()
This should create a new object using the copy constructor and return a VCInterface pointer to it. For example:

    void MyElement::clone()
    {
        return new MyElement(*this);
    }

virtual void calculate()
This is where all of the engineering calculations for the component should be performed. See the Duct and dPqP code in later sections for examples.

void MyElement_setupClassObject(ClassObject* co)
This function must reside in the .C file before the call to NPSS_INIT_CLASSOBJ or NPSS_INIT_CLASSOBJ_NOMATCH. Note that this is NOT a member function of the MyElement class.

3.3.2 Required Macros

NPSS_INIT_CLASSOBJ(className, baseName)
You must call this macro in file scope, i.e., outside of any function, in your .C file. If you don’t call this macro, the list(), hasInterface(), and isA() functions will not work properly with your component. If the internal C++ name of your class does not match the user accessible name, use the NPSS_INIT_CLASSOBJ_NOMATCH(realname, typename, basename) macro instead, where realname is the C++ name and typename is the name that the user sees. Note that the arguments to NPSS_INIT_CLASSOBJ or NPSS_INIT_CLASSOBJ_NOMATCH are NOT enclosed in quotes. For example, NPSS_INIT_CLASSOBJ(MyClass, Element).

NPSS_SETCO(className)
Inside of your component constructor, but not in the copy constructor, there must be a call to the NPSS_SETCO macro.

### 3.3.3 Optional Functions

**void localCreate(const NCPString& baseType, const NCPString& type, const NCPString& name)**

Override this function if you want to restrict the creation of objects in your component, either by preventing your component from instantiating particular child object types, or by keeping other objects in the system from instantiating child object types specific to your component. If `type` doesn’t match the name of your special object, then pass the call on to your base class (usually `Element::localCreate` or `Subelement::localCreate`).

**void variableChanged(const NCPString& varName, const NCPVal& oldValue)**

This function is called every time variable object with `trigger=TRUE` has its value set. Add code here to handle any side effects that should happen when the user sets certain variables. If you don’t recognize `varName`, make sure to pass the variableChanged call down to your base class (usually `Element::variableChanged` or `Subelement::variableChanged`). In general, NCPFunctVariables are a more efficient and powerful way to do the same thing.

**int verify()**

This function checks the integrity of the model, making sure that all of the objects are initialized properly and that all ports are connected, etc. If you are going to do a one-time initialization, e.g., for tables, you should do it when you use this function. This function is guaranteed to be executed prior to any run of the model if the model configuration has changed since the last run. Override this function if you need to perform additional checks, such as making sure that any required tables or functions are present or verifying that any required subelements are plugged into their sockets. See the dPqP code in a later section for an example of how to override `verify()`. Note that you should always call your base class `verify` (usually `Element::verify` or `Subelement::verify`) from your `verify()`, because your base class may perform additional verification. For example, the `Element base class verify()` function checks to see that all of your ports are connected.

### 3.3.4 Convenience Functions

The following functions exist to help developers create and interact with objects inside their component.

**DiscreteStateVariable* addDSV(const NCPString &dsvName)**

Creates a DSV with the given name, registers it with the component, and returns a pointer to it.

**FlowStation* getStation(const NCPString& stationName)**

Returns a pointer to the named FlowStation object. If the object is not found, it returns NULL.

**NCPDependent* addDependent(const NCPString &dependentName)**

Creates an independent with the given name, registers it with the component, and returns a pointer to it.

**NCPFunction* getFunction(const NCPString& functName)**

Returns a pointer to the named function or Table object. If the object is not found, it returns NULL.

**NCPIndependent* addIndependent(const NCPString& indepName)**

Creates an independent with the given name, registers it with the component, and returns a pointer to it.

**void needVerify()**

This forces your verify() function to be called prior to your next execution.

**Socket* getSocket(const NCPString& socketName)**

Returns a pointer to the named Socket object. If the object is not found, it returns NULL.
3.4 Error Handling
There are a number of functions available for reporting error conditions. The preferred ones are discussed in the following sections. It is not necessary to include the name of the object where the error occurred in the error message, because the full pathname of the object will automatically be prepended for you.

3.4.1 Provisional Errors and Warnings
Provisional errors are errors that occur during the solver iteration process before the model has converged to a solution. Provisional errors are collected during the current iteration pass. If that iteration pass results in a converged solution, then the provisional errors become “real” errors and are reported to the user in the form of exceptions. Provisional warnings are similar except that they result in warning messages instead of exceptions. Provisional errors should be reported using the provisionalError(const NCPString& msg) function, and provisional warnings should be reported using the provisionalWarning(const NCPString& msg) function.

3.4.2 Exceptions
To generate an exception call the throwError(const NCPString& msg, const NCPString& functName) function. The second argument to throwError() is optional.

3.4.3 Errors Found During verify()
Errors spotted in the verify() function should be displayed using the error(const NCPString& msg) function. This will print the error message to the error stream but it won’t throw an exception. The warning(const NCPString& msg) function may also be called from within verify() if the problem is not serious enough to be an error.

3.5 How to Add a Compiled Element to NPSS
You can add a compiled element to the NPSS in the following way.

1. Inherit your element from the Element class or inherit it from a class that is inherited from Element. Add all of the required member functions discussed previously to your element, as well as any optional ones that you require.

2. NPSS has an automated system for adding your component to a build. To utilize this system, you must add the following line to your component’s .C file:

   Register_NPSS_Object(<component_name>,<base_class_name>);

Note that <component_name> is NOT in quotation marks. For example:

   Register_NPSS_Object(Compressor,Element);

You should place the call to Register_NPSS_Object near the top of your .C file, after the #include for your component. For example:

   #include "Compressor.H"

   Register_NPSS_Object(Compressor,Element);

You must also add your component directory to the DIRS variable in the parent directory’s Makefile (or Makefile.win32). For example:

   DIRS = \
3. For an internal component, rebuild and re-link the entire NPSS, making sure that the object file for your new element, e.g., Duct.o, is linked in. See the following section on the NPSS make system for information on how to add your new code to the executable.

4. For a DLM component, compile and link it into a DLM, making sure that the DLM is placed in a directory where it can be found by DCLOD.

### 3.5.1 The Makefile System (UNIX)

If you are using the NASA makefile system for NPSS, then this section is relevant to you. If you are using another makefile system, check with your system administrator or other expert if you have questions.

The NPSS makefile system is a fairly complicated system of makefiles that allows the building of multiple targets for multiple systems in the same directory without any direct changes to the makefiles. For component developers, there is just one file of interest with three sections that explain the required functionality. This file is the `Makefile`.

- The first section is the **Config** section. It is at the top and bottom of the Makefile. It looks similar to this:

  ```make
  # Make system specific stuff . . .
  # which "template" we get our info from.
  # TOP = ../.. include $(TOP)/Executive.config

  include $(MAKEPRJ)
  
  <other sections described below>
  
  # Bring in our template stuff
  # include $(MAKEPRJ)
  
  The contents of this section is usually determined by your system administrator or whomever is responsible for your make setup. The thing to note here is that the variable TOP should be set to the location of the configuration file specified in the include directive.

- The second section is the **Directory** section. It usually contains one variable, DIRS. This variable contains a list of the subdirectories the make system is to go into.

  ```make
  DIRS = FuelPort \ 
  \ NewStreamPort \ 
  \ ShaftPort \ 
  \ ThermalPort
  
  NOTE: When a variable is specified on multiple lines, all but the last line must end in a '\'.

  To add a new directory, go into Makefile, and add the directory to the DIRS variable.

- The third section is the Makedef section. It usually looks as follows:
# compiler flags
CFLAGS =
CXXFLAGS =
FFLAGS =

# linker stuff
LOPTS =

# SRC to compile
SRC = DataManagerInterface.C

# local objects to make
OBJS =

# local dependancies
DEPENDS =

# local dirt, this is 0.$(NPSS_CONFIG) relative...
DIRT =

It contains the source files to compile in the SRC variable, and specifies the local compiler and linker options in the various variables:

- **CFLAGS** for C options
- **CXXFLAGS** for C++ options
- **FFLAGS** for Fortran options
- **LOPTS** for the linker options.

To add a new source file to compile, add it to the SRC variable. The make system knows how to compile C, C++ and Fortran files. The OBJS variable is intended for other esoteric uses. Adding a file to the SRC line is enough to get it compiled and into a build.

When adding a new component to NPSS:

- You must add the following line to your component's .C file:

  ```
  Register_NPSS_Object(<component_name>,<base_class_name>);
  ```

  Note that `<component_name>` is NOT in quotes. For example:

  ```
  Register_NPSS_Object(Duct,Element);
  ```

  You should place the call to Register_NPSS_Object near the top of your C file, after the #include for your component. For example:

  ```
  #include  "Duct.H"
  Register_NPSS_Object(Duct,Element);
  ```

- You'll also need to add your component directory to the DIRS variable in the parent directory's Makefile. For example:

  ```
  DIRS   = \    
  Bleed   \     
  Burner  \     
  CDNozzle \    
  Compressor
  ```
• You have to do a gmake in the parent (or higher) directory at least once after you've updated the Makefile.

### 3.5.2 The Makefile System (Windows)

The NPSS makefile system is a fairly complicated system of makefiles that allows the building of multiple targets for multiple systems in the same directory without any direct changes to the makefiles. For component developers, there are two files of interest; MakeVars.win32 and Makefile.win32.

MakeVars.win32 is a high level configuration file. There are multiple MakeVars.win32 files throughout the source tree, typically there will be one at the top of each source "sub" tree. Set variables in MakeVars.win32 that you want to carry though to all of your Makefile.win32 files in that tree. The top of each source sub tree is designated by the `TOP` variable. Each MakeVars.win32 file includes the MakeVars.win32 file above it in the source tree until finally the the MakeVars.win32 file at the top of the entire source distribution is included. This top level file is where all of the command line arguments passed to nmake are processed.

Below is an example of a typical component Makefile.win32 file for a C++ component:

```plaintext
TOP = ..\..\..
COMP = CppElem
NOCONVERT = 1
!include $(TOP)\MakeVars.win32
```

The variable `COMP` determines the name of the DLL. In this example, the DLL would be `CppElem.dll`. Setting `NOCONVERT` is required when building directly from C++ source code rather than converting interpreted source to C++ and then compiling.

### 3.5.3 Example of Adding a Compiled Duct Element

The following example shows you how to add an element called `Duct`. This duct represents an almost minimal element that provides meaningful NPSS functionality.

The Duct Header:

```plaintext
#ifndef __DUCTELEMENT_H
#define __DUCTELEMENT_H
#include <Element.H>
class FluidInputPort;
class FluidOutputPort;
class Socket;

class Duct : public Element
{
public:
    Duct(const NCPString& name);
    Duct(const Duct& duct);

    // required by framework
    virtual VCInterfacePtr clone() { return new Duct(*this); }

protected:
    virtual void calculate();

    // Fluid Ports
    FluidInputPort* _flowInPortPtr; //Inlet Fluid Port
    FluidOutputPort* _flowOutPortPtr; //Outlet Fluid Port
```

Internal and DLM Components 3-17
The Implementation file:

```cpp
#include "Duct.H"
#include <FluidInputPort.H>
#include <FluidOutputPort.H>
#include <UnitNames.H>
#include <Socket.H>
#include <SocketTypes.H>
#include <Constants.H>
#include <StringEngrConstants.H>
#include <V_NoArgMemberFunction.H>

//Add the component to NPSS
Register_NPSS_Object(Duct, Element);

// The following is required by the framework
void Duct_setupClassObject(ClassObject* co)
{
  NCPFunctio* func=0;
  co->addClassMember(new V_NoArgMemberFunction<DUCT>("calculate",
      &Duct::calculate));

  co->setUsageNotes("The duct calculates an adiabatic pressure drop.");
}

NPSS_INIT_CLASSOBJ(Duct, Element);
```

```cpp
Duct::Duct(const NCPString& name : Element(name),
  _a_dPqPaud (0.0),
  _dPqP (0.0),
  _dPqPbase (0.0),
  _s_dPqPaud (1.0)
{  
  NPSS_SETCO(Duct); // this call required by framework

  // add the pressure loss socket and give it permission
  // to set _dPqPbase
  _S_dPqP = addSocket("_dPqP", ADIAB_DPNORM,
      NCPStringArray1D("dPqPbase");

  // add the inlet and exit fluid ports
  _flowInPortPtr = (FluidInputPortPtr) addPort(Port::FLUIDIN,"Inlet") ;
  _flowOutPortPtr =
```
(FluidOutputPortPtr) addPort(Port::FLUIDOUT,"Fl_O");

    registerRefVariables();
}

/*---------------------------------------------------------------------------------------------
// copy constructor
/*---------------------------------------------------------------------------------------------
Duct::Duct(const Duct& duct) : Element(duct),
    _a_dPqPaud(duct._a_dPqPaud),
    _dPqP(duct._dPqP),
    _dPqPbase(duct._dPqPbase),
    _s_dPqPaud(duct._s_dPqPaud)
{
    // ports and sockets are automatically copied by the VariableContainer
    // so here we just have to find ptrs to them instead of creating them
    _S_dPqP = getSocket("S_dPqP");

    // get the inlet and exit fluid ports
    _flowInPortPtr = (FluidInputPortPtr) getPort("Fl_I");
    _flowOutPortPtr = (FluidOutputPortPtr) getPort("Fl_O");

    // since ref variables point to a specific place in this new object,
    // we can't have the VariableContainer copy them for us, so we have
    // to recreate them
    registerRefVariables();
}

/*---------------------------------------------------------------------------------------------
// Creates NCPRefVariables that point to our internal data
/*---------------------------------------------------------------------------------------------
void Duct::registerRefVariables()
{
    registerRealRef("a_dPqPaud", _a_dPqPaud, INPUT, NOUNIT);
    registerRealRef("dPqP", _dPqP, OUTPUT, NOUNIT);
    registerRealRef("dPqPbase", _dPqPbase, INPUT, NOUNIT);
    registerRealRef("s_dPqPaud", _s_dPqPaud, INPUT, NOUNIT);
}

/*---------------------------------------------------------------------------------------------
// Perform engineering calculations
/*---------------------------------------------------------------------------------------------
void Duct::calculate()
{
    // get the Flow Stations from the ports
    FlowStationPtr flowStatIn = _flowInPortPtr->getStation();
    FlowStationPtr flowStatOut = _flowOutPortPtr->getStation();

    // Check to see if the socket is empty. If not,
    // then execute it.
    if( !_S_dPqP->isEmpty() )
        {_S_dPqP->execute(); }

    // Apply audit scalar and audit adder to the bare socket value
    _dPqP = _s_dPqPaud * _dPqPbase + _a_dPqPaud;

    // determine the exit enthalpy and pressure
    NCPReal PtOut = (1.0 - _dPqP) * flowStatIn->getPtotal();
    NCPReal htOut = flowStatIn->get_hTotal();

    // pass the flow information from the inlet to outlet
    flowStatOut->copyFlow( *flowStatIn );
    flowStatOut->setTotal_hP( htOut, PtOut );
}
3.6 How to Add a Compiled Subelement to NPSS

The process for adding a compiled subelement to NPSS is identical to that of adding a compiled element.

3.6.1 Example of Adding a Compiled Aerodynamic Pressure Loss Subelement

The following example shows you how to add a subelement called $dPqP$. This subelement represents a typical subelement that provides typical NPSS functionality.

The $dPqP$ Header:

```c
#ifndef __DPNORMSUBELEMENT_H
#define __DPNORMSUBELEMENT_H

#include "Subelement.H"
#include "NCPOptionVariable.H"

// Derive the pressure loss from the subelement base class.
class DPNorm : public Subelement
{
    public:

        DPNorm(const NCPString& name);
        DPNorm(const DPNorm& original);

        // engineering calculations
        virtual void calculate();

        // return the subelement type
        virtual long type();

        // required by framework
        virtual VCInterfacePtr clone() { return new DPNorm(*this); }

        // Make sure pointers point to the right place.
        virtual int verify();

private:

        virtual void variableChanged(const NCPString& name, const NCPVal& val);

        void registerRefVariables();

        NCPReal  _Closs;       // scaled loss coefficient []
        NCPReal  _a_dPqP;     // adder for normalized pressure drop []
        NCPReal  _a_MN;       // adder for Mach number []
        NCPReal  _dPqPdes;    // Design normalized pressure drop []
        NCPReal  _dPqPmap;    // unscaled pressure loss []
        NCPReal  _dPqPbase;   // Normalized pressure drop []
        NCPReal  _MNmap;      // Mach number used to read table
        NCPReal  _RNI;        // Reynolds number index []
        NCPReal  _s_dPqP;     // scalar for dPqP []
        NCPReal  _s_MN;       // scalar for Mach number []
        NCPReal  _s_RNI;      // scalar for Reynolds number index []
```

```c
// pass the swirl through
flowStatOut->setSwirl( flowStatIn->getSwirl() );
```
NCPOptionVariable _switchMatch; //flag indicating how to match design point
NCPOptionVariable _SwitchDes;   // flag indicating design/off-design
NCFFunctionPtr _CloseTablePtr; // pointer to Close table
FlowStationPtr _flowStatPtr;
}
#endif

The dPqP implementation file:

#include "dPqP.H"
#include "StringEngrConstants.H"
#include "SocketTypes.H"
#include "UnitNames.H"
#include <math.h>
#include <VCCreator.H>
#include <RefVarR.H>
#include <V_NoArgMemberFunction.H>
#include <V_SA_MemberFunction.H>
#include <ValTypeSequence.H>

//Add the component to NPSS
Register_NPSS_Object(DPnorm, Subelement);

/*****************************************************************************/
/* required by framework */
/*****************************************************************************/

void dPqP_setupClassObject(ClassObject* co)
{
  NCFFunction* func=0;
  co->addClassMember(new V_NoArgMemberFunction<dPqP>("calculate",
          &dPqP::calculate));
  ValTypeSequence __variableChanged_args;
  __variableChanged_args.append(NCPVal::STRING);
  co->addClassMember(new V_SA_MemberFunction<dPqP>("variableChanged",
          __variableChanged_args,&dPqP::variableChanged));
  co->addInterface("ADIAB_DPNORM");
  co->setUsageNotes("dPqP returns the scaled normalized pressure drop");
}

NPSS_INIT_CLASSOBJ(dPqP, Subelement);

/*****************************************************************************/
/* constructor - intialize values */
/*****************************************************************************/

dPqP::dPqP( const NCPString& name ):
  Subelement(name, ADIAB_DPNORM),
    _Close(1.0),
    _a_dPqP(0.0),
    _a_MN(0.0),
    _dPqPdes(0.0),
    _dPqPmap(1.0),
    _dPqPbase(0.0),
    _MNmap(0.0),
    _RNI(1.0),
    _s_dPqP (1.0),
    _s_MN(1.0),
    _s_RNI(1.0),

Internal and DLM Components
void dPqP::calculate() {
    static NCPString DPNORMBASE("dPqPbase");
    NCPReal gams = _flowStatPtr->getGammaStatic();
}
NCPReal Ps = _flowStatPtr->getPstatic();
NCPReal Pt = _flowStatPtr->getPtotal();
NCPReal Tt = _flowStatPtr->getTtotal();
NCPReal MN = _flowStatPtr->getMach();
NCPReal Vhead = .5 * gams * Ps * MN * MN;

NCPReal powTt = pow( Tt, 1.1 );

// check to see if it is design mode
// If so, determine appropriate scalars.
if ( __SwitchDes == DESIGN ) {

    // Set all audit adders to zero and scalars to one.
    _a_dPqP = 0.0;
    _s_dPqP = 1.0;

    // Set the RNI scalar to give no Reynolds.
    // scaling at design
    _s_RNI = powTt / Pt;
    _RNI = Pt / powTt * _s_RNI;
    _MNmap = _s_MN * MN + _a_MN;

    // Check to see if there is a table to be read
    if ( __CloseTablePtr ) {
        _Closs = ( __CloseTablePtr->evaluate( _MNmap, _RNI ) ).rval();
    }

    if ( __switchMatch == SCALAR ) {
        // Determine the correct _dPqP scalar
        _s_dPqP = _dPqPdes * Pt / ( Vhead * _Closs );
    } else if ( __switchMatch == ADDER ){
        _a_dPqP = _dPqPdes - Vhead * _Closs / Pt;
    }
}

// perform off-design calculations

    // Scale the map independent parameters.
    _RNI = Pt / powTt * _s_RNI;
    _MNmap = _s_MN * MN + _a_MN;

    // Check to see if there is a table to be read
    if ( __CloseTablePtr )
        _Closs = ( __CloseTablePtr->evaluate( _MNmap, _RNI ) ).rval();

    // determine the unscaled pressure drop
    _dPqPmap = _Closs * Vhead / Pt;

    // determine the scaled pressure drop
    _dPqPbase = _dPqPmap * _s_dPqP + _a_dPqP;

    // set the pressure loss in the parent
    setValueInParent( DPNORMBASE, _dPqPbase);
}

// Allow for the subelement to change from design to off-design mode
// and back

void dPqP::variableChanged(const NCPString& name, const NCPVal& oldVal)
if (name == "SwitchDes") {
    // check the mode and set the I/O status accordingly
    if (_SwitchDes == DESIGN) {
        set("a_dPqP.IOstatus", OUTPUT);
        set("dPqPdes.IOstatus", INPUT);
        set("s_dPqP.IOstatus", OUTPUT);
    } else if (_SwitchDes == OFFDESIGN) {
        set("a_dPqP.IOstatus", INPUT);
        set("dPqPdes.IOstatus", OUTPUT);
        set("s_dPqP.IOstatus", INPUT);
    } else
        Subelement::variableChanged(name, oldVal);
}

// Check our table and flowstation pointers
-------------------
int dPqP::verify() {
    int retVal = ElementBase::verify();
    // get the table function pointer
    _CloseTablePtr = getFunction("CloseTable");
    // get the inlet flow conditions from the parent
    _flowStatPtr = getStationFromParent("P1_I");
    if (!_flowStatPtr) {
        NCPString msg = "can't get Station from parent";
        error(msg);
        return 0;
    } else
        return retVal;
}

3.7 Inheritance of Internal Elements and Subelements

The following figure shows the inheritance hierarchy for internal elements and subelements. Each box shows the functions that the element or subelement inherits from the object.

Figure 3-1. Inheritance of Internal Elements and Subelements
Internal and DLM Components

```
VCInterface
getName()
getFullPathName()
find()
isA()
hasInterface()
callclone()
sendEvent()
variableChanged()
verify()
throwError()
needVerify()
error()
warning()
message()

VariableOnlyContainer
contains()
getVC()
getVCList()
callclone()
registerVC()
unregisterVC()
list()

VariableContainer
getFunction()
insertFunction()
localExecuteFunction()

ElementBase
getStation()
socket()
addSocket()
getSocket()
addFlowStation()
getFlowStation()
addSocket()
addIndependent()
addDependent()
removeIndependent()
removeDependent()
calculate()
localCreate()
verify()

Element
localCreate()
getPort()
addPort()
verify()

Subelement
type()
setValueInParent()
getValueFromParent()

Assembly
linkPorts()
promotePort()
runPostSolverSequence()
runePreSolverSequence()
runeSolverSequence()
unlink()
```
4 DLM Development Kit

4.1 Overview of the DLM Development Kit

The DLM Development Kit (DK) is now a standard part of the NPSS distribution. It is located in the DLMdevkit directory off of the installed NPSS release environment. The Development Kit’s primary purpose is to allow particular sites to build what are termed Component Suites into DLMs—either single (one component) or lib (for multiple components) type—or to produce their own custom executables with certain component sets. For many situations the production of DLMs is all that is necessary.

4.1.1 What the DLM Dev Kit Can Do

The DLM Development Kit (DK) gives users the ability to do the following:

1. Build an NPSS executable, either the standard NPSS deliverable or one with custom, site-specific components (elements, subelements, thermos, etc.)
2. Build Dynamically Loadable Modules (DLMs) for use with NPSS, starting from either interpreted input files or directly from C++ source code.

Note: A technical paper entitled “Using a DLM to Couple One-Dimensional Compressor Code to an NPSS Simulation” was presented at the 2001 HPCC/CAS Workshop (July 31-August 2, 2001). It cannot be included in this document because it includes information on 1-D code. However, it may be a useful example of how to use a DLM.

From here on, the DLM Development Kit will be referred to as the “DK.” The phrase “a build” refers to an NPSS executable, with or without custom components and/or a DLM.

In the NPSS make system, there are currently three high-level conceptual groupings:

1. Component Suites
2. Front End
3. Executive Library

A component suite is usually referred to as a component source tree, as that is the physical item most often manipulated. A component source tree can be turned into a DLM module or combined with the Executive library and an appropriate Front End to produce a custom executable. Any number of component source trees can be combined, in either source form or compiled libraries, with a Front End (once again compiled or source form) and the Executive library to produce an executable. Thus, the standard NPSS executable can be viewed as a conglomeration of the following:

- Executive Library
- Standard Airbreathing Elements
- Standard Airbreathing Subelements
- Standard Ports
- Standard Thermos
- CommandLine Front End

After the DK is set up, it builds executables by producing NPSS Standard Component source trees and then instructing make to link some component source trees with the Executive library and a front end to produce an executable. Producing a DLM, either a lib type or a single type, is simpler; you just have to make in the
appropriate source directory with a different make option. The generated DLM(s) can then be dynamically loaded into an existing NPSS executable, such as the one provided in the release.

The remainder of this section explains how to configure an NPSS Development Kit and test an installation.

Requirements for Using the DK
To create a build using the DK, a site must have the following:

- A DLM Development Kit
- A supported NPSS platform with C, C++ and Fortran compilers
- GNU Make (UNIX only. nmake used under Windows)
- Python language interpreter (UNIX only)

The last two items can be viewed as either system utilities or NPSS Development required build tools, depending on whether they are on your system or not. Source for them is not included in the NPSS release but is available at the same location as the NPSS release. A section on how to test for their presence and install them is contained in a later section.

**NOTE:** Throughout the remainder of this document, wherever make is referred to it is assumed to be GNU Make (or NMAKE under Windows).

The NPSS Development Kit consists of:

- Makefiles and build-related items
- Standard NPSS Component suites as libraries
- The Standard NPSS CommandLine Front End
- Various build examples demonstrating building executables, DLMs, and customer decks

In practice, builders will have multiple component source trees that will be built either as DLMs or as part of a site-specific executable.

### 4.1.2 Installing the DLM Development Kit

This section details the installation of the NPSS Development Kit. The process consists of the following steps that are explained in detail below:

**Under UNIX:**
- Verify that GNU make and Python are available
- Verify that the appropriate environment variables are set
- (Optional) Modify the high level makefiles in the `NPSSvXX/DLMdevkit/makefiles` directory

**Under Windows:**
- Verify that NMAKE and Visual C++/FORTRAN compilers are available
- Verify that the appropriate environment variables are set
- (Optional) Modify the high level makefiles in the `NPSSvXX/DLMdevkit` directory

The DK is now part of the release, and its installation is contained as a part of the NPSS release installation. If a suitable version of GNU Make and Python are installed, the remainder of this section can be omitted, except perhaps the section on customizing the makefiles.
As per the installation instructions, when installing the NPSS release under UNIX, the script `npssenv` is run to determine the installation variables. It also attempts to determine whether a DevKit was installed and whether Python and GNU Make were found. It is rather primitive and not comprehensive, but it will find weather Python and GNU Make are in your path. It cannot yet determine whether the version of GNU Make is later than 3.77. It also only checks for make under the names “make” and “gmake.” Thus, for a quick, easy answer run the `npssenv` script. For more advanced detection, and for installation, proceed to the following section.

When installing under Windows, you can run a similar script, `npssenv.nt`. It will generate a batch file called `npssenv.bat` that you can run to set environment variables and add necessary directories to your path.

### 4.1.2.1 Testing for Make and Python (UNIX only)

Most of what is needed to produce a build (excluding the compiler, of course) is included in the kit. To use the DK, you will need two components that you may have to configure and install on your system. These are:

1. a version of GNU Make, 3.77 (or later).
2. a version of Python, 1.5.2 (or later).

Some systems already have these items installed.

To check whether you have GNU Make, do the following.

- **Type**: `make --version` (there are two hyphens).
  
  If you see a message similar to the following one, you probably have a usable copy of GNU Make:

  ```
  GNU Make version 3.77, by Richard Stallman and Roland McGrath.
  Copyright (C) 1988, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98
  ```

  Sometimes GNU Make is installed on a system under a different name, such as “gmake” or “gnumake.” If you find a version of GNU Make, but it is earlier than 3.77, you may want to have your local system administrator upgrade your current install.

- **To test whether you have Python**, simply type python. If the Python interpreter is installed, you should see something similar to the following:

  ```
  Python 1.5.2 (#1, May 4 1999, 16:29:22) [GCC egcs-2.91.60 19981201 (egcs-1.1.1 on sunos5
  Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam] >>>
  ```

  To exit the Python interpreter, press **CTRL-D**.

### 4.1.2.2 Installing Make and Python (UNIX only)

If you do not have GNU Make and/or Python and cannot get someone to install them, you must find a suitable location to make and install them yourself. The two programs are configured and installed in the same fashion, so the same instructions will work for both. The source for the version used at NASA Glenn is located at the same site where you acquired your NPSS release.

- **To install the two programs**, you will first need a suitable directory tree to install them to. These programs make a bin, lib, and perhaps other directories. If you do not want to use a directory such as `/usr/local` or `/usr/contrib`, then you could put them into the `util/` directory of wherever you untarred the DLMDevKit.

- **These programs are fairly intuitive to install and come with thorough installation instructions. Since they are typical net-available programs, you should be able to install them if you do the following:**
1. Untar the make `src` file, and then go to the top level directory created when you untarred the source file.

2. Type the following:

   ```
   ./configure --prefix=<full path to where you want to install make stuff>
   make
   make install
   ```

3. The `bin/` directory created in the `<prefix>` directory you specified is the `bin/` directory you should add to your path. If any complications arise, check the included build instructions or contact your local system guru for help.

4. After Make and Python are installed, check to be sure they run as described previously. Then verify that the Python scripts (`makeaddins.py` and `makemanifest.py`) in `DLM/DevKit/makefiles` run. These are shell scripts, but instead of using `sh` or `csh` they use Python. Since they use the `#!/usr/bin/env` syntax, as long as Python is in your path, the scripts should work. If they are not there, you can either add Python to your path or modify the first line of the scripts to read `#!/<full path to python interpreter>`.

   The scripts should produce error messages relating to missing arguments as opposed to a missing Python interpreter.

### 4.1.2.3 Setting Environment Variables

As described in the NPSS README file, certain environment variables should be set for the build environment to work properly. These are:

- **NPSS_TOP**
  - Points to the top of the NPSS distribution.

- **NPSS_DEV_TOP**
  - Points to the top of the NPSS Development Kit location or is empty if a Development Kit is not installed.

- **NPSS_CONFIG**
  - Holds a key identifying the platform the user is running on. Should be one of the following: sun, hp, sgi, nt, linux.

- **NPSS_MAKE**
  - Contains the default make program the development kit will use. It may be empty.

The script `npssenv` (or `npssenv.nt` under Windows) provided with the release has default values, but a particular site may want to customize the above values. `NPSS_DEV_TOP` and `NPSS_CONFIG` must be set properly to use the DK. `NPSS_MAKE` is currently of only limited value.

### 4.1.2.4 Customizing the System makefiles (UNIX)

Since you are building on an approved NPSS platform, one of the provided system makefiles should suffice, with only minor changes; you may have to change a few values. These values are the location of your `ar` command, C, C++ and Fortran compilers. The `which` command should be able to help you here.

Since the NPSS makefiles are set up to work on multiple machines simultaneously, there needs to be a way to specify which system makefile to use. The `system makefile` has the makefiles that contains the platform-specific information, such as which compiler and libraries to use, and where to put the compiled files. The makefile you should use is determined by the contents of the environment variable `NPSS_CONFIG`. The make system uses the
system-specific makefile named mk.$NPSS_CONFIG. In the makefiles directory you will see a few files labeled mk.something, such as mk.hp or mk.sgi. The comments in the specific make file are self explanatory.

4.1.2.5 Customizing the System makefiles (Windows)
You may need to modify the MakeVars.win32 file in the NPSSvXX/DLMdevkit directory. It contains most of the settings for the compiler, linker, and NPSS-to-C++ converter.

4.2 Creating and Running the Provided Example Builds
The Examples directory of the DK contains examples demonstrating the capabilities of the DK. These include:

- Turning a Component Suite into single type DLMs
- Turning a Component Suite into a lib type DLM
- Producing a custom executable with a given Component Suite.

The examples demonstrate the different builds that can be created with a properly configured NPSS Component Source Tree. The relevant directories in the DLMdevkit/Examples/src directory are as follows:

MultiCompDLM/
  Compiles multiple components into one DLM

C++CompToSingleDLM/
  Compiles a single C++ component into a single DLM.

InterpCompToSingleDLM/
  Compiles a single interpreted component into a single DLM.

ExampleDuct/
ExamPedPqP/
ExampleFlightConditions/
ExampleFlowEnd/
  Provides example components used in a simple system model. Under UNIX, these components can be built either as individual DLMs or combined into one DLM.

4.2.1 Component Source Tree and Building Overview

4.2.1.1 Component Source Trees
Source trees can be organized many different ways. Site-specific concerns will most likely dictate how a user’s particular source tree is organized. The source tree layout that follows is recommended but may be modified to satisfy a site’s particular needs.

The following is a typical source tree directory layout and contents (# is used before comments):

SiteComponents/ #Name relevant to contained components
  src/ #Where the components live
    Component1/
    Component2/
  lib/$NPSS_CONFIG #where the component compiled lib goes
  DLMComponents/$NPSS_CONFIG #where the component DLMs go.
  include/$NPSS_CONFIG #where the component generated headers go
  MetaData/$NPSS_CONFIG #where the component meta data files go
The recommended source tree is designed to be self-sufficient and easy to package and move around. It consists of a *TOP* level directory named with something relevant to the components contained within.

### 4.2.1.2 Building Under the NPSS Make System (UNIX)

Building items controlled by the NPSS make system usually entails typing **make** from the *TOP* or *src* directory of a source component tree. This causes make to move downward and make all the directories underneath it, producing that particular executable or library.

Building an executable usually consists of building in the appropriate component directories with the appropriate options, then in the *FrontEnds* directory. Producing DLMs is similar; you just make in each component source tree using the appropriate options.

Since the NPSS Dev Kit uses the standard NPSS make system, all the make commands described in Chapter 3, “Internal and DLM Components,” apply. The most useful are:

- **make clobber**: Clobbers everything from the current directory and below. Clobbering entails removing the exe/DLM, the .o’s and the make-created directories
- **make clean**: Removes the .o’s from the current directory and below

Doing a **make clobber** from the *TOP* directory cleans out all make-generated items.

In addition to the above make commands, make also supports the setting of make variables on the command line. Make variables, such as “debug” and “profile,” can enable/disable special compilation of the source or trigger the inclusion of extra libraries.

Some common make variables include:

- **debug**: enable/disable debugging build
- **profile**: enable/disable profiling in the exe
- **optimize**: enable/disable optimization of this build
- **purify**: link with purify support (if available at your site)

Typing **make help** in a particular directory with a make file will provide a more comprehensive list of the settable make variables for that make location. Make variables can be specified as environment variables on make’s command line or in the makefiles. Make prioritizes the values as follows, highest precedence first:

1. Command line option
2. Value in a makefile
3. Value from environment

For example, **make debug=1** passed as a **command line option** will override any debug settings in a makefile or the environment.

Since make variables are strings, their behavior when used as Boolean switches is non-intuitive. This is because an empty or non-existent string is considered “False,” whereas any non-empty string is “True.” For example:

- **make debug=0** enables debug
- **make debug=1** enables debug
make debug=<anything> enables debug
make debug= disables debug

Setting a variable to empty does not mean the variable is not set. For example, setting debug to 1 in an environment variable and then setting “debug= ” in a makefile file will not use the environment variable’s value because the makefile variable’s value has a higher priority, and it is being set to empty.

4.2.1.3 Building Under the NPSS Make System (Windows)

Building items controlled by the NPSS make system usually entails typing nmake /f Makefile.win32 from the TOP or src directory of a source component tree. This causes make to move downward and make all the directories underneath it, producing that particular executable or library.

Building an executable usually consists of building in the appropriate component directories with the appropriate options, then in the FrontEnds directory. Producing DLMs is similar; you just make in each component source tree using the appropriate options.

Since the NPSS DLMdevkit uses the standard NPSS make system, all the make commands described in Chapter 3, “Internal and DLM Components,” apply. The most useful are:

nmake /f Makefile.win32 clobber
Clobbers everything from the current directory and below. Clobbering entails removing the exe/DLM, the .obj’s and the make-created directories

nmake /f Makefile.win32 clean
Removes the .obj’s from the current directory and below

Doing a nmake /f Makefile.win32 clobber from the TOP directory cleans out all make-generated items.

In addition to the above make commands, make also supports the setting of make variables on the command line. Make variables, such as “debug,” can enable/disable special compilation of the source or trigger the inclusion of extra libraries.

Some common make variables include:

debug enable/disable debugging build
nodll compile everything into the executable, or build a given component tree into a lib instead of a dll.

Make variables can be specified as environment variables on make’s command line or in the makefiles. Make prioritizes the values as follows, highest precedence first:

1. Command line option
2. Value in a makefile
3. Value from environment

For example, nmake /f Makefile.win32 debug=1 passed as a command line option will override any debug settings in a makefile or the environment.

4.2.2 Compiling a Component Source Tree as a DLM (UNIX)

A DLM component’s source code is created exactly the same way that internally compiled NPSS components are created. The big differences are in how it is made and how it is packaged. How it is made includes the different
compiler options and the way build issues are handled behind the scenes. How it is packaged is strongly tied to the organization of the component’s source tree.

There are two major modes for packaging DLMs.

The first is for a DLM that contains multiple components. An example of this is `ExampleComponents.scl`, which contains the following four components: ExampleDuct, ExampleFlowEnd, ExampleFlightConditions, and ExamplePnorm. This method will be referred to as the lib method because a source tree produces one DLM shared component library. The other DLM packing scheme is one in which each component builds to a DLM component (e.g., DLMDuct compiles to `DLMDuct.sc`, and so on). This method is referred to as the single method. Once again, the only difference is in how they are made, or more specifically, the actual arguments given to make.

Some items of note: The default DLM build produces a lib type DLM. Single component DLMs are made by setting the make variable (shared) to single. This can be done from the command line as in:

```
make shared=single
```

Single component DLMs are made and placed in `$TOP/DLMComponents/$NPSS_CONFIG` directory by default. Set the `DLLDIR` make variable in `Makefile.win32` in the component directory to change the destination. The DLM component will be a DLL under Windows, so it will have the .dll extension, for example, `DLMDuct.dll`. The name of the .dll file is determined by the value of the `COMP` variable in `Makefile.win32`. If the component is written directly in C++, i.e., it is not converted from interpreted code, you must set `NOCONVERT=1` in `Makefile.win32`. Make sure that `COMP` matches the class name of your component that will be contained in the DLL.

The other DLM packing scheme is for a DLM that contains multiple components. An example of this is `MultiCompDLM.dll`, which contains two simple components. This method will be referred to as the lib method because a source tree produces one DLM shared component library. Lib DLMs are placed in the same directory as individual DLMs. Their name is specified by setting the `TARGNAME` variable in their local `Makefile.win32`.

### 4.2.3 Compiling a Component Source Tree as a DLM (Windows)

A DLM component’s source code is created exactly the same way that internally compiled NPSS components are created. The big differences are in how it is made and how it is packaged. How it is made includes the different compiler options and the way build issues are handled behind the scenes. How it is packaged is strongly tied to the organization of the component’s source tree.

There are two major modes for packaging DLMs.

The first is one in which each component builds to a DLM component (e.g., DLMDuct compiles to `DLMDuct.dll`, and so on). This method is referred to as the single method. Single component DLMs are made and placed in the `$TOP/DLMComponents/nt` directory by default. Set the DLLDIR make variable in `Makefile.win32` in the component directory to change the destination. The DLM component will be a DLL under Windows, so it will have the .dll extension, for example, `DLMDuct.dll`. The name of the .dll file is determined by the value of the `COMP` variable in `Makefile.win32`. If the component is written directly in C++, i.e., it is not converted from interpreted code, you must set `NOCONVERT=1` in `Makefile.win32`. Make sure that `COMP` matches the class name of your component that will be contained in the DLL.

The other DLM packing scheme is for a DLM that contains multiple components. An example of this is `MultiCompDLM.dll`, which contains two simple components. This method will be referred to as the lib method because a source tree produces one DLM shared component library. Lib DLMs are placed in the same directory as individual DLMs. Their name is specified by setting the `TARGNAME` variable in their local `Makefile.win32`. 
When NPSS loads a DLM, it looks for a function called `<dlm_name>_init`, where `<dlm_name>` is the name of the DLM file without the extension. When we want to create a DLM containing multiple components, we need to call the `<comp_name>_init` function for each component in the DLM from our `<dlm_name>_init` function, so that NPSS will know how to create each type of component. There is also a required function called `<dlm_name>_version` that works in the same way as `<dlm_name>_init`. This is done in an automated way using Python under UNIX, but under Windows it must be done manually.

The function prototypes for the required functions are:

```c
extern "C" {
    void <comp_name>_init();
    const char* <comp_name>_version();
}
```

If components are converted from interpreted code to C++, their C++ source code will automatically contain the necessary `<comp_name>_init` and `<comp_name>_version` functions. If the component is written directly in C++, then the functions must be added manually. This is easily done using the `Register_NPSS_Object` macro, e.g.,

```cpp
Register_NPSS_Object(MyElement,Element);
```

The DLMdevkit example, MultiCompDLM, demonstrates how multiple components can be added to a DLM. In the DLMdevkit/Examples/src/MultiCompDLM directory, there is a file called `CompManifest.C` that contains the `MultiCompDLM_init` and `MultiCompDLM_version` functions.

### 4.2.4 Examining the Provided Examples

There are three examples provided in the `Examples/src` Directory. This section discusses how they work.

The first example is located in the `C++CompToSingleDLM` directory. It is intended to demonstrate how to build a DLM component when starting with a source written in C++. Building the component is the primary focus here, so this component has minimal functionality. Some of the other example components, e.g., ExampleDuct, show more details of component internals. To build this example under Windows, type:

```bash
cd C++CompToSingleDLM
nmake /f Makefile.win32
```

Under UNIX, type:

```bash
cd C++CompToSingleDLM
make
```

This builds the DLM, which is called `CppExample.dll` under Windows, and `CppExample.sc` under UNIX.

To run a simple test on the CppExample DLM, type:

```bash
nmake /f Makefile.win32 test
```

or

```bash
make test
```

The output should look something like this (UNIX example shown):

```
Hello world! (C++ example)
Resource found in: ../../DLMComponents/sun/CppExample.sc
```
The second example is located in the InterpCompToSingleDLM directory. It is intended to demonstrate how to build a DLM component when starting with a component written in NPSS interpreted input. This component has FlowStations and also demonstrates the use of trigger variables. To build this example under Windows, type:

```
        cd InterpCompToSingleDLM
        nmake /f Makefile.win32
```

Under UNIX, type:

```
        cd InterpCompToSingleDLM
        make
```

This builds the DLM, which is called `InterpElement.dll` under Windows, and `InterpElement.sc` under UNIX.

To run a simple test on the InterpElement DLM, type:

```
        nmake /f Makefile.win32 test
```

or

```
        make test
```

The output should look something like this (UNIX example shown):

```
foo1: copying flow from input to output ... done
foo2: copying flow from input to output ... done
foo1.triggerVar changed
foo2.triggerVar changed
Resource found in: ../../DLMComponents/sun/InterpElement.sc
```

The third example is located in the MultiCompDLM directory. It is intended to demonstrate how to build a DLM that contains multiple components. In this case, there are two components in the DLM, Comp1 and Comp2. These components are interpreted, but C++ components can also be combined into the same DLM in a similar fashion. To build this example under Windows, type:

```
        cd MultiCompDLM
        nmake /f Makefile.win32
```

Under UNIX, type:

```
        cd MultiCompDLM
        make
```

This builds the DLM, which is called `MultiComp.dll` under Windows, and `MultiComp.sc` under UNIX.

To run a simple test on the MultiComp DLM, type:

```
        nmake /f Makefile.win32 test
```

or

```
        make test
```

The output should look something like this (UNIX example shown):

```
Comp1::calculate called from foo1
Comp2::calculate called from foo2
Resource found in: ../../DLMComponents/sun/MultiComp.sc
```
There are also a number of example components in the Examples/src directory, named ExampleDuct, ExampleFlightConditions, ExampleDpP, and ExampleFlowEnd that are tested together as a simple system. These are fully functional engineering components, so they are more detailed than previous examples. The simple system test is found in the Examples/test directory. After building the various Example??? components, cd to the Examples/test directory and type make or nmake /f Makefile.win32, and you should see output like this:

Duct1 Pressure Loss: 0.03
Duct1 Entrance Mach: 0.3
Duct1 Entrance Flow: 10
Duct1 Entrance Area: 56.2233
Duct1 Mode: DESIGN
Duct1 FluidIn.MachDesign: 0

Duct1 Pressure Loss: 0.03
Duct1 Entrance Mach: 0.3
Duct1 Entrance Flow: 12
Duct1 Entrance Area: 67.468
Duct1 Mode: OFFDESIGN
Duct1 FluidIn.MachDesign: 0

4.2.4.1 Building a Custom Executable (UNIX)

To produce an executable the main file, FrontEnds/CommandLine/npss.C, must be compiled. This is accomplished as follows:

```
cd FrontEnds/CommandLine
make ADDINS="$NPSS_DEV_TOP/Examples/ExampleComponents" BINDROPDIR=cur_dir
```

The above make line actually demonstrates some new make variables. BINDROPDIR specifies the directory where the produced executable goes. It is similar to the ODIR variable described below. The ADDINS variable is used to add component source trees to the produced executable.

The above make line builds the executable adding in the ExampleComponents components and drops the executable in cur_dir. The ADDINS option above will only work if you have built ExampleComponents.lib by using the `shared=' make argument.

The following make variables are useful no matter what script you are using:

- **ADDINS**: It takes a space-separated list of Paths to Component Source Trees, or ar libraries containing components. If your Component Source Tree’s name is the same as the library it generates (see below), the make system will figure everything out for you if you give it the name of the Component Source Tree. If the lib produced by your Component Source Tree does not have the same name as the root directory, you can include the full path to the library.

- **ODIR**: Specifies which directory the generated object files should be dropped into. By default this is O.$NPSS_CONFIG underneath the directory the source file is compiled in. If multiple people are expected to build from that same source at the same time, ODIR and the other DIRS described below should probably be set to something specific for them.

- **BINDROPDIR**: Specifies where the generated executable will be placed.
COMPDROPDIR  Specifies the location the generated DLM Components will be placed in.

LIBDROPDIR  Specifies the location where ar component libraries will be placed.

with_ab  Instructs the make system to link in the NPSS Standard AirBreathing components as:

debug  enable/disable debugging build
profile  enable/disable profiling in the exe
optimize  enable/disable optimization of this build
purify  link with purify support (if available at your site)

A full list of the variables supported may be obtained by running make help in a Component Source Tree dir FrontEnds directory.

4.2.4.2 Building a Custom Executable (Windows)

To produce an executable the main file, FrontEnds/CommandLine/npss.C, must be compiled. This is accomplished as follows:

        cd FrontEnds/CommandLine
        nmake /f Makefile.win32

To link additional libraries into the executable, add them to the DEP_LIBS variable in Makefile.win32.

4.3 The Layout of an NPSS Component Source Tree

To produce a build involves many complicated steps; these are all handled by the NPSS make system. The recommended way to set up the make system for a source directory is to copy the files from an existing source tree, such as the Examples tree in the NPSS Dev Kit.

Make works by examining a file named “Makefile” in the directory where make is invoked. The contents of this Makefile tell it to do various things, most notably to enter a directory and compile some source files. There is a lot of make file code involved in producing a build. Most of it is standard and tucked away where it will not interfere with the developer, but there are a few things you do need to specify. These are:

1. The location of the NPSS DevKit makefiles
2. The library names the compiled files should go into and where they reside
3. Global compiler options
4. Build specific make options
5. The source files to compile
6. The directories to go into

4.3.1 UNIX Configuration

Some of the previously mentioned items are necessary for all the make files, and some are more specific to a particular make file. In light of this, the items necessary for all the make files are placed in a config file and then included from the other makefiles. Thus, in addition to hiding much of the boilerplate make code (actually abstracting it out), you can also abstract some of the items from the above list. Specifically, a config file tends to contain the following:

1. The location of the NPSS DevKit makefiles
2. The library names the compiled files should go into and where they reside
3. Global compiler options
4. Build-specific make options

For example, a typical config file looks something like this:

```bash
## example config file
## This file is included by every makefile down this directory tree.
## The make flags, variables, and options common
## to all the items in this tree should go in here.
##
##NOTE: These can always be overridden at the command line level or
## in a local makefile.

# Default make flags should go here.
# debug=1
shared = lib

# Compiler type flags for everyone go in this tree.
# PROJECT_CFLAGS +=
PROJECT_CXXFLAGS += -I. \n-Is$(NPSS_DEV_TOP)/Executive/include \n-Is$(NPSS_DEV_TOP)/AirBreathing/include \n-Is$(NPSS_DEV_TOP)/Ports/include

PROJECT_FFLAGS +=
PROJECT_LOPTS +=

# for a static library and lib type DLMs
# LIBDIR = $(TOP)/lib
LIBNAME = ExampleComponents

# DLMs where nothing is specified go into the following dir by default.
# COMPDROPDIR=$(TOP)/DLMCcomponents/$(NPSS_CONFIG)

# These are standard and would probably not be touched.
MAKEFILEDIR= $(NPSS_DEV_TOP)/makefiles
MAKETYPE=Component.make
MAKEPRJ= $(MAKEFILEDIR)/$(MAKETYPE)
```

Most of the contents will be the same for your source trees; they may differ only in the LIBNAME and perhaps the LIBDIR variables. Each source tree is meant to stand alone, and each source tree references things relative to itself—from its top level directory. In the makefiles this is represented in the value of the variable TOP. Each makefile sets its TOP variable (also at the top of its file) and from this can figure out where its config file is.

Following is the example component directory from the NPSS Dev Kit, with the appropriate values of TOP shown.

Example/
ExampleComponents.config
For each makefile, you still have to specify:

- Which source files to compile
- Which directories to go into

Two typical makefiles are shown below. The first one goes into only the directories specified in the DIRS line. This makefile is typical of all the makefiles except for the ones in the actual component directory. Those are just used to guide the make down particular directories.

```make
TOP = ..
include $(TOP)/ExampleComponents.config

# We want to do a manifest

TARGETS = manifest

# DIRS to recurse into

DIRS = \nExampleDuct \nExampleFlowConditions \nExampleFlowEnd \nExampledPqP

#compiler flags
CFLAGS =
CXXFLAGS =
FFLAGS =

#linker stuff
LOPTS =

#SRC to compile
SRC =
```

The makefile presented here is from the src directory, the directory that contains all the component directories. This makefile has a special requirement in that it must make as a manifest. It needs the line

```
TARGETS = manifest
```

to instruct the make system to create a manifest. A manifest lists the contents of the library you are building. **NOTE:** This makefile, i.e., the one above the component source, controls which components are made and which components will be in the library.

This next makefile is one from a component directory, specifically the Examples/src/ExampleDuct component. It differs from the previous one in that no DIRS has been accessed, it has SRC to compile, and it does NOT make a manifest.

```
###
TOP = ../.. include $(TOP)/ExampleComponents.config
TARGETS =

#
# DIRS to recurse into
#
DIRS =

#compiler flags
CFLAGS =
CXXFLAGS =
FFLAGS =

#linker stuff
LOPTS =

#SRC to compile
SRC = ExampleDuct.C

#local dirt, this is $(ODIR) relative...
DIRT =

include $(ODIR) relative...

include $(MAKEPRJ)
```

Using the information on the structure of a source component tree, you should now know how to make it by typing `make` in the top directory of a source component tree. The library made should exist as the value of LIBNAME in the LIBDIR/sun, hp or sgi directory.

### 4.3.1.1 Converting Interpreted Components

The NPSS make system can automatically convert interpreted components into compiled code. If the SRC line of a makefiles contains an interpreted component, such as
then the NPSS Converter will be invoked to convert the component. Then the generated code will be compiled and processed like the rest of the tree, that is, converted into DLMs or put into archive libraries.

4.3.2 Windows Configuration

Under Windows, configuration files are named MakeVars.win32. Usually there is a MakeVars.win32 file at the top of each source tree, and this file is included in each make file in the source tree. Set any variables here that you want to carry through all of your make files in the tree.

4.3.2.1 C++ vs. Interpreted Components

Make files for interpreted components are the same as make files for C++ components except for a single line. In the C++ case, the following extra line is needed, before the MakeVars.win32 file is included:

```
NOCONVERT = 1
```
5 NPSS Model Delivery Development Kit

The Model Delivery Development Kit, or MDdevkit, provides the developer with a set of tools and make files to allow him to create stand-alone executables that will only run a specific model. This executable will allow the user to access only the variables and functions that the developer has specified. There are two examples provided with the MDdevkit, and they can be found in the $NPSS_TOP/MDdevkit/Examples directory. One builds a customer deck that uses the NPSS native API. It can be found in the native directory. The other creates a 4191 common block interface for use with legacy FORTRAN codes. It can be found in the 4191 directory. Each example directory in the Windows distribution contains a detailed README file describing how to build the example customer deck.

5.1 Security Model and Access Control

NPSS includes two distinct methods for controlling access to objects in a model or within NPSS itself. Even though these methods are distinct, they do interact in determining whether one object may view another. The first method is defined as one object being secure or insecure with respect to another object. In this method, we discuss "security level" which, in NPSS, is actually implemented as a security ID attribute (not visible to the user). The second method is defined by assigning an object a certain access permission, which is defined as one of three: RW = Read-Write, RO = Read-Only, or PRIV = Private. How these attributes are manipulated is described in the next two sections. The following is a description of how they interact.

When a request is made to access any object within an NPSS model, access is granted if any of the following is true:

1. The requested object is marked as visible (Read-Write or Read-Only)
2. The requesting object is "secure" (with respect to the requested object)
3. The requesting object and the requested object are both "unsecure" (i.e., at the same security level)

The following table describes the potential combinations of these methods. $O_0$ represents an object with Security Level 0, and $O_1$ represents an object with Security Level 1. $O_0$ is considered "secure," while $O_1$ is "unsecure." All objects at the same security level can see into each other regardless of the access permissions. Objects with RO permission can be seen but not changed by "unsecure" objects.

Table 5-1. Does Requester Have Access to Provider?

<table>
<thead>
<tr>
<th>Requestor</th>
<th>Provider</th>
<th>Access Permission assigned to Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PRIV</td>
</tr>
<tr>
<td>$O_0$</td>
<td>$O_1$</td>
<td>Y</td>
</tr>
<tr>
<td>$O_1$</td>
<td>$O_0$</td>
<td>N</td>
</tr>
</tbody>
</table>

All objects that are part of the original model (i.e., the model that is compiled into the customer simulation by the model developer or stored in the encrypted model file) can be made "secure." Any new objects created by the customer are "unsecure." Aliases are always visible, as are any objects which have been set to Read-Write or Read-Only access using the setAccess() function, usually from an encrypted model file. The following paragraphs
describe in more detail the security ID and attributes of NPSS which enable the customer deck builder to ensure that certain parts of the model are protected.

5.1.1 Security ID - secure, unsecure

Every object that is created in NPSS also has a security object which is created along with it. This security object contains an ID attribute, which indicates the associated object’s security level, and is set to the value of the security ID of the static security object at that time. This static security object is created before any other objects are created and initially the ID is set to 0. This is the “most secure” ID and anything with this security ID can see into any other objects created after it. In order to create a “less secure” layer of objects, the current static security object is deleted and a new one created. This new security object will now have a security ID of 1. Objects with security ID = 1 cannot see into objects with a security ID of 0, but those with ID = 0 CAN see into those with ID = 1. Each time the static security object is deleted and recreated, this ID increments by 1. Every object created after the new static security object is created will have a security object with its ID set to that of the static object. So, for instance, in the following lines of code:

```java
Element Duct MyDuct;
activateSecureMode();
Element Compressor MyComp;
```

assuming this is the first time we have called `activateSecureMode()`, `MyDuct` will have a security ID of 0 and `MyComp` will have a security ID of 1. Therefore, `MyDuct` is “secure” in relation to `MyComp`, and `MyDuct` can see `MyComp`, but `MyComp` cannot see `MyDuct`.

5.1.2 Access Control - PRIV, RO, RW

Each object has an attribute known as access (previously known as public). This attribute can be one of three values: RW = Read-Write, RO = Read-Only, PRIV = Private. There is also a static variable called defaultAccess which determines what access will be set to as each object is created. This is set to “RW” by default but can be changed by the user by calling the setDefaultAccess() function. Thereafter, every object created will have its access attribute set to defaultAccess. After the object is created, the access attribute can be changed by calling the setAccess() function and supplying the object name and the desired access type (both as string arguments). A third argument to this function is called recurse. If set to 1, the function will recurse down below the object and set ALL of its children and grandchildren and their children, etc., to this access type. If set to 0, it will set ONLY the attribute of the object itself. This function also takes wildcard naming for the object name argument (“*” or “?”).

This function follows the same security model rules, in that it cannot change the access attribute of an object it is not allowed access to. In general, it is expected that these functions will be used to configure a model within the encrypted model file. After the encrypted model file is read in for a customer deck, security is turned on and the customer no longer has access to any part of the model that hasn’t been explicitly opened for them (has access = “RW” or access = “RO”).

Example:

```java
class Example extends Element {
    int var1;
    int var2;
} // end Example

Element Example MyEx_1;
MyEx_1.var1 = 1;
MyEx_1.var2 = 2;
```
cout << "For MyEx_1:
   var1 = " << MyEx_1.var1 << ", var2 = " << MyEx_1.var2 << endl;

// changes access for JUST 'MyEx_1.var2'
setAccess("MyEx_1.var2", "PRIV");

// should print var1 and var2
cout << "\nFor MyEx_1:
   var1 = " << MyEx_1.var1 << ", var2 = " << MyEx_1.var2 << endl;

// set default access to private (0)
setDefaultAccess(0);

Element Example MyEx_2;
MyEx_2.var1 = 11;
MyEx_2.var2 = 12;

// should print var1 and var2 for MyEx_1 and MyEx_2
cout << "\nFor MyEx_1:
   var1 = " << MyEx_1.var1 << ", var2 = " << MyEx_1.var2 << endl;

// end Example

This code produces the following output:

For MyEx_1:
   var1 = 1, var2 = 2

For MyEx_1:
   var1 = 1, var2 = 2

For MyEx_1:
   var1 = 1, var2 = 2

For MyEx_2:
   var1 = 1, var2 = 12

Entering secure mode...

For MyEx_1:
   var1 = 1, var2 = ERROR(9100) in file docTest - line 36 - CASE 0 - Unresolved variable "MyEx_1.var2"

For MyEx_2:
   var1 = ERROR(9100) in file docTest - line 38 - CASE 0 - Unresolved variable "MyEx_2.var1"

ERROR(9000) in file docTest - line 41 - CASE 0 - in MemberFunction 'setAccess': Could not find 'MyEx_1.var2'.

NPSS Model Delivery Development Kit
5.1.3 Functions for Manipulating Security and Access

void activateSecureMode()
Turns security on and increments the current security ID by one, thus creating a security barrier between objects already existing and objects that get created thereafter. Objects already existing can see into new objects, but the new objects cannot see into the existing objects unless they have been made public (have their access attribute set to "RW" or "RO").

void setDefaultAccess(int access)
Sets the static attribute defaultAccess. All objects created after this will have their access attribute set to this value. The access parameter can be set to one of three values: 2 (Read-Write), 1 (Read-Only), 0 (Private or hidden). This access is initially set to 2 (Read-Write) upon NPSS startup. (NOTE: The int argument access is currently inconsistent with the string argument taken by the setAccess() function and will be corrected in the next release, with CR1404.)

int getDefaultAccess()
Returns the current value of the static variable defaultAccess, either 2 ("RW"), 1 ("RO"), or 0 ("PRIV"). (NOTE: Again, the consistency issue between the return value being int or string will be resolved when CR1404 is implemented. A string will be returned.)

void setAccess(string name, string access, int recurse)
This flexible command allows the user to change the value of the access attribute of objects in the model. 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.

int getAccess(string name)
Returns the value of the access attribute belonging to the object called name (either 0, 1, or 2).

void setAlias(string name, string alias)
Creates an alias object for name which can then be accessed using alias, which is accessible, even if the original object is Private.

string getAlias(string name)
Returns the alias name for the object called name, if there is one. Otherwise, an error is produced.

5.2 Building a Customer Simulation Using the Native NPSS API

A customer simulation using the native NPSS API operates much like a full version of NPSS, except that the set of variables, functions, and object types available to the customer is only a subset of those available to a user of the full version of NPSS. The model developer determines which variables, functions and object types will be visible to the customer.

5.2.1 Native API Description

The native NPSS API consists of the functions below. All functions have an integer return value. A return value of 0 indicates successful completion.
int NPSS_init(char *fnames[], int numArgs)
This function processes the commandline arguments passed through fnames[] by setting certain flags and creating an array of files to be read in. The number of arguments stored in fnames is numArgs. The internal function setupCustomerModel is then called, which initializes NPSS using either the compiled in encrypted model, or an external encrypted model file if the -e option is specified. The encrypted model typically will contain function calls to set aliases, create permissions, and access permissions. After parsing the encrypted model, security is turned on. Next, the additional encrypted and non-encrypted model files from the commandline are read in. Lastly, if the interactive flag is set, interactive command line mode is started. Note that this won't work if the user does not have access permission for icl, the interactive command line object. The NPSS_init function must be called before any of the other API functions will work.

int NPSS_parseFile(char *filename)
 Parses the file indicated by filename. The file must contain valid NPSS input syntax.

int NPSS_parseString(char *str)
 Parses the string str. The string must contain valid NPSS input syntax.

int NPSS_parseEfile(char *encfile)
 Reads in encrypted model files after NPSS has been initialized using NPSSinit(). This function is not needed for the first encrypted model file as the NPSSinit() function processes that one.

int NPSS_parseEstring(const char*s, unsigned long size);
Parses the encrypted string contained in s. The string must contain valid (after decryption) NPSS input syntax.

int NPSS_getI(char *name, int* i)
 Stores the value of the model variable specified by name in the int pointed to by i.

int NPSS_getF(char *name, float* f)
 Stores the value of the model variable specified by name in the float pointed to by f.

int NPSS_getD(char *name, double* d)
 Stores the value of the model variable specified by name in the double pointed to by d.

int NPSS_getS(char *name, char *buff, int buffSize)
 Stores the value of the model variable specified by name in the string pointed to by buff.

int NPSS_getI1D(char *name, int *buff, int buffSize, int *numRead)
 Stores the value of the model variable specified by name in the int array pointed to by buff. Stores the number of entries in the array in the int pointed to by numRead.

int NPSS_getF1D(char *name, float *buff, int buffSize, int *numRead)
 Stores the value of the model variable specified by name in the float array pointed to by f. Stores the number of entries in the array in the int pointed to by numRead.

int NPSS_getD1D(char *name, double *buff, int buffSize, int *numRead)
 Stores the value of the model variable specified by name in the double array pointed to by buff. Stores the number of entries in the array in the int pointed to by numRead.

int NPSS_getI1Dentry(char *name, int index, int* i)
 Stores the value of the model variable specified by name and index into the int pointed to by i.

int NPSS_getF1Dentry(char *name, int index, float* f)
 Stores the value of the model variable specified by name and index into the float pointed to by f.
int NPSS_getF2Dentry(char *name, int index1, int index2, float *f)
Stores the value of the model variable specified by name, index1, and index2 into the float pointed to by f.

int NPSS_getD1Dentry(char *name, int index, double* d)
Stores the value of the model variable specified by name and index into the double pointed to by d.

int NPSS_getD2Dentry(char *name, int index1, int index2, double *d)
Stores the value of the model variable specified by name, index1, and index2 into the double pointed to by d.

int NPSS_getS1Dentry(char *name, int index, char *buff, int buffSize)
Stores the value of the model variable specified by name and index into the string pointed to by buff.

int NPSS_get1Dsize(char *name, int *dim)
Stores the number of entries in the model array variable specified by name in the int pointed to by dim.

int NPSS_get2Dsize(char* name, int* size1, int* size2)
Gets the dimensions of a two-dimensional array identified by name. Sizes are returned in the call list. The function also returns a status integer ID: 0 = success, -1 = failure.

int NPSS_get3Dsize(char* name, int* size1, int* size2, int* size3)
Gets the dimensions of a three-dimensional array identified by name. Sizes are returned in the call list. The function also returns a status integer ID: 0 = success, -1 = failure.

int NPSS_getDimSize(char* name, int* buff, int buffsize, int* numRead)
Gets the dimensions of a one-, two-, or three-dimensional array. The sizes are returned in the buff array. The number of entries in the array are returned in numRead. The user may enter buffsize, the maximum allowed size for buff as input (a value of 3 is recommended). The function also returns a status integer ID: 0 = success, -1 = failure.

int NPSS_setI(char *name, int i)
Sets the value of the model variable specified by name to the value of i.

int NPSS_setF(char *name, float f)
Sets the value of the model variable specified by name to the value of f.

int NPSS_setD(char *name, double d)
Sets the value of the model variable specified by name to the value of d.

int NPSS_setS(char *name, char *buff, int buffSize)
Sets the value of the model variable specified by name to the string pointed to by buff.

int NPSS_setIID(char *name, int *buff, int buffSize)
Sets the value of the model variable specified by name to the int array pointed to by buff.

int NPSS_setFID(char *name, float *buff, int buffSize)
Sets the value of the model variable specified by name to the float array pointed to by buff.

int NPSS_setDID(char *name, double *buff, int buffSize)
Sets the value of the model variable specified by name to the double array pointed to by buff.

int NPSS_setIIDentry(char *name, int index, int i)
Sets the value of the entry specified by index in the model variable specified by name to the value of i.
int NPSS_setF1Dentry(char *name, int index, float f)
Sets the value of the entry specified by \textit{index} in the model variable specified by \textit{name} to the value of \textit{f}.

int NPSS_setF2Dentry(char *name, int index1, int index2, float f)
Sets the value of the entry specified by \textit{index1} and \textit{index2} in the model variable specified by \textit{name} to the value of \textit{f}.

int NPSS_setD1Dentry(char *name, int index, double d)
Sets the value of the entry specified by \textit{index} in the model variable specified by \textit{name} to the value of \textit{d}.

int NPSS_setD2Dentry(char *name, int index1, int index2, double d)
Sets the value of the entry specified by \textit{index1} and \textit{index2} in the model variable specified by \textit{name} to the value of \textit{d}.

int NPSS_setS1Dentry(char *name, int index, char *buff, int buffSize)
Sets the value of the entry specified by \textit{index} in the model variable specified by \textit{name} to the string pointed to by \textit{buff}.

int NPSS_onNPSSTermination()
Calls the CommandInterface version of onNPSSTermination. It takes no arguments and returns 0 upon completion.

int NPSS_variableExists(char* name)
Returns 1 if the variable exists in the model, 0 otherwise.

\subsection{5.2.2 Creation Process}

This section describes the step-by-step process of taking an NPSS input file and generating a stand-alone customer simulation from it. There are two variations on the creation process. In the first, the NPSS input file is encrypted and compiled into the executable. Therefore, every time this particular executable is run, it will be initialized with the same encrypted model. If the model needs to be changed, the executable must be rebuilt. This is the default creation process. In the second variation, the NPSS input file is encrypted and stored in a file external to the executable. This file is specified by providing CDgen with the \textit{--e} option followed by a filename in which to store the encrypted model. By supplying the \textit{--k} option followed by the name of an existing key file, the encryption key contained in the file will be used by CDgen to encrypt the model. If the key file does not exist, CDgen will generate a random key and write it to the key file. Having an external key file allows the customer deck creator to encrypt additional input files that are compatible with a particular customer deck. This is done by using the external key file with the \textit{scramble} program to produce a new encrypted file. NPSS native customer models are able to read in multiple encrypted and non-encrypted files after being set up with the initial model. Again, all encrypted files must be encrypted using the same key that is stored in the executable.

The following programs are supplied with the DevKit to help the customer deck creator to generate and test encrypted input files:

\texttt{scramble <keyfile> <modelfile> <encryptedfile>}
Takes a text file \texttt{<modelfile>} and encrypts it using the key supplied in \texttt{<keyfile>}. The encrypted output file is called \texttt{<encryptedfile>}.

\texttt{descramble <keyfile> <encfile> <decryptfile>}
Takes an encrypted input file \texttt{<encfile>} and decrypts it using the key from \texttt{<keyfile>}. The output is written to \texttt{<decryptfile>}. The primary purpose of descramble is to verify that the encrypted file is being decrypted properly. The descramble program uses the same decryption code used inside of NPSS.
**singlefile**  <-d preproc_var> <-I include_dir> <inputfile>

The singlefile program takes `<inputfile>` and expands out any #includes, allowing the user to fully expand a model, encrypt it using scramble, and feed it to a customer model that uses the same key. The `-d` command line argument allows the user to define preprocessor variables. The `-I` option allows the user to add a directory to the file search path. The output of singlefile is sent to standard output, which can then be redirected into an output file.

The following steps assume that the model developer has all of the necessary NPSS infrastructure libraries available in non-debug form, i.e., libraries built with code compiled without the `-g` option.

1) **Specify Creation Permissions**

The customer model developer can limit the types of objects that the customer is allowed to create. This is done by placing calls to the `addCreatePermission()` function. The function calls must be placed somewhere within the encrypted input, prior to any non-encrypted files that require the specified permissions to exist. For example:

```plaintext
addCreatePermission("UserFunction");
addCreatePermission("Independent");
addCreatePermission("Dependent");
addCreatePermission("VarDumpViewer");
```

In the example above, the user will be able to create only the objects shown.

2) **Create Aliases**

The `setAlias()` function creates an alias for an existing object in the model. The function calls should be placed within the encrypted input. For example:

```plaintext
setAlias("realName", "aliasName");
```

3) **Set Default Access Permission**

The following sets the default access permission for any subsequent objects:

```plaintext
setDefaultAccess("PRIV");
```

In this instance, all subsequent objects will be created with access permission of “PRIV”. If no call to `setDefaultAccess()` is made, the internal default access permission is “RW”, i.e., all objects are visible.

4) **Set Specific Access Permissions**

Like aliases and create permissions, access permissions can be set by placing function calls within the encrypted input. For example:

```plaintext
setAccess("myObj", "PRIV", 0);
setAccess("obj2", "RO", 1);
setAccess("obj3", "RW");
```

The first argument specifies the object whose access permissions are being set. The second argument determines what the access permission is. Possible values are private (“PRIV”), read-only (“RO”), and read-write (“RW”).

The third argument is optional. It determines if the setAccess call should apply to the given object only (if arg=0), or to the object and all of its children (if arg!=0). Note that it is much faster to call `setDefaultAccess(<accessType>)` at the beginning of the input rather than to call `setAccess("", <accessType>, 1)` after all the objects in the model have been created.

Access permissions in NPSS are similar to access permissions in a file system. Objects containing other objects are analogous to folders, i.e. if an object container has restricted access permission, its children are also restricted by
that permission. For example, if a given Element has its access permission set to “PRIV”, then none of its child objects will be visible to the user.

5) Use CDgen to Create C++ Files
CDgen takes the model file it to generate C++ code necessary for the setup of the model. The following C++ file is generated by CDgen:

```
setupCustomerModel.C  // loads the model
```

CDgen has the following command line options:

```
-m <model file>  // specifies the name of the file containing the customer model
-e <encrypted file>  // specifies the filename in which to store the encrypted model
-k <key file>  // specifies the file containing the encryption key
-d  // specifies that the customer deck will be built with the capability to load DLMs
-I <include dir>  // adds a directory to the #include search path
```

For example:

```
CDgen.sun -m modelfile
```

Explanation of Command Line Options (-e, -k, -d) Used with CDgen
The `-e` command line option, followed by a filename, is used to specify an external file into which the encrypted model file will be written. If this option is not included, the model file will be encrypted and compiled into the executable. The encrypted model will also be written out to a default filename of “model.enc”.

The `-k` option, followed by a filename, is used to specify an external file into which the encryption key is written. If this file already exists, the key within the file will be used by CDgen to encrypt the model. If this file does not exist, a new encryption key is generated (which is used to encrypt the current model file). It is then saved using the filename that follows the `-k`. If no `-k` option is provided, then a new key is generated and used for encryption, but it is not saved. Saving the key file allows the customer deck generator to reuse an encryption key to encrypt multiple model files using the `singlefile` and `scramble` programs and to then read these encrypted files into the customer deck which generated that key. This avoids generating a new executable for each encrypted model file created and allows one executable to read in more than one encrypted model file.

The `-d` option tells the CDgen program to build a customer deck capable of searching for DLM components using DCLOD. Search directories for DCLOD can be specified on the command line using the `-I` option, or as specified in the NPSS_PATH environment variable. If this option is not provided, the customer deck will NOT run with DLMs.

Examples:

```
CDgen.sun -m modelfile
```

This command causes CDgen to create a customer deck where the encrypted model will be compiled into the executable. It will also be stored externally in a file called `model.enc`. The key will NOT be stored external to the customer deck, and the capability to load DLMs will NOT be included.
CDgen.sun -m modelfile -e encModel -k keyfile -d
This command causes CDgen to create a customer deck where the encrypted model file will be stored in encModel. The key will be stored in the file keyfile and also compiled into the customer deck, and CDgen will search for DLMs using the NPSS_PATH environment variable search path.

6) Supply a Driver Program
A simple driver program is required to initialize the customer model and to allow the customer to supply additional input files to the simulation. An example driver program is shown below. Also check the example driver.C program in the Devkit for additional detail.

```c
/* ------------------ Program driver.c --------------------- */
#include <NPSSNativeAPI.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    if(NPSS_init(argv, argc)) {
        fprintf(stderr, "Error in NPSS initialization\n");
        NPSS_terminate();
        return -1;
    }
    NPSS_terminate();
    return 0;
}
```

Providing the “-i” option to this program on the command line allows the customer to run the customer simulation in interactive mode. Note that interactive mode will not work unless the icl object is made public by calling setAccess("icl", "RO") from your encrypted model file.

7) Compile and Link
Compile your driver program and the files generated by CDgen and link them with any necessary NPSS libraries.

NOTE: Make sure that the libraries are non-debug libraries.

5.3 Building an In-House Simulation Using the Native NPSS API

5.3.1 Native API Description
For in-House simulations one additional NPSS API function is used (see section 5.2.1 for a list of all the other native NPSS API functions).

```c
void NPSS_setInHouse(int inHouseModel)
```
This function passes the value of inHouseModel to the NPSS Native API. The default value is 0, which tells the Native API to produce a customerdeck. Setting the parameter inHouseModel to ‘1’, tells the NPSS Native API to create an inHouse Model.

5.3.2 Creation Process
To generate a stand-alone inHouse simulation a driver program must be compiled and linked with any necessary NPSS libraries. This driver program is used to initialize the inHouse model and is somewhat different from the
driver.C example file shown in section 5.2.2. Basically, the changes include a call to set the variable "inHouseModel" and an empty "setupCustomerModel()" to satisfy an external reference. An example driver program for inHouse models is shown below.

```c
/* ------------------ Program driver.c --------------------- */
#include <NPSSNativeAPI.h>
#include <stdio.h>

void setupCustomerModel()
{
}

int main(int argc, char *argv[])
{
    NPSS_setInHouse(1);
    if(NPSS_init(argc, argv)) {
        fprintf(stderr, "Error in NPSS initialization\n");
        NPSS_terminate();
        return -1;
    }
    NPSS_terminate();
    return 0;
}
```

5.3.3 Building the Customer Simulation as a Shared Library

**NOTE:** The shared library version of the customer deck is not yet supported on the HP platform.

This is an enhancement to the Model Delivery system, which allows an NPSS model to be built and delivered as a dynamic shared library (DLM/dll). To use this method to deliver a model, the developer follows the same procedure for generating the model code as with the stand-alone version. With the dll version of the customer deck, there is no driver in the shared library. Users will need to develop the driver for the test suite or code they are using. The dll version of the NPSS model can be used with FORTRAN and C codes.

The customer deck developer will need to include the:
- DLM/dll
- platform specific APITranslate.o file
- APITranslate.h file.

APITranslate contains the functions necessary to access the customer deck shared library. These functions are wrappers to the associated NativeAPI functions used for the standalone version of the customer deck.

The following is a list of these functions:

Each function returns a zero (0) if successful or a one (1) upon failure.

```c
int FAPI_dlopen_(char* sc);
Takes the dlm/dll path and name as the argument.

int FAPI_init_(char* file);
Takes the customer deck arguments in a space delimited test file. The first argument is the non-white-space model name.
```
The examples below are examples of how the shared version of the customer deck was used and tested with FORTRAN and C. There are slight differences between the platforms and compilers on how the FORTRAN code need to be written. LINUX needs “0” as the last character in a string being passed from the FORTRAN source to the shared library and the HP need aliases as shown in the second FORTRAN example file.

The example Makefile shows the compiler flags used to build the drivers used for the tests and can be used as a reference.

FORTRAN Example 1, accesses the model (filename: ftest.f):

```
PROGRAM MAIN
   C This program has been tested using FORTRAN77 to compile the executable

   INTEGER  mode,buff,size,m,I,D(8)
   REAL BBB, C
   REAL YY(4)
   REAL*8 ZZ2,Z,FZZZ
   REAL*8 Y(4)
   CHARACTER STR*20
```
CHARACTER STR2*7 /'PROGRAM'/
ZZZ = 0
X = 333.3
mode = 1

C Load the shared code lib
WRITE (*,*) "Trying to load CD_dll"
CALL FAPI_dlopen("./CD_dll.sc")
WRITE (*,*) "CD_dll loaded"

C Set the API string compatability mode
WRITE (*,*) "Testing FAPI_setStringMode"
CALL FAPI_setStringMode(mode)

C loads the command arguments from the CD_in.file and initializes the model
WRITE (*,*) "Testing FAPI_init"
CALL FAPI_init("./CD_in.file")

C test set and get integers
WRITE (*,*) "Testing FAPI_getI(intrenalElem.pl,m)"
CALL FAPI_getI("internalElem.pl",m)
WRITE (*,*) m
m = 2

WRITE (*,*) "Testing FAPI_setI(intrenalElem.pl,m)"
CALL FAPI_setI("internalElem.pl",m)
WRITE (*,*) m
m = 0

WRITE (*,*) "Testing FAPI_getI(intrenalElem.subE.se_pl,m)"
CALL FAPI_getI("internalElem.subE.se_pl",m)
WRITE (*,*) m
m = 0

WRITE (*,*) "Testing FAPI_setI(intrenalElem.subE.se_pl,m)"
CALL FAPI_setI("internalElem.subE.se_pl",m)
WRITE (*,*) m
m = 7

WRITE (*,*) "Testing FAPI_getI(intrenalElem.subE.se_pl,m)"
CALL FAPI_getI("internalElem.subE.se_pl",m)
WRITE (*,*) m
m = 0

WRITE (*,*) "Testing FAPI_getI1D"
CALL FAPI_getI1D("inumId",D,buff,size)
WRITE (*,*) D(1)
WRITE (*,*) D(2)
WRITE (*,*) D(3)
WRITE (*,*) D(4)
WRITE (*,*) D(5)
WRITE (*,*) D(6)
WRITE (*,*) D(7)
WRITE (*,*) D(8)

WRITE (*,*) "Testing FAPI_setIIDentry"
I = 99
buff = 2
CALL FAPI_setIIDentry("inumId",buff,I)
buff = 8
size = 8
CALL FAPI_getIID("inumId",D,buff,size)
WRITE (*,*) D(1)
WRITE (*,*) D(2)
WRITE (*,*) D(3)
WRITE (*,*) D(4)
WRITE (*,*) D(5)
WRITE (*,*) D(6)
WRITE (*,*) D(7)
WRITE (*,*) D(8)

WRITE (*,*) "Testing FAPI_getIID"
CALL FAPI_getIID("inumId",D,buff,size)
WRITE (*,*) D(1)
WRITE (*,*) D(2)
WRITE (*,*) D(3)
WRITE (*,*) D(4)
WRITE (*,*) D(5)
WRITE (*,*) D(6)
WRITE (*,*) D(7)
WRITE (*,*) D(8)

WRITE (*,*) "Testing FAPI_setIID"
D(1) = 8
D(2) = 7
D(3) = 6
D(4) = 5
D(5) = 4
D(6) = 3
D(7) = 2
D(8) = 1
CALL FAPI_setIID("inumId",D,size)
D(1) = 0
D(2) = 0
D(3) = 0
D(4) = 0
D(5) = 0
D(6) = 0
D(7) = 0
D(8) = 0
CALL FAPI_getIID("inumId",D,buff,size)
WRITE (*,*) D(1)
WRITE (*,*) D(2)
WRITE (*,*) D(3)
WRITE (*,*) D(4)
WRITE (*,*) D(5)
WRITE (*,*) D(6)
WRITE (*,*) D(7)
WRITE (*,*) D(8)
WRITE (*,*) "Testing FAP_setIlDentry"
I = 99
buff = 3
size = 8
CALL FAPI_setIlDentry("inuml1",buff,I)
buff = 8
size = 8
CALL FAPI_getIlD("inuml1",D,buff,size)
WRITE (*,*) D(1)
WRITE (*,*) D(2)
WRITE (*,*) D(3)
WRITE (*,*) D(4)
WRITE (*,*) D(5)
WRITE (*,*) D(6)
WRITE (*,*) D(7)
WRITE (*,*) D(8)

WRITE (*,*) "Testing FAPI_getIlDentry"
CALL FAPI_getIlDentry("inuml1",size,I)
WRITE (*,*) I

buff = 0
size = 0

C test set and get doubles

WRITE (*,*) "Testing FAPI_getD"
CALL FAPI_getD("rnum",222)
WRITE (*,*) 222

WRITE (*,*) "Testing FAPI_setD"
CALL FAPI_setD("rnum",X)
WRITE (*,*) X

WRITE (*,*) "Testing FAPI_getD"
CALL FAPI_getD("rnum",222)
WRITE (*,*) 222

X = 777.7
WRITE (*,*) "Testing FAPI_setD"
CALL FAPI_setD("rnum",X)
WRITE (*,*) X

WRITE (*,*) "Testing FAPI_getD"
CALL FAPI_getD("rnum",222)
WRITE (*,*) 222

WRITE (*,*) "Testing FAPI_getD1D"
buff = 4
size = 4
CALL FAPI_getD1D("fnum1d",Y,buff,size)
WRITE (*,*) Y(1)
WRITE (*,*) Y(2)
WRITE (*,*) Y(3)
WRITE (*,*) Y(4)

WRITE (*,*) "Testing FAPI_setD1Dentry"
buff = 3
X = 8.8
CALL FAPI_setD1Dentry("fnum1d",buff,X)
WRITE (*,*) X
WRITE (*,*) "Testing FAPI_getD1D"
buff = 4
size = 4
CALL FAPI_getD1D("fnum1d", Y, buff, size)
WRITE (*,*) Y(1)
WRITE (*,*) Y(2)
WRITE (*,*) Y(3)
WRITE (*,*) Y(4)
WRITE (*,*) "Testing FAPI_setD1D"
buff = 4
size = 4
Y(1) = 9.9
Y(2) = 8.8
Y(3) = 7.7
Y(4) = 6.6
CALL FAPI_setD1D("fnum1d", Y, size)
CALL FAPI_getD1D("fnum1d", Y, buff, size)
WRITE (*,*) Y(1)
WRITE (*,*) Y(2)
WRITE (*,*) Y(3)
WRITE (*,*) Y(4)
WRITE (*,*) "Testing FAPI_getD2Dentry"
buff = 1
size = 1
CALL FAPI_getD2Dentry("fnum2d", buff, size, X)
WRITE (*,*) X
buff = 1
size = 2
CALL FAPI_getD2Dentry("fnum2d", buff, size, X)
WRITE (*,*) X
buff = 1
size = 3
CALL FAPI_getD2Dentry("fnum2d", buff, size, X)
WRITE (*,*) X
buff = 2
size = 1
CALL FAPI_getD2Dentry("fnum2d", buff, size, X)
WRITE (*,*) X
buff = 2
size = 1
CALL FAPI_getD2Dentry("fnum2d", buff, size, X)
WRITE (*,*) X
buff = 2
size = 2
CALL FAPI_getD2Dentry("fnum2d", buff, size, X)
WRITE (*,*) X
buff = 2
size = 3
CALL FAPI_getD2Dentry("fnum2d", buff, size, X)
WRITE (*,*) X
buff = 2
size = 1
X = 5.5
CALL FAPI_setD2Dentry("fnum2d", buff, size, X)
WRITE (*,*) X
WRITE (*,*) "Testing FAPI_getD2Dentry"
buff = 1
size = 1
CALL FAPI_getD2Dentry("fnum2d", buff, size, X)
WRITE (*,*) X
buff = 1
size = 2
CALL FAPI_getD2Dentry("fnum2d",buff,size,X)
WRITE (*,*) X
buff = 1
size = 3
CALL FAPI_getD2Dentry("fnum2d",buff,size,X)
WRITE (*,*) X
buff = 2
size = 1
CALL FAPI_getD2Dentry("fnum2d",buff,size,X)
WRITE (*,*) X
buff = 2
size = 2
CALL FAPI_getD2Dentry("fnum2d",buff,size,X)
WRITE (*,*) X
buff = 2
size = 3
CALL FAPI_getD2Dentry("fnum2d",buff,size,X)
WRITE (*,*) X

C test set and get floats

WRITE (*,*) "Testing FAPI_getF"
CALL FAPI_getF("rnum",BBB)
WRITE (*,*) BBB

C = 999.9
WRITE (*,*) "Testing FAPI_setF"
CALL FAPI_setF("rnum",C)
WRITE (*,*) C

WRITE (*,*) "Testing FAPI_getF"
CALL FAPI_getF("rnum",BBB)
WRITE (*,*) BBB

C = 444.4
WRITE (*,*) "Testing FAPI_setF"
CALL FAPI_setF("rnum",C)
WRITE (*,*) C

WRITE (*,*) "Testing FAPI_getF"
CALL FAPI_getF("rnum",BBB)
WRITE (*,*) BBB

WRITE (*,*) "Testing FAPI_getF1D"
buff = 4
size = 4
CALL FAPI_getF1D("fnum1d",YY,buff,size)
WRITE (*,*) YY(1)
WRITE (*,*) YY(2)
WRITE (*,*) YY(3)
WRITE (*,*) YY(4)

WRITE (*,*) "Testing FAPI_setF1Dentry"
buff = 3
C = 8.7
CALL FAPI_setF1Dentry("fnum1d",buff,C)
C = 0
CALL FAPI_getF1Dentry("fnum1d",buff,C)
WRITE (*,*) C
WRITE (*,*) "Testing FAPI_getF1D"
buffer = 4
size = 4
CALL FAPI_getF1D("fnum1d",YY,buffer,size)
WRITE (*,*) YY(1)
WRITE (*,*) YY(2)
WRITE (*,*) YY(3)
WRITE (*,*) YY(4)

WRITE (*,*) "Testing FAPI_getF2Dentry"
buffer = 1
size = 1
CALL FAPI_getF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C
buffer = 2
size = 2
CALL FAPI_getF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C
buffer = 1
size = 3
CALL FAPI_getF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C
buffer = 2
size = 1
CALL FAPI_getF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C
buffer = 2
size = 2
CALL FAPI_getF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C
buffer = 2
size = 3
CALL FAPI_getF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C

WRITE (*,*) "Testing FAPI_setF2Dentry"
buffer = 2
size = 1
C = 7.8
CALL FAPI_setF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C

WRITE (*,*) "Testing FAPI_getF2Dentry"
buffer = 1
size = 1
CALL FAPI_getF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C
buffer = 2
size = 2
CALL FAPI_getF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C
buffer = 2
size = 2
CALL FAPI_getF2Dentry("fnum2d",buffer,size,C)
WRITE (*,*) C

NPSS Model Delivery Development Kit
buff = 2
size = 3
CALL FAPI_getF2Dentry("fnum2d",buff,size,C)
WRITE (*,*) C

C test string stuff
size = 6
WRITE (*,*) "Testing FAPI_getS"
CALL FAPI_getS("s",STR,size)
WRITE (*,*) STR

WRITE (*,*) "Testing FAPI_setS"
WRITE (*,*) "using a string literal"
size = 7
CALL FAPI_setS("s","TESTING",size)
CALL FAPI_getS("s",STR,size)
WRITE (*,*) STR
WRITE (*,*) "using a FORTRAN string"
CALL FAPI_setS("s",STR2,size)
CALL FAPI_getS("s",STR,size)
WRITE (*,*) STR

buff = 1
WRITE (*,*) "Testing FAPI_getS1Dentry"
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR
buff = 2
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR
buff = 3
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR
buff = 4
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR
buff = 5
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR

buff = 2
size = 7
WRITE (*,*) "Testing FAPI_setS1Dentry"
WRITE (*,*) "using a string literal"
CALL FAPI_setS1Dentry("s1d",buff,"TESTING",size)
buff = 1
WRITE (*,*) "Testing FAPI_getS1Dentry"
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR
buff = 2
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR
buff = 3
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR
buff = 4
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR
buff = 5
CALL FAPI_getS1Dentry("s1d",buff,STR,size)
WRITE (*,*) STR

buff = 3
size = 7
WRITE (*,*) "Testing FAPI_setSlDentry"       
WRITE (*,*) "using a FORTRAN string"       
CALL FAPI_setSlDentry("sld",buff,STR2,size)
  buffer = 1
WRITE (*,*) "Testing FAPI_getSlDentry"       
CALL FAPI_getSlDentry("sld",buff,STR,size)
WRITE (*,*) STR
  buffer = 2
CALL FAPI_getSlDentry("sld",buff,STR,size)
WRITE (*,*) STR
  buffer = 3
CALL FAPI_getSlDentry("sld",buff,STR,size)
WRITE (*,*) STR
  buffer = 4
CALL FAPI_getSlDentry("sld",buff,STR,size)
WRITE (*,*) STR
  buffer = 5
CALL FAPI_getSlDentry("sld",buff,STR,size)
WRITE (*,*) STR

C testing more stuff
WRITE (*,*) "Testing changed zzz to rnum"

C read initial values
CALL FAPI_getD("rnum", FZZZ)
WRITE (*, *) FZZZ

C set and get
FZZZ = 333.3
WRITE (*, *) "Setting zzz to 333.3"
CALL FAPI_setD("rnum", FZZZ)
CALL FAPI_getD("rnum", FZZZ)
WRITE (*, *) FZZZ

C test reading an external file
WRITE (*,*) "Testing FAPI_parseFile"
CALL FAPI_parseFile("./modelfile2")
WRITE (*,*) "Testing FAPI_run"
CALL FAPI_run()
WRITE (*,*) "Testing FAPI_terminate"
CALL FAPI_terminate()
WRITE (*,*) "Testing FAPI_dlclose"
CALL FAPI_dlclose()

END

FORTRAN example 2, a basic test (filename: runcd2.f);

PROGRAM MAIN
C This program has been tested using FORTRAN77 to compile the executable
  !$HP$ ALIAS fapi_dlopen = 'FAPI_dlopen'
  !$HP$ ALIAS fapi_setStringMode = 'FAPI_setStringMode'
  !$HP$ ALIAS fapi_init = 'FAPI_init'
  !$HP$ ALIAS fapi_dlclose = 'FAPI_dlclose'
INTEGER  mode
mode = 1

C Load the shared code lib

WRITE (*,*) "Trying to load CD_dll"
CALL  FAPI_dlopen("./CD_dll.sc")
WRITE (*,*) "CD_dll loaded"

C Set the API string compatibility mode

WRITE (*,*) "Testing FAPI_setStringMode"
CALL  FAPI_setStringMode(mode)

C loads the command arguments from the CD_in.file and initializes the model

WRITE (*,*) "Testing FAPI_init"
CALL  FAPI_init("./CD_in.file")

WRITE (*,*) "Testing FAPI_terminate"
CALL  FAPI_terminate()

WRITE (*,*) "Testing FAPI_dlclose"
CALL  FAPI_dlclose()

END

The "C" example (filename: runcdl.c);

#include <stdio.h>
#include <stdlib.h>
#include "APITranslate.h"

int main (int argc[], char* argv[]){
    FAPI_dlopen_(argv[1]);
    FAPI_init_(argv[2]);
    FAPI_terminate_();
    FAPI_dlclose_();
    return 0;
}

Makefile UNIX Example:

TOP = .

# compilers
AC = cc
CC = CC
FC = f77

ifeq ($(NPSS_CONFIG),hp)
AC = cc
CC = aCC -AA
FC = f90
endif

ifeq ($(NPSS_CONFIG),linux)
AC = cc
CC = gcc
endif
FC = g77
endif

# compiler flags
CFLAGS = -c -KPIC -w
CXXFLAGS= -c -KPIC
FFLAGS = -KPIC -Bdynamic -U
ifeq ($(NPSS_CONFIG),sgi)
CFLAGS = -DSGI_DLL -c -LANG:std -KPIC -w
CXXFLAGS= -DSGI_DLL -c -LANG:std -KPIC
FFLAGS = -DSGI_DLL -LANG:std -KPIC -Bdynamic -U
endif
ifeq ($(NPSS_CONFIG),hp)
CFLAGS = -c -g +Z -DUSE_DLL -w -BAportable
CXXFLAGS= -c +Z
FFLAGS = -K -g +Z -DUSE_DLL -dynamic
endif
ifeq ($(NPSS_CONFIG),linux)
CFLAGS = -c -fPIC -w
CXXFLAGS= -c -fPIC
FFLAGS = -ff77 -ff2c -fcase-preserve -fno-automatic -finit-local-zero -fno-second-underscore -fPIC
endif

# linker stuff
LOPTS = -L -lm -ldl
CLOPTS = -L -ldl -lcstd
ifeq ($(NPSS_CONFIG),sgi)
LOPTS = -DUSE_DLL -L -lm -ldl
CLOPTS = -woff 15 -DUSE_DLL -L -ldl -lc
endif
ifeq ($(NPSS_CONFIG),hp)
LOPTS = -L -lm -lP90 -lcurses -lc -lcl -lstd -lcstd -ldld
CLOPTS = -L -ldld
endif
ifeq ($(NPSS_CONFIG),linux)
LOPTS = -L -lm -lc -ldl
CLOPTS = -L -lc -ldl
endif

NAME = runcd.$(NPSS_CONFIG)
NAME1 = runcd1.$(NPSS_CONFIG)
NAME2 = runcd2.$(NPSS_CONFIG)
NAME3 = ftest.$(NPSS_CONFIG)

# SRC to compile
CSRC = APITranslate.c
FSRC = runcd.f
FDIR = runcd1.c
FSRC2 = runcd2.f
FEXTRA = ftest.f

OBJ = APITranslate.o
COBJ = runcd1.o

runcd:$(FSRC)
   $(CC) $(CXXFLAGS) $(CSRC)
$(FC) -o $(NAME) $(FFLAGS) $(OBJ) $(LOPTS) $(FSRC)

runcd1:$ (CSRC1)
  $(CC) $(CFLAGS) $(CSRC)
  $(CC) $(CFLAGS) $(CSRC1)
  $(CC) -o $(NAME1) $(OBJ) $(CLOPTS)

runcd2:$ (FSRC2)
  $(CC) $(CXXFLAGS) $(CSRC)
  $(FC) -o $(NAME2) $(FFLAGS) $(OBJ) $(LOPTS) $(FSRC2)

ftest:$ (FEXTRA)
  $(CC) $(CXXFLAGS) $(CSRC)
  $(FC) -o $(NAME3) $(FFLAGS) $(OBJ) $(LOPTS) $(FEXTRA)

clean:
  rm -f $(NAME)
  rm -f $(NAME1)
  rm -f $(NAME2)
  rm -f $(NAME3)
  rm -f runcd1.o
  rm -f APITranslate.o

clean_runcd:
  rm -f $(NAME)
  rm -f APITranslate.o

clean_runcd1:
  rm -f $(NAME1)
  rm -f runcd1.o
  rm -f APITranslate.o

clean_runcd2:
  rm -f $(NAME2)
  rm -f APITranslate.o

clean_ftest:
  rm -f $(NAME3)
  rm -f APITranslate.o

Makefile Windows example:

TOP = .
FFLAGS = /nologo /warn:nofileopt /assume:underscore /iface:nomixed_str_len_arg
       /iface:cref /debug:none /names:as_is

NAME1 = runcd1.$(NPSS_CONFIG).exe
NAME2 = runcd2.$(NPSS_CONFIG).exe
NAME3 = ftest.$(NPSS_CONFIG).exe

#SRC to compile
CSRC   = APITranslate.c
CSRC1  = runcd1.c
FSRC2  = runcd2.f
FEXTRA = ftest.f

OBJ    = APITranslate.obj
COBJ   = runcd1.obj

runcd1:$ (CSRC1)
  cl /c $(CSRC)
5.4 Building a Customer Simulation Using the FORTRAN API

A customer simulation using the NPSS FORTRAN API contains an ARP4191 compliant interface. The customer’s FORTRAN application links the NPSS simulation, passes data to and from the NPSS simulation via FORTRAN COMMONs and executes one or more of a specified set of function calls to control the model. The model developer defines the variables that will be accessible to the customer.

5.4.1 FORTRAN API Description

The NPSS FORTRAN API is a layer over the Native NPSS API and shares the security model and creation process. The baseline functions for controlling the model are as follows.

`int INITMD(char *encModel)`

This loads the encrypted model file `encModel`, turns on “secure” mode, copies the values of the input variables from the simulation into the input commons, and prints the proprietary/classified cover page. This function must be called before any of the other API functions will work.

`int PRTNOT()`

This prints the proprietary/classified cover page. It is called from INITMD automatically.

`int STOREI()`

This copies the values of the input common variables into internal storage for later use by the RESETI function. This function is called in INITMD for storage of the design point values. Calling this function will destroy all previous values of these variables.

`int RESETI()`

This copies the values of the previously stored input common variables into the input commons.

`int RUNMD()`

This zeros out all the output common variables, maps the input common values into their respective model variables, runs the NPSS simulation, maps the model variables into their respective output commons, and returns control to the FORTRAN program.

5.4.2 Security Model

See Section 5-1, Security Model and Access Control, for detailed information on the security model.
Complete access is limited to only those variables in the customer commons. Note, however, that the FORTRAN API allows only read access to all simulation output variables and only write access to all simulation input variables. If the customer attempts to set an output variable, the run command will set it to zero and the model will most likely set it to some other value. Input variables modified by the model will not get mapped back to the input commons.

### 5.4.3 Creation Process

There are two variations on the creation process. In the first, the NPSS input file is encrypted and compiled into the executable. Therefore, every time this particular executable is run, it will be initialized with the same encrypted model. If the model needs to be changed, the executable must be rebuilt. This is the default creation process. In the second variation, the NPSS input file is encrypted and stored in a file external to the executable. This file is specified by providing CDgen with the “–e” option followed by a filename in which to store the encrypted model. By supplying the “–k” option followed by a filename, the encryption key can also be written to an external key file. In this case, a new model file can be created by the customer deck generator by encrypting the new model with the same key that is stored in the executable. This is done by using the external key file (generated by CDgen when given the “–k” option followed by a filename) in the scramble program to produce a new encrypted file. Both variations provide the capability to read in multiple encrypted and non-encrypted files after being set up with the initial model. Again, all encrypted files must be encrypted using the same key that is stored in the executable.

The following steps assume that the model developer has all of the necessary NPSS infrastructure libraries available in non-debug form, i.e., libraries built with code compiled without the –g option.

1) **Create a Model Input File**

A top level model file, written in NPSS interpreter syntax, which defines the model and includes references to any macros or run commands needed to create a “designed” model must be supplied. This file with all of the #include commands will be expanded. It will be encrypted and either compiled into the executable or stored in a file external to the executable, depending on the command line options given to CDgen. In addition to combining all of the input into one string, all comments and unnecessary white space will be removed.

2) **Create a Customer Common File**

You will need to generate a file containing all the FORTRAN COMMONs needed. CDgen expects that ARP4191 COMMON requirements are met. If additional COMMONs are included, their names must comply with the rule that all input commons have to end with the letters “IN” and no output commons can end with the letters “IN.”

3) **Create a Proprietary/Classification Notice File**

Provide a file containing the text of the cover page that will be printed at the top of the output file every time the model is initialized or the PRTNOT function is called.

4) **Create an Alias File**

You will have to supply an alias file which contains a list of the model variables and their respective FORTRAN COMMON names. You may also define initialization values for input variables in this file that are not passed to the simulation, by using the keyword “dummy” in place of the real simulation name. This initialization value will be stored in the input common and in the internal storage by the STOREI function. Any line in the alias file beginning with a “#” will be ignored. The alias file has the following format:

```
<real simulation name> <FORTRAN COMMON name>
<real simulation name> <FORTRAN COMMON name>
<real simulation name> <FORTRAN COMMON name>
dummy <FORTRAN COMMON name><optional initialization value>
... 
<real simulation name> <FORTRAN COMMON name>
```
NOTE: This file no longer controls which objects are visible to the end user. See Section 5 to determine how to control access to your model.

5) Create a Sample Print Routine File
Provide a file containing a sample FORTRAN print routine for checkout of the model on the customer’s computer system. The routine must be called SUBROUTINE PRTMD(). It should contain an INCLUDE statement for the customer common file and print the data to the file defined by the unit number in the FORTRAN variable NOUT.

6) Use CDgen to create C++ files
CDgen takes the model file, the customer commons file, the notice file, the alias file, and the creation permission file and uses them to generate C++ code necessary for the setup of the model. The following C++ files are generated by CDgen for FORTRAN interfaces:

- setupCustomerModel.C // loads the model
- api.H // creates the C structure overlays for the FORTRAN COMMONs.
- map.C // creates all the mapping functions for the interface.

The following FORTRAN files are also generated:

- blockdata.f // initializes the “dummy” variables and all FORTRAN strings.
- sampleMain.f // sample driver main program.

CDgen has the following command line options:

- `-m <model file>` // specifies the name of the file containing the customer model
- `-a <alias file>` // specifies the name of the alias file
- `-e <encrypted file>` // specifies the filename in which to store the encrypted model
  // (default: model.enc)
- `-k <key file>` // specifies the filename in which to store the encryption key
  // (default: none)
- `-d` // specifies that the customer deck will be built with the
  // capability to load in DLMs
- `-f <customer common file>` // specifies the name of the customer commons file
- `-n <notice file>` // specifies the name of the notice data file

For example:

```
CDgen.sun -m modelfile -a aliasfile
    -f customercommons.i -n notice.dat
```

Inclusion of the `-f` option automatically invokes the FORTRAN API creation.

Explanation of Command Line Options (-e, -k, -d) Used with CDgen
See the earlier section on Native NPSS API for details on these CDgen command line options.

Examples:

```
CDgen.sun -m modelfile -a aliasfile -f customercommons.i -n notice.dat
```
This command causes CDgen to create a customer deck where the encrypted model file will compiled into the executable. It will also be stored externally in a file called model.enc. The key will NOT be stored external to the customer deck, and the capability to load DLMs will NOT be included.

`CDgen.sun -m modelfile -a aliasfile -e encModel -k keyfile -d -f customercommons.i -n notice.dat`

This command causes CDgen to create a customer deck where the encrypted model will be stored in `encModel`. The key will be stored in the file `keyfile` and also compiled into the customer deck, and CDgen will search for DLMs using the `NPSS_PATH` environment variable search path.

The customer deck generator can encrypt other model files for use with this executable by running them through the `scramble` program (which is delivered with the Devkit) and providing the saved key filename, for example:

```
scramble.sun keyfile modelfile2 newEmodel
```

In the above example, `modelfile2` is a new unencrypted model file, and `newEmodel` is the encrypted version of that file.

**8) Compile and Link**

Compile your driver program and the files generated by CDgen and link them with any necessary NPSS libraries.

**NOTE:** Make sure that the libraries are non-debug libraries.
6 Testing An Internal Component

6.1 General Guidelines for Testing an Internal Component

Internal components and their subelements must be tested at the component level before being integrated with the NPSS software. The purpose of the component tests is to verify that the components are behaving correctly, that they produce the correct numerical results, and that they interact with their subelements correctly. Thus, the goal of the component tests is to verify the unique attributes of each component not the generic capabilities of the NPSS Element API.

Elements should be run standalone as a single-element engine. Subelements must be run with their corresponding elements.

Several types of tests should be conducted on the components and their corresponding subelements.

- The first test should be a numerical validation of the element without any subelements. This test is done to verify the element functionality in isolation before the subelements are added. The element should be run in all modes, and all element-specific errors should be produced.

- The next set of tests will be run with the element and subelement together because there is no easy way to isolate the subelement without using an element. The element/subelement should be run in all modes, and all component-specific errors should be produced. When running in off-design mode, the generation of solver error terms should be verified if applicable.

- All numerical results should be matched with an independent source of data (preferably a hand calculation). Note that some tests may not involve any subelements and will include only the first set of tests. Some tests will involve running new subelements with already tested elements and will involve only the second set of tests.

- A Test Plan detailing the tests to be conducted should be developed using the FirstOrderLag.tst file (see the following example) as a guide. A self-documented input file should be created using NPSS comment blocks. Include all test descriptions and hand calculations. There will not be a separate test plan and input file. The test output will also be self-documenting (see FirstOrderLag.tst.out for a sample output file). The results from the hand calculations will output next to the code output. In addition, the results from the code will be compared to the hand calculations. A PASS/FAIL indicator will be presented for each individual test as well as for the test as a whole.

In addition to relevant numerical results, the following items should be included in the generated output:
- test name
- platform
- run date
- tester name
- version of the NPSS software used in test
- pass/fail status.

The input and output files should conform to the following convention:
- input file name: `ElementName.tst`
- output file name: `ElementName.tst.out`
All tests for an element and all its corresponding subelements may be combined into a single test.

### 6.2 Sample Plan Test

The following is a sample test plan for the First Order Lag, which is an internal subelement in the NPSS. The Duct element (used in previous examples) is not presented here because the test plan/procedures were revised after the Duct was tested. This test plan is part of an input file and generates output that documents the testing, thereby eliminating the need for a separate test report. This is the method the NASA development team uses to test elements and subelements.

```%
FIRST ORDER LAG TEST   (Input Filename: FirstOrderLag.tst)
```

1. Introduction

1.1 Identification and Scope

The purpose of the test is to verify that the FirstOrderLag subelement is functioning correctly. The goal of this test is to verify the unique attributes of the FirstOrderLag and not the generic capabilities of the NPSS system.

1.2 Organization

2. Applicable Documents

2.1 Reference Documents

NA

2.2 Information Documents

NA

2.3 Parent Documents

The NPSS Software Requirements Specification Document describes the full requirements of NPSS.

The NPSS Systems Analysis Document and the NPSS Systems Design Document describe the general NPSS framework and usage. They show how the different elements and subelements will interact with the NPSS system.

The NPSS/NICE Verification and Validation Plan describe the general NPSS/NICE testing procedures.

The NPSS FirstOrderLag Engineering Specification describes the functionality of the FirstOrderLag.
3 Test Plan

3.1 Test Overview

This test will be run by inserting the FirstOrderLag into a user-supplied subelement. The element will be run from 0 to 10 seconds. The first order lag will lag an input that traces the time in seconds.

3.2 Test Environment

This test will be run on a Sun workstation

3.3 Traceability

The FirstOrderLag is part of the controls toolbox

4.0.1 Test Procedures

Run NPSS on this input file redirecting the standard output to the file test.out. NPSS will report at the bottom of the test.out file if it passed the test or not.

4.0.2 Test Environment

The following files are validated by this test:
   FirstOrderLag.C
   FirstOrderLag.H

*/

//create model for test

//flag indicating success or failure
int pass = 0;

class FOL extends Element {

}

Model FOLTest {

Element FOL myLag {
   real y, output;

   Subelement FirstOrderLag tomLag {
      //FOL tracks y
      lagIn = "y";
      tau = 2;
   }
}
void preexecute() {
    // y goes with time
    y = time;
    output = tomLag.lagOut;
}

void postexecute()
    if ( time > 9.91 )
    cout << "time = " << time << " " << "input = " << myLag.y << " " << "output = " << myLag.output << "\n";
}

setOption("solutionMode", "TRANSIENT");
transient.stopTime = 10.0;
transient.baseTimeStep = .05;

/*
4.1 STEP Test

The purpose of this test is to check the step response of the
FirstOrderLag. In this case the FirstOrderLag is run for ten seconds.
Verification is accomplished by examining the calculations for the
last time step.

Hand Calculations for Step Function

\[
\text{output} = \text{outHis.prevValue()} + \left( \text{inpHis.currentValue()} - \text{outHis.prevValue()} \right) \times \left( 1 - \exp\left(\frac{-\text{outHis.dt()}}{\text{tau}}\right)\right);
\]

\[
\text{output} = \frac{7.98854 + (10.00 - 7.98854) \times (1 - \exp(-.05/2))}{1 - \exp(-.05/2)}
\]

\[
\text{output} = 8.0382
\]
*/
cout << "\n\nSTEP FUNCTION TEST\n\n";
cout << "Output last two time steps to determine input for hand calc\n";
run();
cout << "Output from hand calculation is 8.0382\n" << "Output from code is " << myLag.output << "\n";
if ( abs ( myLag.output - 8.0382 )/ 8.0382 < .0001 ) { cout << "PASSED STEP Test\n";
} else {
    pass = 1;
    cout << "FAILED STEP Test\n";
}
*/
4.2 Ramp Test

The purpose of this test is to check the Ramp response of the FirstOrderLag. In this case the FirstOrderLag is run for ten seconds. Verification is accomplished by examining the calculations for the last time step.

Hand Calculations for Ramp Function

\[
\text{output} = \text{outHis.prevValue} + \\
( \text{inpHis.prevValue} - \text{outHis.prevValue} ) * ( 1 - \exp ( - ( \text{outHis.dt} / \tau) )) + \\
( \text{inpHis.currentValue} - \text{inpHis.prevValue} ) * \\
( 1 - \tau / \text{outHis.dt} ) * ( 1 - \exp ( - ( \text{outHis.dt} / \tau) ));
\]

\[
\text{output} = 7.96382 + (9.95 - 7.96382) * (1 - \exp(-.05/2)) + (10.0 - 9.95) * (1 - (2/.05) * (1 - \exp(-.05/2)))
\]

output = 8.0136816

*/

cout << "\n\nRAMP FUNCTION TEST\n\n\n";

setOption( "switchForm", "RAMP");
cout << "Output last two time steps to determine input for hand calc\n";

time = 0;
myLag.tomLag.lagOut = 0.;
myLag.y = 0;
initializeHistory();
run();

cout << "Output from hand calculation is 8.013816\n" << "Output from code is " << myLag.output << "\n";
if ( abs ( myLag.output - 8.013816 )/8.013816 < .0001 ) {
    cout << "PASSED RAMP Test\n";
} else {
    pass = 1;
    cout << "FAILED RAMP Test\n";
}

*/

4.3 Derivative Test

The purpose of this test is to check the derivative calculation of the FirstOrderLag. In this case the FirstOrderLag is run for ten seconds. Verification is accomplished by examining the calculations for the last time step.

Hand Calculations for Derivative

\[
_doutputdt = ( \text{inpHis.current} - \text{output} ) / \tau;
_doutputdt = ( 10 - 8.013816 ) / 2;
_outputdt = .99326;
\]
cout << "\n\nDERIVATIVE TEST\n\n\n";

setOption( "switchForm", "RAMP");
cout << "Output last two time steps to determine input for hand calc\n";

    time = 0;
    myLag.tomLag.lagOut = 0.;
    myLag.y = 0;
    initializeHistory();
    run();

    cout << "Output from hand calculation is .99326\n" << "Output from code is \n" <<

    myLag.tomLag.dlagOutdt << "\n";

    if ( abs ( myLag.tomLag.dlagOutdt - .99326 )/ .99326 < .0001 ) {
    cout << "PASSED DERIVATIVE Test\n";
    }
    else {
    pass = 1;
    cout << "FAILED DERIVATIVE Test\n";
    }

    /*

4.4 Steady-State Test

The purpose of this test is to check the steady-state response of the FirstOrderLag. Since the driving value of y is currently 10, the steady-state response should return 10.

    */

    cout << "\n\n STEADYSTATE TEST\n\n\n";
    cout << "Output last two time steps to determine input for hand calc\n";
    cout << "Output from hand calculation is 10.00\n" << "Output from code is \n" <<

    myLag.output << "\n";

    if ( abs ( myLag.tomLag.lagOut - 10 )/ 10 < .0001 ) {
    cout << "PASSED STEADYSTATE Test\n";
    }
    else {
    pass = 1;
    cout << "FAILED STEADYSTATE Test\n";
    }
setOption( "switchForm", "ADD_SOLVER" );

system( "cp test.out regression.out" );

cout << "\n\nTest Summary:\n";
cout << VERSION << " " << USER << " " << date << " " <<"\n";
system( "uname -a" );
if ( pass == 0 ){
    cout << "PASSED FirstOrderLag Test\n";
} else {
    cout << "FAILED FirstOrderLag Test (see above)\n";
}

### 6.3 Sample Output from Component Testing

The following is the output generated from the testing done using the FirstOrderLag.tst input file.

FIRSTORDERLAG TEST OUTPUT  (Filename: FirstOrderLag.tst.out)

*  

**STEP FUNCTION TEST**

Output last two time steps to determine input for hand calc

<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.95</td>
<td>9.95</td>
<td>7.98854</td>
</tr>
<tr>
<td>9.95</td>
<td>9.95</td>
<td>7.98854</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8.0382</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8.0382</td>
</tr>
</tbody>
</table>

Output from hand calculation is 8.0382

Output from code is 8.0382

PASSED STEP Test

**RAMP FUNCTION TEST**

Output last two time steps to determine input for hand calc

<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.95</td>
<td>9.95</td>
<td>7.96382</td>
</tr>
<tr>
<td>9.95</td>
<td>9.95</td>
<td>7.96382</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8.01348</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8.01348</td>
</tr>
</tbody>
</table>

Output from hand calculation is 8.01348

Output from code is 8.01348

PASSED RAMP Test

**DERIVATIVE TEST**

Output last two time steps to determine input for hand calc

<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.95</td>
<td>9.95</td>
<td>7.96382</td>
</tr>
<tr>
<td>9.95</td>
<td>9.95</td>
<td>7.96382</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8.01348</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8.01348</td>
</tr>
</tbody>
</table>

Output from hand calculation is .99326
Output from code is 0.993262
PASSED DERIVATIVE Test

STEADYSTATE TEST

Output last two time steps to determine input for hand calc
time = 10 input = 10 output = 10
Output from hand calculation is 10.00
Output from code is 10
PASSED STEADYSTATE Test

Test Summary:
REL_0.2.0 - Rev: K (CR233)  Tom Lavelle 06/12/98
SunOS golconda 5.5.1 Generic sun4u sparc SUNW,Ultra-2
PASSED FirstOrderLag Test

(Note: This sample test is old, so operating systems and platform names are outdated.)
7 Utilities

7.1 NPSS to C++ Converter

A utility called `npssconvert` will convert the definition of an interpreted NPSS component into a component written in C++. Given an input file containing a declaration of an interpreted component, the converter is used as follows:

```
npssconvert [-v][-ext][-I<include path>][-D<preproc_var=val>][-o <outFileDir>]
[-ho <headerOutDir>][-mo <metaDataOutDir>][-dep <depClassName>]
[-(stable|-nostable)][-targetACL <num>] [-setup <setupFile>]
```
```
inputFile
```

The arguments enclosed in square brackets [] above are optional.

**Command Line Options**

- **-v** Prints the version of the converter.
- **-ext** Designates the desired handling of external variables during conversion. When the **-ext** option is used, all out-of-scope variables encountered during conversion will be assumed to be of type real. The default behavior with out-of-scope variables of unknown types is simply to assume they are from the parent and make no decisions on their types.
- **-I** Adds a directory to the include search path. The user must provide the converter with all of the same include search directories that would be required for NPSS to process the same input file.
- **-D** Defines and optionally sets the value of a preprocessor variable.
- **-o** Specifies output directory for generated files.
- **-ho** Specifies output directory for generated header files (overrides **-o**).
- **-mo** Specifies output directory for generated metadata files (overrides **-o**).
- **-dep** Adds an entry to the “depends” metadata variable for the generated class.
- **-stable** Globally enables the string reference stable configuration optimization.
- **-nostable** Globally disables the string reference stable configuration optimization.
- **-targetACL** Reconfigure instance and class hash tables for a target average chain length of `num`.
- **-setup** Use `setupFile` to set up converter mappings, etc. Default is `converterSetup.ncp`.

Three files will be generated for each class declaration found in `inputFile` or in any of its include files. For example, if the input file contains an interpreted class named `MyElement`, then `MyElement.H`, `MyElement.C` and `MyElement.met` files will be generated. Examples can be found at the end of this appendix.

**User Accessible Mapping Functions**

The following functions may be called from the input file in order to set up additional variable or function mappings that are not built into the converter.

```c
void addVarMap(string parentClassName, string varName, string getFunctName, string setFunctName)
```

For a given variable with name `varName` in a class called `parentClassName`, this function specifies the name of a C++ get function (getFunctName) and optionally a set function (setFunctName) to provide access to the variable from other C++ objects.

```c
void addFunctMap(string parentClassName, string functName, string cppName)
```

For a given function with name `functName` in a class called `parentClassName`, this function specifies the name of a C++ function (cppName) and the name of its C++ equivalent (cppName).
void copyVarFunctMap(string className, string copyClassName)
Copies all of the variable/function mappings for a given class to another class.

void addClassMap(string userClassName, string internalClassName, string includeFileName)
When a user-accessible class name differs from a C++ class name, this function sets up a mapping between the two. It also specifies the name of the include file necessary to use the class.

Converter Variables
The converter has a number of variables that modify the contents of the converter output files. They are as follows:

_HFILE_HEADER This string will appear at the top of all generated .H files.
_CFFILE_HEADER This string will appear at the top of all generated .C files.
_FUNCT_HEADER This string will appear at the top of each function definition in the .C file.
_STD_C_INCLUDES This string should contain any additional include files needed in the .C file.

NOTE: _STD_C_INCLUDES already contains a number of include files that are necessary by default, so you should append to it rather than overwriting it. Note also that include file names in the string must be separated by newlines. Finally, note that the converter variables listed above do not exist in NPSS, so in order to be able to use the interpreted version of the component in a model, you must surround the converter variables with an #ifdef __NPSSCONVERTER__ block so they will be ignored by NPSS. For example:

ife ndef __NPSSCONVERTER__
_STD_C_INCLUDES += "\n#include <InterStageBleedOutPort.H>";
endif

Performance Considerations
Any strings or string arrays that use the - operator should be declared in the interpreted class and not as function local variables. If they are declared local to a function, they will have to re-resolve the objects they point to every time the function is called. However, sometimes such string references really should be local to the function; for instance, when what is referenced depends on what the function is doing for this particular call. If a large number of such references are required within the function, and the model configuration is known not to change within the function, the “stableConfig” optimization may help. This optimization resolves the reference to a local C++ pointer variable whenever the associated string reference variable is modified. Then references to the target are done through the C++ pointer, rather than re-resolving the string reference. This optimization can be controlled on a per-function basis, through the stableConfig function attribute, or globally via the -stable and -nostable command line options.

Any constants should also be declared in the class, and not as function local variables, to prevent them from being initialized each time the function is called.

Setting an option variable’s attribute rewritableValues to FALSE will allow the converter to translate comparisons between the option variable and a constant string as an integer comparison. By default rewritableValues is TRUE, and the converter must use a slower indirect string comparison.

Use of the -targetACL option can help in some situations with components which have a large number of attributes and/or functions. Some care should be used though, since a targetACL close to 1 can cause a significant waste of memory on empty hash chains. Statistics of hash tables that have been reconfigured from their default will be displayed on cout.
General Usage Notes
Do not name any variables with leading and trailing double underscores, for example, __X__. The converter creates internal variables using this naming convention. This could cause name collisions if any component variables use the same convention.

The converter requires an input file called converterSetup.ncp, which is typically located in the same directory as the converter. When running the converter in stand-alone mode, i.e., outside of the NPSS make system, you will have to supply a -I command line option that specifies the directory containing the converterSetup.ncp file. You can specify a special setup file via the -setup option. Typically this file would contain special definitions for the component(s) to be converted, as well as #include “converterSetup.ncp” to get the standard set of definitions. Note that converterSetup.ncp itself uses #include “ConstantsSetup.in” to get definitions for global constants.

Do not set the "ptrType" attribute from anywhere other than the class definition. The ptrType set inside of a member function, for instance, does not get executed immediately. If that ptrType is not executed, then it is never set, and in turn, the pointer ends up not being cast to the proper type within the converted code.

To generate a converted type of RefVarVCStringRef1D*, set ptrType to "*[]";

Do not attempt to use getPathName() on variables such as int and real, which when converted, do not contain the properties in C++ that they do in NPSS. Instead, construct the pathname by getting the path of the variable’s parent and appending "." and the local variable’s name to the end of the path. For example:

```cpp
int test;
string path = ../getPathName() + ".test";
```

Integrating the Component into NPSS
The NPSS make system has been modified to allow files with a .int extension to be converted and compiled automatically. This requires a slightly different Makefile than the one required for a C++ component. Below is an example of a UNIX make file:

```make
TOP = ../..../
include $(TOP)/Components.config
NPSSCONVERTERFLAGS += -I../..../include
LIBNAME = NCPElements
#source to compile
SRC = Compressor.int
#local dirt, this is O.$(MACHTYPE) relative...
DIRT = Compressor.C Compressor.H

include $(MAKEPRJ)
```

The example above will work when the interpreted component is placed in a subdirectory under /NPSS/dev/AirBreathing/Elements or /NPSS/dev/AirBreathing/Subelements. If the component will be placed elsewhere, the value of the TOP variable will have to be modified. Also note that it is very important to define the DIRT variable to contain all of the names of files that the converter will generate. Otherwise, a “make clean” command will not remove the generated files and future makes may not reflect recent changes in the .int file.
Example Converter Input and Output Files

The following are example converter input and output files:

MyElement.int - The Interpreted Class Definition

class MyElement extends Element {
  real x, y, z;

  Socket S_mySocket {
    allowedValues = { "y", "z" }
  }

  void calculate() {
    if(!S_mySocket.isEmpty()) {
      S_mySocket.execute();
    }

    x = TB_myTable(y, z);
  }
}

MyElement.H - The C++ Header File Generated by npssconvert

#ifndef __MyElement_H
#define __MyElement_H

#include <Element.H>
#include <VCStringRef.H>
#include <FunctionRef.H>

class Socket;

class MyElement : public Element {
  
  public:

    MyElement(const NCPStrings name);
    MyElement(const MyElement&);

    virtual VCInterface* clone() { return new MyElement(*this); }

  protected:
    void calculate();

  private:
    void registerRefVariables();

    NCPReal x;
    NCPReal y;
    NCPReal z;

    FunctionRef __TB_myTable;
    Socket *S_mySocket;

};

#endif
#include "MyElement.H"
#include <math.h>
#include <Socket.H>
#include <NCPRefVariable.H>
#include <Socket.H>
#include <UnitNames.H>
#include <StringEngrConstants.H>
#include <SocketTypes.H>
#include <Constants.H>

Register_NPSS_Object(MyElement);

//
//
// MyElement::MyElement(const NCPString& name) :
Element(name),
x(0),
y(0),
z(0),
__TB_myTable("TB_myTable", this)
{
    NCPStringArray1D S_mySocketallowedVals;
    S_mySocketallowedVals.append("y");
    S_mySocketallowedVals.append("z");
    S_mySocket = addSocket("S_mySocket", ",", S_mySocketallowedVals);

    makeInterface("MyElement");
    registerRefVariables();
}

//
//
//
MyElement::MyElement(const MyElement& original) :
Element(original),
x(original.x),
y(original.y),
z(original.z),
__TB_myTable("TB_myTable", this)
{
    S_mySocket = getSocket("S_mySocket");

    registerRefVariables();
}

//
//
void MyElement::registerRefVariables()
{
    VariableBase *tmpvarptr;
    tmpvarptr = new NCPRefVariable<NCPReal>("x", x, this);
    tmpvarptr = new NCPRefVariable<NCPReal>("y", y, this);
    tmpvarptr = new NCPRefVariable<NCPReal>("z", z, this);
}
//
//
//
void MyElement::calculate()
{
    if(!S_mySocket->isEmpty()) {
        S_mySocket->execute();
    }
    x = __TB_myTable.evalReal(y,z);
}

7.2 **NPSS to C++ Table Converter**
Interpreted NPSS Table instances can be converted to C++ code, similar to converting NPSS Elements.

**Usage**

```
npsstable.sun [-single libname] [NPSS options] filename
-single optional; places multiple tables into a single DLM named libname
NPSS options optional; -I,-D, etc. see NPSS command line options
filename required; file to parse
```

The table converter will recurse the passed file and generate a .C file for every table encountered.

**Example**

**MyTable.int**

```
Table MyTable (real x, real w,real y) {
    w = 1 {
        x = 1{
            y = {14.0,15.0,16.0}
            z = {55,66,77}
            zz = {1,2,3}
        }
        x = 2 {
            y = {44.0,55.0}
            z = {155,66}
            zz = {11,22}
        }
    }
    w = 10 {
        x = 3{
            y = {-4.0,5,0,6.0}
            z = {55,66,77}
            zz = {111,222,333}
        }
        x = 8 {
```
Utilities

The table converter will generate the following:

MyTable.C

```c
#include <TableFunction.H>
#include <XYTable.H>
#include <StackedTable.H>

static void MyTable_initializeTable(TableFunction* ptf) {
    NCPRealArray1D ivals;
    NCPRealArray1D dvals;
    Table* tab1;
    tab1 = new Table("w");
    Table* tab10;
    tab10 = new Table("x");
    Table* tab100;
    tab100 = new StackedTable("y");
    ivals.clear();
    ivals.push_back(14);
    ivals.push_back(15);
    ivals.push_back(16);
    ((StackedTable*)tab100)->setIndependents(ivals);
    dvals.clear();
    dvals.push_back(55);
    dvals.push_back(66);
    dvals.push_back(77);
    ((StackedTable*)tab100)->addDependentSet("z", dvals);
    dvals.clear();
    dvals.push_back(1);
    dvals.push_back(2);
    dvals.push_back(3);
    ((StackedTable*)tab100)->addDependentSet("zz", dvals);
    tab10->addSubTable(tab100, 1);
    tab100 = new StackedTable("y");
    ivals.clear();
    ivals.push_back(44);
    ivals.push_back(55);
    ((StackedTable*)tab100)->setIndependents(ivals);
    dvals.clear();
    dvals.push_back(155);
    dvals.push_back(66);
    ((StackedTable*)tab100)->addDependentSet("z", dvals);
    dvals.clear();
    dvals.push_back(11);
    dvals.push_back(22);
    ((StackedTable*)tab100)->addDependentSet("zz", dvals);
    tab10->addSubTable(tab100, 2);
    tab1->addSubTable(tab10, 1);
    tab10 = new Table("x");
    tab100 = new StackedTable("y");
    ivals.clear();
    ivals.push_back(-4);
    ivals.push_back(5);
    ivals.push_back(6);
    ((StackedTable*)tab100)->setIndependents(ivals);
    
```
dvals.clear();
dvals.push_back(55);
dvals.push_back(66);
dvals.push_back(77);

{((StackedTable*)tab100)->addDependentSet("z",dvals);
dvals.clear();
dvals.push_back(111);
dvals.push_back(222);
dvals.push_back(333);

{((StackedTable*)tab100)->addDependentSet("zz",dvals);
	tab10->addSubTable(tab100,3);
	tab100 = new StackedTable("y");
	ivals.clear();
	ivals.push_back(4);
	ivals.push_back(5);
	ivals.push_back(6);

{((StackedTable*)tab100)->setIndependents(ivals);
	ivals.clear();
	dvals.push_back(55);
dvals.push_back(66);
dvals.push_back(77);

{((StackedTable*)tab100)->addDependentSet("z",dvals);
	dvals.clear();
	dvals.push_back(1111);
	dvals.push_back(2222);
	dvals.push_back(3333);

{((StackedTable*)tab100)->addDependentSet("zz",dvals);
	tab10->addSubTable(tab100, 0);
	tab1->addSubTable(tab10, 10);

ptf->setTable(tab1);
ptf->set("x.interp","linear",0);
ptf->set("x.extrap","linear",0);
ptf->set("w.interp","linear",0);
ptf->set("w.extrap","linear",0);
ptf->set("y.interp","linear",0);
ptf->set("y.extrap","linear",0);

NPSSvector<size_t> argIndex;
argIndex.push_back((size_t)1);
argIndex.push_back((size_t)0);
argIndex.push_back((size_t)2);

ptf->setArgOrder(argIndex);
}
#endif
__declspec( dllimport )
extern "C" const char* MyTable_version()
{
return NPSS_VERSION_STRING;
}
#endif
__declspec( dllimport )
extern "C" void MyTable_loadCompiledObjects(VCInterface* vci)
{
NCPStringArray1D rargs;
ValSequence dumargs;

rargs.append("x");
dumargs.append(NCPVal::REAL);
rargs.append("w");
dumargs.append(NCPVal::REAL);
rargs.append("y");
dumargs.append(NCPVal::REAL);

TableFunction *ptf = new TableFunction("MyTable", dumargs, rargs);
MyTable_initializeTable(ptf);
vci->insertFunction("", ptf);
}

To load the table into your model, use the loadCompiledObjects method:

    loadCompiledObjects("./MyTable");

    real eric[2];
    eric = MyTable.evalStacked(1,1,14);
    cout << eric[0] << " " << eric[1] << endl;

To load a specific table from a single DLM containing several tables, use the 2 argument version of loadCompiledObjects:

    loadCompiledObjects("./Single","Tablename"); //loads the table named Tablename
     //from Single.dll

7.3 VCopy Utility

The VCopy utility is intended to assist component developers and V&V personnel when testing new or modified components prior to deployment. It is available in interactive mode only.

Usage

    VCopy("<basetype>", "<type>");

or

    doCreate = 0;
    VCopy("<sourceObject>", "<targetObject>");

Other interactive command line options which may be set prior to calling VCopy() are as follows:

    verbose = 0 | 1;   // turns OFF|ON informational messages
    checkValue = 0 | 1; // turns OFF|ON value comparisons for variables
    doCreate = 0 | 1;   // turns OFF|ON creation of source object by VCopy

NOTE: When the VCopy utility is #included or loaded, it automatically lists the global variables and their preset defaults.

The VCopy utility verifies that the user accessible copy() function produces an exact duplicate of an existing element, subelement, or component. When the utility is called using its default options, it uses create() and copy() to perform the instantiation and duplication. Indirectly, it provides the value-added service of comparing a class’ constructor and copy constructor for consistency (which are used by create() and copy() respectively).
VCopy() also performs an isa() on each object to verify that both objects return the same result and exhaustively uses the list() function to compare the source and target objects’ attributes. A complete list of the attribute types that are compared by VCopy is cited in the “Implementation” section that follows.

VCopy can be run in either a short or verbose mode by setting an option variable, verbose, prior to calling VCopy(). When verbose is OFF (the default), only inconsistencies and errors during the comparison are reported to the caller. When verbose is ON, VCopy() notes each attribute type that is listed [using list()], the name(s) of each object of that type, as well as any errors.
When the utility compares the list("Variable",0) for the two source and target objects, the actual values of matching variables can be compared by turning the option variable checkValue ON prior to calling VCopy.

A third option value, doCreate, can be set to indicate whether or not the initial object has already been created. When doCreate is ON (the default), the caller is expected to be passing strings representing the baseType and type of the object to be created and subsequently copied.

When doCreate is OFF, the caller is expected to be passing strings representing the name of the object to be copied and the name for the duplicate object. This second form of the VCopy() call is useful for those components which are not included in the DataManager and thus cannot be created, such as interpreted components.

**Implementation**

VCopy() relies on the list() command to non-recursively generate an array of names for each possible object type. The current types that are checked by VCopy() are:

- Input
- Option
- Output

Variable
Socket
DataReader
CaseColumnViewer
CaseRowViewer
PageViewer
VarDumpViewer
Element
Assembly
ElementBase
InterpretedElement
InterpretedSubelement
Subelement
Function
UserFunction
FlowStation
FuelStation
Port
FluidPort
FluidInputPort
FluidOutputPort
InterStageBleedOutPort
Solver

Dependent
DiscreteStateVariable
GSLoop
Independent
Stream
InFileStream
OutFileStream
Table
VCopy() also uses the getName() and getPathName() methods to resolve functions names and the exists() method and value function to verify successful duplication.

**Future Enhancements**

VCopy() could be modified to use separate signatures for testing create-able (DataManager based) components from baseType and type arguments, and those components that may only be directly instantiated. This would eliminate the need for the option variable, doCreate, which changes the meaning of VCopy()’s arguments. For example, VCopyByObject() and VCopyByType() might replace VCopy().

**Example**

VCopy() can be used in unit test procedures to check new or updated components for consistency between their constructor and copy constructor. It is highly recommended that all such tests to be included as part of the NPSS regression suite call VCopy() with verbose mode turned OFF. This allows for thorough review of tested components during regression tests without failures resulting from valid changes in a component’s attributes.

The following example illustrates using VCopy() to verify a Compressor Element using verbose output:

```bash
missle 30\npss.sun2.6 -i
>> #include "/NPSS/dev/Test/UnitTest/cr251/VCopy.util"
Global information variables are:
verbose? (print all check information) 0
doCreate? (let VCopy create the component) 1
checkValue? (check Variable values also) 1
>> setThermoPackage("GasTbl");
>> verbose=1;
>> VCopy("Element", "Compressor");
--------------------------- Verifying copy() on 'Element Compressor':
                Original is a 'Compressor'.
                Copy ... matched.
                Variable 'switchAud' matched.
                Variable 'switchDes' matched.
```

Variable 'a_effAud' matched.
Variable 'a_PRaud' matched.
Variable 'a_WCaud' matched.
Variable 'background' matched.
Variable 'classDescription' matched.
Variable 'description' matched.
Variable 'eff' matched.
Variable 'effPoly' matched.
Variable 'inertia' matched.
Variable 'Nr' matched.
Variable 'NcBase' matched.
Variable 'NcDes' matched.
Variable 'NpqNpDes' matched.
Variable 'PR' matched.
Variable 'PRbase' matched.
Variable 'pwr' matched.
Variable 'Qhx' matched.
Variable 's_effAud' matched.
Variable 's_PRaud' matched.
Variable 's_WCaud' matched.
Variable 'SMNBase' matched.
Variable 'SMWBase' matched.
Variable 'pwrBldSum' matched.
Variable 'WbldSum' matched.
Variable 'switchAud' matched.
Variable 'switchDes' matched.
Variable 'switchEff' matched.
Variable 'time' matched.
Variable 'timeStep' matched.
Variable 'title' matched.
Variable 'TR' matched.
Variable 'TRbase' matched.
Variable 'usageNotes' matched.
Variable 'WpBase' matched.
Variable 'WcDes' matched.
Variable 'WcIn' matched.

Checking Sockets.
Socket 'S_map' matched.

Checking DataViewers.
Checking CaseColumnViewers.
Checking CaseRowViewers.
Checking PageViewers.
Checking VarDumpViewers.
Checking Elements.
Checking Assemblies.
Checking ElementBases.
Checking InterpretedElements.
Checking InterpretedSubelements.
Checking Subelements.
Checking Functions.
Function 'clearSolverTerms' matched.
Function 'copy' matched.
Function 'create' matched.
Function 'delete' matched.
Function 'error' matched.
Function 'execute' matched.
Function 'exists' matched.
Function 'getName' matched.
Function 'getPathName' matched.
Function 'getVal' matched.
Function 'guessAll' matched.
Function 'hasInterface' matched.
Function 'initializeHistory' matched.
Function 'isA' matched.
Function 'list' matched.
Function 'message' matched.
Function 'move' matched.
Function 'provisionalError' matched.
Function 'provisionalWarning' matched.
Function 'run' matched.
Function 'setOption' matched.
Function 'setVal' matched.
Function 'throwError' matched.
Function 'tree' matched.
Function 'updateHistory' matched.
Function 'VCinit' matched.
Function 'verify' matched.
Function 'warning' matched.

Checking UserFunctions.
Checking FlowStations.
FlowStation 'FL_Oideal' matched.
Checking FuelStations.
Checking Ports.
Port 'FL_I' matched.
Port 'FL_O' matched.
Port 'Sh_O' matched.
Checking FluidPorts.
FluidPort 'FL_I' matched.
FluidPort 'FL_O' matched.
The following output is produced when the default options are used:

```
missile 31%
>> load /NPSS/dev/Test/UnitTest/cr251/VCopy.util
Global information variables are:
verbose? (print all check information) 0
doCreate? (let VCopy create the component) 1
checkValue? (check Variable values also) 1
>> setThermoPackage("GasTbl");
>> VCopy("Element", "Compressor");
--------------------------- Verifying copy() on 'Element Compressor':
Original is a 'Compressor'.
Copy is a 'Compressor'.
Finished verifying copy() for 'Element Compressor'.
---------------------------
```

```
>> quit
missile 31%
```

7.4 DLMgen Utility (UNIX Only)

The DLMgen utility can be used to create individual DLM(.sc) files.

**Usage:**

```
DLMgen inputfilepath [targetdirectory]
```

The DLMgen utility takes the file “inputfilepath,” creates a DLM of the same name with a “.sc” extension, and moves the DLM to the target directory. The user must have write access to the current working directory. The inputfilepath can be any interpretive (.int) or C++ (.C) file the user has read access to. The target directory can be any directory the user has write access to. The default target directory is the current working directory.
Implementation:
DLMgen verifies the existence of the input file and the write permissions of the working files and directories. It then creates a directory (inputfilename+processID) in the current working directory and copies all the necessary files to it. The DLM is created using the NPSS DevKit npssconvert utility and make system and then moved to the target directory. The temporary working directory is then deleted.

Future Enhancements:
DLMgen will be enhanced to enable the building of multiple files into DLM libraries (.scl).

Examples:
```
```
It takes the file “$SIM_SHR/shared/Duct/v4.3/Duct.C”, creates the file Duct.sc, and places it in the directory “$SIM_SYS/shared/Duct/v4.3”

```
DLMgen /home/userid/Burner.int
```
It takes the file “/home/userid/Burner.int,” processes it with “npssconvert,” creates “Burner.sc,” and stores it in the current working directory.

7.5 DLMgen2.py
DLMgen2.py is a Python script used to generate a DLM which contains multiple components.

The script requires a minimum of three arguments:
```
--outdir=<directory> This is a temporary directory and is removed after completion.
--modelPaths=<directory path> This is the directory where the model containing the interp files to be converted and compiled into the DLM is located.
--fint=<filename.int> This is the file to be converted. This option can be repeated for as many objects as the DLM need to contains.
--debug=1 This is optional and will prevent the clean up and removal of the supplied temporary working directory.
```

The script will generate a temporary working directory where the converted C and H files are stored. The script will then generate the necessary “make” files and manifest needed by the DLM. Once the DLM is compiled, the DLMs are placed in a DLMComponents directory under the “modelPaths” directory supplied in a sub-directory by platform.

Example: CR2437/model/DLMComponents/<platform>

The setEnv.bat and testscript.bat are example files provided to aid in building the DLM/dll under a Windows native command prompt (no cygwin needed)
Examples:

**UNIX systems:**

```
DLMgen2.py --outdir=mytemp --modelPaths=./<path>/<model directory> --
fint=file_1.int --fint=file_2.int --fint=file_3.int --fint=file_n.int
```

**Windows:**

Windows command window (with Python in the PATH environment variable):

```
python DLMgen2.py --outdir=mytemp --modelPaths=./<path>/<model directory> --
fint=file_1.int --fint=file_2.int --fint=file_3.int --fint=file_n.int
```

Cygwin command window:

```
/cygdrive/<drive letter>/path_to_python/python.exe DLMgen2.py --outdir=mytemp --modelPaths=./<path>/<model directory> --
fint=file_1.int --fint=file_2.int --fint=file_3.int --fint=file_n.int
```

DLMgen2 will call the converter on the supplied interp files and then create the "_manifest.C" file needed by the DLM to call the init functions. DLMgen2 then creates the make files and builds the requested DLM. Once the build has completed DLMgen2, the new DLM is copied into the DLMComponents directory under the directory supplied in the "modelPaths" option.

### 7.6 ConstConverter (All Platforms)

The constants have been changed to a DLM format. The DLM format allows additional constants, which are contained in a user-defined constants DLM, to be defined. A standard set of constants must be defined in the user-defined constants DLM; they are contained in the NPSSConstants.def file included in the ConstConverter directory. This definition file is used by the ConstConverter to generate the source code used to build the DLM containing the constants and their values.

The ConstConverter utility is located in the /NPSS/dev/util/src/ConstConverter directory. To build the utility for use, simply type "make" or "nmake /f Makefile.win32" at the command prompt.

Use the commandline "ConstConverter <definition file>" to generate the source needed to build a constants DLM. This command will generate the C and H files needed to build a user-defined constants DLM. The needed functions for loading compiled objects and reporting the version string will take the name of the file provided.

The files generated are written to the current working directory. Command line options are available to direct the output of the generated source code.

These options are:

- `-H <path>`, to place Constants.H file in a desired directory
- `-F <path>`, to place Constants.inc file in a desired directory
- `-C <path>`, to place Constants.C file in a desired directory
- `-D <path>`, to place Const.C file in a desired directory
- `-Z <path>`, to place ConstantsSetup.in file in a desired directory

To regenerate the needed file for the default set of constants used by NPSS, run the ConstConverter as follows: "ConstConverter NPSSConstants.def --S"
The constants converter requires only one argument to generate the required source files for a user-defined set of constants in a DLM. This argument is the definition file containing the user-defined constants. This file must also contain the default set of constants as defined in the NPSSConstants.def file in the ConstConverter directory.

```
NPSSConstants.def

// *** CONSTANTS ***
// ALL CONSTANTS ARE FROM THE NIST STANDARDS: NIST Guide to SI Units
// SP811, Appendix B9, retrieved from NIST website Jan 2002

// GRAVITY value was 32.17, now 32.174049
name C_GRAVITY
type real
units "ft*lbm/(lbf*sec2)"
//value 1 "m*kg/(N*sec2)"
value 32.174049
description "gravitational constant, (AKA g sub c)"
IStatus CONST

// G_ACCEL value was 32.17, now 32.174049
name C_G_ACCEL
type real
units "ft/sec2"
//value 9.80665 "m/sec2"
value 32.174049
description "acceleration due to gravity"
IStatus CONST

// PSTD value was 14.696, now 14.695951
name C_PSTD
type real
units "psia"
//value 101.325 "kPa"
value 14.695951
description "standard pressure"
IStatus CONST

// R AIR value was 1716.56, now 1716.351736
name C_RAIR
type real
units "ft2/(sec2*R)"
//value 101.325/1.22515/288.15 "kPa*m3/(kg*K)"
value 1716.351736
description "gas constant for air at Tstd (15 C)"
IStatus CONST

// R p/(T*rho) (101.325 kPa)/(1.22525 kg/m3)/(288.15 K)
// values retrieved from NIST Standard Ref. Database 72, NIST Thermophysical
// properties of Air and Air Component Mixtures

// TSTD was/will remain 518.67 R
name C_TSTD
type real
units "R"
//value 15 "C"
value 518.67
description "standard temperature"
IStatus CONST

// Pi defined as part of system as 3.1415926535897931,
// so is E as 2.71828182845905
```
// CONVERSION CONSTANTS

// JOULES_CONST value was 778.1600, will be 778.169233
name C_BTUtoFT_LBF
type real
value 1055.05585262/1.355818
units "ft*lbf/Btu"
description "mechanical equivalent of heat"
IOstatus CONST
// to use current NIST numbers had to use BTUtoJ(international)/FT_LBFtoJ

// HP_TO_FTLB_PER_SEC no change in value
name C_HPttoFT_LBF_PER_SEC
type real
value 550
units "ft*lbf/(sec*hp)"
description "conversion of hp to ft-lbf/sec"
IOstatus CONST

// C_HP_SEC_PER_BTU value was 1.414836, will be 1.414853
name C_BTU_PER_SECtoHP
type real
value C_BTUtoFT_LBF/C_HPttoFT_LBF_PER_SEC
units "hp/sec/Btu"
description "conversion from enthalpy to power"
IOstatus CONST

// IN_PER_FT value no change
name C_FTtoIN
type real
value 12.0
units "in/ft"
description "conversion from feet to inches"
IOstatus CONST

// IN2_PER_FT2 value no change
name C_FT2toIN2
type real
value 144.0
units "in2/ft2"
description "conversion from feet squared to inches squared"
IOstatus CONST

// IN3_PER_FT3 value no change
name C_FT3toIN3
type real
value 1728.0
units "in3/ft3"
description "conversion from cubic feet to cubic inches"
IOstatus CONST

// FT_PER_IN value no change
name C_INtoFT
type real
value 1/C_FTtoIN
units "ft/in"
description "conversion from inches to feet"
IOstatus CONST

// SEC_PER_MIN
name C_MINtoSEC
type real
value 60.0
units "sec/min"
description "conversion from minutes to seconds"
IOstatus CONST

// SEC_PER_HOUR
name C_HOURtoSEC
type real
value 3600.0
units "sec/hr"
description "conversion from hour to seconds"
IOstatus CONST

// RAD_PER_SEC_PER_REVS_PER_MIN value was 0.1047198000, will be 0.1047197551
name C_RPMtoRAD_PER_SEC
type real
value 2*PI/C_MINtoSEC
units "rad/(sec*rpm)"
description "conversion from rpm to rad/sec"
IOstatus CONST

// C_FT_LBF_RPM_PER HP value was/will remain 5252.113122
name C_HP_PER_RPMtoFT_LBF
type real
value (C_HPtoFT_LBF_PER_SEC / C_RPMtoRAD_PER_SEC)
units "ft*lbf/rpm/HP"
description "conversion from power to torque"
IOstatus CONST

// **** OLD VARIABLE NAMES WITH NEW VALUES
// THIS WILL GIVE PEOPLE WITH PRIVATE ELEMENTS TIME TO CHANGE TO NEW NAMES

name GRAVITY
type real
units "ft*lb/(lbf*sec2)"
//value 1 "m*kg/(N*sec2)"
value 32.174049
description "gravitational constant, (AXA g sub c)"
IOstatus CONST

name G_ACCEL
type real
units "ft/sec2"
//value 9.80665 "m/sec2"
value 32.174049
description "acceleration due to gravity"
IOstatus CONST

name PSTD
type real
units "psia"
//value 101.325 "kPa"
value 14.695951
description "standard pressure"
IOstatus CONST

name RAIR
type real
units "ft2/(sec2*R)"
//value 101.325/1.22515/288.15 "kPa*m3/(kg*K)"
value 1716.351736
description "gas constant for air at Tstd (15 C)"
IOstatus CONST
// R p/(T*rho)  (101.325 kPa)/(1.22525 kg/m3)/(288.15 K)
// values retrieved from NIST Standard Ref. Database 72, NIST Thermophysical
// properties of Air and Air Component Mixtures

ame TSTD
  type real
  units "R"
  //value 15 "C"
  value 518.67
  description "standard temperature"

IOstatus CONST
//CONVERSION CONSTANTS

name JOULES_CONST
  type real
  value 1055.05585262/1.355818
  units "ft*lbf/Btu"
  description "mechanical equivalent of heat"

IOstatus CONST
// to use current NIST numbers had to use BTUtoJ(international)/FT_LBFtoJ

name HP_TO_FTLB_PER_SEC
  type real
  value 550
  units "ft*lbf/(sec*hp)"
  description " conversion of hp to ft-lbf/sec"

IOstatus CONST

name C_FT_LBF_RPM_PER_HP
  type real
  value C_HPtoFT_LBF_PER_SEC/C_RPMtoRAD_PER_SEC
  units "ft*lbf*rpm/hp"
  description "conversion from power to torque"

IOstatus CONST

name C_HP_SEC_PER_BTU
  type real
  value C_BTUtoFT_LBF/C_HPtoFT_LBF_PER_SEC
  units "hp*sec/Btu"
  description "conversion from enthalpy to power"

IOstatus CONST

name FT_PER_IN
  type real
  value 1.0/12.0
  units "ft/in"
  description " conversion from inches to feet"

IOstatus CONST

name IN_PER_FT
  type real
  value 12.0
  units "in/ft"
  description " conversion from feet to inches"

IOstatus CONST

name IN2_PER_FT2
  type real
  value 144.0
  units "in2/ft2"
  description " conversion from feet squared to inches squared"

IOstatus CONST
name IN3_PER_FT3  
type real  
value 1728.0  
units "in3/ft3"  
description "conversion from cubic feet to cubic inches"  
I0status CONST

name RAD_PER_SEC_PER_REVS_PER_MIN  
type real  
value 2*PI/C_MINtoSEC  
units "rad/(sec*rpm)"  
description "conversion from rpm to rad/sec "  
I0status CONST

name SEC_PER_HOUR  
type real  
value 3600.0  
units "sec/hr"  
description "conversion from hour to seconds"  
I0status CONST

name SEC_PER_MIN  
type real  
value 60.0  
units "sec/min"  
description "conversion from minutes to seconds"  
I0status CONST

name NANQ  
type real  
value -9999  
units "none"  
description "not a number"  
I0status CONST

// IN2_PER_FT2 value no change  
name C_FT2toIN2  
type real  
value 144.0  
units "in2/ft2"  
description " conversion from feet squared to inches squared"  
I0status CONST

// IN3_PER_FT3 value no change  
name C_FT3toIN3  
type real  
value 1728.0  
units "in3/ft3"  
description "conversion from cubic feet to cubic inches"  
I0status CONST

// FT_PER_IN value no change  
name C_INtoFT  
type real  
value 1/C_FTtoIN  
units "ft/in"  
description " conversion from inches to feet"  
I0status CONST

// SEC_PER_MIN  
name C_MINtoSEC  
type real  
value 60.0
units "sec/min"
description "conversion from minutes to seconds"
IOstatus CONST

// SEC_PER_HOUR
name C_HOURtoSEC
type real
value 3600.0
units "sec/hr"
description "conversion from hour to seconds"
IOstatus CONST

// RAD_PER_SEC_PER_REVS_PER_MIN value was 0.1047198000, will be 0.1047197551
name C_RPMtoRAD_PER_SEC
type real
value 2*PI/C_MINtoSEC
units "rad/(sec*rpm)"
description "conversion from rpm to rad/sec"
IOstatus CONST

// C_FT_LBF_RPM_PER_HP value was/will remain 5252.113122
name C_HP_PER_RPMtoFT_LBF
type real
value (C_HptoFT_LBF_PER_SEC / C_RPMtoRAD_PER_SEC)
units "ft*lbf/rpm/HP"
description "conversion from power to torque"
IOstatus CONST

// **** OLD VARIABLE NAMES WITH NEW VALUES
// THIS WILL GIVE PEOPLE WITH PRIVATE ELEMENTS TIME TO CHANGE TO NEW NAMES

name GRAVITY
type real
units "ft*lbm/(lbf*sec2)"
// value 1 "m*kg/(N*sec2)"
value 32.174049
description "gravitational constant, (AKA g sub c)"
IOstatus CONST

name G_ACCEL
type real
units "ft/sec2"
// value 9.80665 "m/sec2"
value 32.174049
description "acceleration due to gravity"
IOstatus CONST

name PSTD
type real
units "psia"
// value 101.325 "kPa"
value 14.695951
description "standard pressure"
IOstatus CONST

name RAIR
type real
units "ft2/(sec2*R)"
// value 101.325/1.22515/288.15 "kPa*m3/(kg*K)"
value 1716.351736
description "gas constant for air at Tstd (15 C)"
IOstatus CONST
// R p/(T*rho) (101.325 kPa)/(1.22525 kg/m3)/(288.15 K)
// values retrieved from NIST Standard Ref. Database 72, NIST Thermophysical // properties of Air and Air Component Mixtures

name TSTD
  type real
  units "R"
  //value 15 "C"
  value 518.67
  description "standard temperature"
  IOstatus CONST

// CONVERSION CONSTANTS

name JOULES_CONST
  type real
  value 1055.05585262/1.355818
  units "ft*lb/ftu"
  description "mechanical equivalent of heat"
  IOstatus CONST
  // to use current NIST numbers had to use BTUtoJ(international)/FT_LBtoJ

name HP_TO_FTLB_PER_SEC
  type real
  value 550
  units "ft*lbf/(sec*hp)"
  description " conversion of hp to ft-lbf/sec "
  IOstatus CONST

name C_FT_LBF_RPM_PER_HP
  type real
  value C_HPtoFT_LBF_PER_SEC/C_RPMtoRAD_PER_SEC
  units "ft*lbf.rpm/hp"
  description "conversion from power to torque"
  IOstatus CONST

name C_HP_SEC_PER_BTU
  type real
  value C_BTUtoFT_LBF/C_HPtoFT_LBF_PER_SEC
  units "hp/sec/Btu"
  description "conversion from enthalpy to power"
  IOstatus CONST

name FT_PER_IN
  type real
  value 1.0/12.0
  units "ft/in"
  description " conversion from inches to feet"
  IOstatus CONST

name IN_PER_FT
  type real
  value 12.0
  units "in/ft"
  description " conversion from feet to inches"
  IOstatus CONST

name IN2_PER_FT2
  type real
  value 144.0
  units "in2/ft2"
  description " conversion from feet squared to inches squared"
  IOstatus CONST
name IN3_PER_FT3
type real
value 1728.0
units "in3/ft3"
description "conversion from cubic feet to cubic inches"
IOstatus CONST

name RAD_PER_SEC_PER_REVS_PER_MIN
type real
value 2*PI/C_MINtoSEC
units "rad/(sec.rpm)"
description "conversion from rpm to rad/sec"
IOstatus CONST

name SEC_PER_HOUR
type real
value 3600.0
units "sec/hr"
description "conversion from hour to seconds"
IOstatus CONST

name SEC_PER_MIN
type real
value 60.0
units "sec/min"
description "conversion from minutes to seconds"
IOstatus CONST

name NANQ
type real
value -9999
units "none"
description "not a number"
IOstatus CONST
<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
<th>Description of change</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>06/11/98</td>
<td>Updated after doc inspection. Added: more detail on components, info on functions, attributes, moved makefile chapter after Chapter 4, added 2nd makefile section. Incorporates CR 301. Note: Placeholders (headings) for new material were added but information incomplete. Released with REL_0.2.0.</td>
</tr>
<tr>
<td>B</td>
<td>8/20/99</td>
<td>Incorporates the following CRs: 250 (C++ converter), 254 (port and socket names), 255 (variable names), 268 (array functionality—serve-side utility incomplete), 324 (override nextExecTime), 360 (DSVs), Transient histories. Incorporates suggestions from doc inspection 5/8/99—expand sections, make them consistent, generic test plan, Secant Solver, general info on components. Added variable naming conventions. Released with REL_1.0.0.</td>
</tr>
<tr>
<td>C</td>
<td>11/18/98</td>
<td>Incorporates CR395 (CCDK) and new directory structure.</td>
</tr>
<tr>
<td>D</td>
<td>2/1/99</td>
<td>Incorporates the following CRs in build Rel_1.0.3L: 362 (Secant Solver update, sect. 3.2.4.2, 4.4.8.2)[12/7/98] and 258 (build system changes, 4.5, 4.5.1, 4.5.2)[2/1/99]</td>
</tr>
<tr>
<td>E</td>
<td>7/22/99</td>
<td>Incorporates the following CRs in build REL_1.0.4: CR377 (replaced DELTA with ADDER in 3.7.1, 4.6.1)[4/5/99] CR519 (done on branch #536, if-else syntax, 4.2.8.1, 4.5.2, 4.6.1)[4/8.99] CR488 (new variable class w/adder &amp; scalar, 3.2.1, 4.2.2, 4.2.3)[5/19] CR295 (copyright)[6/17/99] CR475 (autoSolverSetup: active/inactive removed)[7/15] CR600 (DLMs and NPSS Dev Kit), CR602 (Model Delivery, added to new chapter)[7/23] REL_1.0.4 is the same as NPSS_0.1.0.</td>
</tr>
<tr>
<td>F</td>
<td>3/29/00</td>
<td>The following CRs are incorporated into Version 1, NPSS_1.0.0: New index entries.[7/23/99] Deleted “Work” from ShaftWorkInputPort (CR453, rel 1.0.4)[8/5] CR655 (audit factors, 2.6.1, 3.5.2)[8/11] CR693 (including interpreted components, 2.6, 2.6.1, 2.7.1)[9/17] CR717 (NCopy Utility, relates to CR251, new chapter 7)[11/9/99] CR605 (new postcreate function, 2.3.2.) CR823 (npgs to C++ converter) added to Utilities chart.; minor text changes for clarification CR786 (runDiscreteCalcs returns int)[12/17] updated sections on creating solver objects; changes related to CR475 (autoSolver, 3.2.8, 3.3.3)[1/3/99] indexing[1/6] checked changes per 475, minor corrections[2/18] CR578 (Porttran Interface, 3.4.5, 5.5); some general updates per Bret; formatting consistencies.[3/6] CR816 (General updates per Bret, CR for doc REQ). CR893 (DLMs, CMFs, chapter 5) (same chapter, removed info on FORTRAN interface[CR578] as problem with code)[11/9/99] CR893 Updated Dev Kit info, 5.0[3/29/00] corrected typog to script on 3/30, no substantive change Rev F completes Version 1, NPSS_1.0.0</td>
</tr>
<tr>
<td>G</td>
<td>8/23/00</td>
<td>CRs incorporated into NPSS_1.1.0: CR847 (constraints, 2.2.4.1, 2.3.2.1, 3.3.4)[4/27/00] CR578 (FORTRAN API info)[5/5] added back; work on branch needed. NOTE: Does not work on HP.)[5/25/00] CR912 (phrase for reference stations, 2.2.8)[6/12/00] CR981 (DMUX utility, 7.3)[6/16/00]</td>
</tr>
<tr>
<td>H</td>
<td>11/15/00</td>
<td>CRs that comprise NPSS_1.2.0: CR1003 (inheriting from existing type, new 2.6.2.) CR2911 (2 command line options added to autodoc, 9.0)[9/22] Ref station info added—but Stephen to enhance[10/30]</td>
</tr>
<tr>
<td>I</td>
<td>6/13/01</td>
<td>CRs incorporated into NPSS_1.3.0: CR1064 (new info on using autodoc to generate records ref sheets, 9.0)[12/14/00] CR233 (deleted RealTrigger, IntTrigger, StringTrigger, 2.2.4.5; deleted ref. to trigger variables in 2.3.2, added line to 2.7.1)[1/2/01] CR1109 (Added tech paper to appendix—reformatted from PDF)[2/14] CR1015 &amp; CR1147 (External elements, CCDK updates; Chapter 4—major revisions: 4.2, 4.5.4.7 new text; other subsections deleted)[4/5/01] CR1253 (Added paper on DLMS, new chap 5)[4/6/01] CR1196 (Converter updates, 8.1)[5/23] CR1205 (customer dock enhancements, 5.4.1, 5.4.3, 5.5.1, 5.5.3)[5/24/01] CR1154 (fixed CD bugs, same sections as 1205) Paper added for CR1253—just moved to later chapter, after CCDK discussion.[6/1] Changed rev to NPSS_1.3.0[6/13/01]</td>
</tr>
<tr>
<td>K</td>
<td>3/20/02</td>
<td>CRs incorporated into NPSS_1.5.0 (highlight indicates change from 1.4.0. If a heading highlighted the entire section is new or has changed significantly) CR1420 (additions to doc for CR969, security model &amp; access control, 5.4.5.6)[10/4/01] Formatting changes only [11/21] CR1404 (delete part of #3 in 5.4)[12/5/01] CR1098 (deleted references to Rogue Wave)[7/20/02] CR1464 (CCDK updates 4.2, 4.6.4, 4.8.1, &amp; 4.8.2 revised; 4.10 new)[3/15] 1.5.0 markings 3/20/02</td>
</tr>
<tr>
<td>M</td>
<td>9/26/03</td>
<td>CRs incorporated into NPSS_1.6.0: CR1751 (interpreted components as indirect wrappers, BRSTK, related updates; 2.8, 4.4.6.2.1, 4.7.1, 4.7.3, 4.8.3) [REV: 1.5.1, 9/9/03] CR1697 (DevKit update–Unix and Windows, 5.0; model delivery DevKit split out, 6.0, distrb scripts and examples updated, misc. updates)[9/11] CR1923 (broke out doc into separate docs for CCDK, removed chapter on external components, removed tech paper on DLM 1-D coupling and appendix, [9/12/03] [REV: O] Fixed typos and updated title page and headers for 1.6.0 [9/23/03]</td>
</tr>
<tr>
<td>O</td>
<td>9/30/04</td>
<td>CRs incorporated into NPSS_1.6.2: CR2050 (updates to 2.8.1.2, 2.8.2.1)[REV: I, 7/14] CR2099 (corrected 4.2.2 where DLMS placed in directory)[7/29/04, REV: L]</td>
</tr>
</tbody>
</table>
**CRs incorporated into NPSS_1.6.3:** CR2129 (relative path name change, 17.1) [10/15/04, Rev: A] CR1839 (new class RemoteJobExec. 2.8.2.6) [10/19/04, REV: B] CR2160 (table converter, new section 8.20) [2/8/05, Rev: Q] CR2166 (Updates to station objects, 2.2.3, 3.2.4) ("this" pointer added to the PsSecant solver initialize function, 2.2.4.2 & 3.2.6.2.) [Rev: Y, 3/23/05] CR1835 (secant solver needs to throw errors, 3.2.6.2) [Rev: Y, 3/25] CR2141 (building CD using shared library, 5.2.3) [3/29/05, Rev: AF]

**CRs incorporated into NPSS_1.6.3:** CR1957 (new function initChildHistories(), 2.3.2, 3, 3.2.5) [Rev: D 6/1/05] CR1843 (Changes to fix dataports, 2.2.1, 2.8.11, 2.8.4, 2.8.5, 3.2.1, 3.2.2, 7.1) Rev: H, 6/15/05. CR2249 (multiple tables in a single DLM, 7.2 [Rev :R , 9/2/05] CR1655 (utility for constant converter, 7.5 added)[Rev: V, 11/8/05] CR2292 (converter fix for proper conversion of Matrix element access, fixed an unhandled exception problem in the converter; added “depends” attribute to metadata, Chapter 7)[Rev: AF; 2/3/06]. CR1739 (updates re converter, batchJob, WrapperBase, etc.; sections 2 & 7 affected) [3/17,06, Rev: AL) CR2324 (update doc for CR2283, 2.3.2) [5/12, Rev; AO] CR2319 (WrapperBase functions 2.8.1; setup option, 7.1) [Rev:AP, 5/15/06] CR2313 (changed to constants converter, replaced 7.5) Rev: AO 5/18/06

**CRs incorporated to make NPSS_1.6.5:** CR2336 (In-house simulation using native NPSS API; created new section 5.3 so 5.4 moved up) [8/10/06, Rev: H] CR2362 (correct typo in 5.2.2.) [9/20/06, Rev: ] CR1140 (converter changes and new options) [11/3/06, Rev: ] CR2386 (added LAPIN, updates to section 2) [Rev: AA, 5/17/07] CR1844 (unstructured domain variable changes, sections 2.8.1.1, 2.8.4) [Rev: AI, 01/17/08] CR2437 (Python script to generate 1 DLM with multiple components, new section 7.5.) [2/15/08, 164AK] CR2441 (no substantive changes; updates to markings only for release package) [3/12/08]
Index

access
  attribute of an object, 5-2
activateSecureMode(), 5-4
addClassMap, 7-2
addConstraint, 3-6
addCreatePermission(), 5-8
addDependent(), 3-13
addDSV(), 3-13
addFunctMap, 7-1
addIndependent(), 3-13
ADDINS, 4-11
addVarMap, 7-1
alias file, 5-25
Assembly, 1-1
autoSolverSetup, 2-3, 3-6

batch jobs
  classes for, 2-16
BatchJob, 2-16
BatchJobExec, 2-17
BINDROPDIR, 4-11
building
  under UNIX, 4-6
  under Windows, 4-7
building custom executable (UNIX), 4-11
building custom executable (Windows), 4-12

calculate(), 2-7, 2-9, 3-12
CDgen
  to create C++ files (FORTRAN API), 5-26
  using to create C++ files, 5-9
clear(), 3-3
clearSolverTerms function, 3-6
command line options
  used with CDgen, 5-9
COMPDROPDIR, 4-12
component
  definition, 1-1
  primary types, 1-1
component directory
  example, 4-13
component source tree, 4-1, 4-5
  layout, 4-12
component suite, 4-1
constants, 7-12
constraining target Dependents, 3-6
constraint Dependents, 3-6
converter variables, 7-2
copyVarFunctMap, 7-2
CORBA Component Development Kit (CCDK), 1-1
customer simulation
  building with FORTRAN API, 5-24
  building with native NPSS API, 5-4
  creating, 5-7
defaultAccess, 5-2
descramble, 5-7
Development Kit
  example builds, 4-5
Devkit, 5-10
DLM
  compiling component source tree (UNIX), 4-7
  compiling component source tree (Windows), 4-8
DLM components, 1-1
DLMgen utility, 7-10
DLMs
  modes for packing, 4-8
  multiple components, 4-8
  single component, 4-8
driver program, 5-10
Element, 1-1
  adding interpreted, 2-12
Elements
  inheriting from existing, 2-12
  encrypted files, 5-25
  encryption key, 5-25
  environment variables
    setting, 4-4
  error handling, 3-14
    internal component, 3-14
    interpreted component, 2-11
  errorFlag(), 3-8
  errorFound(), 3-8
example DLM build
  using C++ code for single DLM, 4-9
  exceptions, 2-11, 3-14
  exists(), 2-7
  external components, 1-1
  extrapolate(), 3-3
  extrapolateTo(), 3-3
FlowStations, 2-3, 3-3
FORTRAN API
  description, 5-24
FuelStation, 3-3
Function objects, 3-10
functions
convenience, 3-13

G
getAccess(), 5-4
getAlias(), 5-4
gGetCurrentIter(), 3-8
gGetCurrentTime(), 3-4
gGetCurrentValue(), 3-4
ggetDefaultAccess(), 5-4
gGetDerivAt(), 3-4
gGetDerivative(), 3-4
gGetDatAt(), 3-4
gGetDValue(), 3-4
gGetDValueAt(), 3-4
gGetFunction(), 3-13
gGetMaxDx(), 3-8
gGetMaxIter(), 3-8
gGetPastTime(), 3-4
gGetPerturb(), 3-9
gGetPrevTime(), 3-4
gGetPrevValue(), 3-4
gGetStation(), 3-13
gGetTolerance(), 3-9
gGetValue(), 3-4
GlobusJobExec, 2-18
GNU Make, 4-3
installing, 4-3
gtDt(), 3-4

I
indirect wrappers, 2-14
initChildHistories(), 2-9, 3-4, 8-2
initHistoryFull(), 3-4
initialize(), 3-9
initializeHistory(), 2-9, 3-4
INITMD(), 5-24
interactive command line options
set prior to calling VCopy, 7-9
internal components, 1-1, 6-1
convenience functions, 3-13
function objects, 3-10
generic NPSS values, 3-1
optional functions, 3-13
required functions, 3-12
tables, 3-11
variable types, 3-2
internal element
adding, 3-14, 3-17
internal elements and subelements
inherance hierarchy, 3-24
internal subelement
adding, 3-20
interpolate(), 3-4
interpreted components, 1-1, 2-1, 4-15
converting to C++ code, 7-1
objects to aid development, 2-1
sockets, 2-6
tables, 2-6
interpreted elements, 2-2, 2-13
adding to NPSS, 2-11
interpreted subelements
adding, 2-13
isConverged(), 3-9
iterate(), 3-9

L
LIBDROPDIR, 4-12
localCreate(), 3-13
LSFJobExec, 2-19

M
make system, 4-7, 4-12, 4-15
make variables, 4-6, 4-7, 4-11
Makedefs, 3-15
makefile system, 3-15
manifest, 4-15
MDdevkit, 5-1
MemberFunction<T>, 3-10
Model Delivery Development Kit, 5-1
modes, Solver, 2-4
MyElement::MyElement(), 3-12

N
National Cycle Program, 1-1
NCPFunctVariable<T, R>, 3-2
NCPOptionVariable, 3-2
NCPRealScalarAdder, 3-1
NCPRefVariable<T>, 3-2
NCPVal, 3-1
NCPVariable, 3-3
needVerify(), 3-13
nextExecTime(), 2-9
NPSS API
description of, 5-4, 5-10
NPSS Development Kit
installing, 4-2
requirements for using, 4-2
NPSS Development Kit (DK), 4-1
NPSS_getFDS(), 5-6
NPSS_get2DSize(char* name, int size1, int size2), 5-6
NPSS_get3DSize(char* name, int size1, int size2, int size3), 5-6
NPSS_getID(), 5-6
NPSS_getID1(), 5-6
NPSS_getID1Dentry(), 5-6
NPSS_getID2Dentry(), 5-6
NPSS_getDimSize(char* name, int* buff, int buffsize, int numRead), 5-6
NPSS_getF(), 5-5
NPSS_getF1D(), 5-5
NPSS_getF1Dentry(), 5-5
NPSS_getF2Dentry(), 5-6
NPSS_getI(), 5-5
NPSS_get1D(), 5-5
NPSS_getIDentry(), 5-5
NPSS_getS(), 5-5
NPSS_get1Dentry(), 5-5
NPSS_init(), 5-7
NPSS_onNPSSTermination(), 5-7
NPSS_parseEfile(), 5-5
NPSS_parseFile(), 5-5
NPSS_parseString(), 5-5
NPSS_setD(), 5-6
NPSS_set1D(), 5-6
NPSS_set1Dentry(), 5-7
NPSS_set2Dentry(), 5-7
NPSS_setF(), 5-6
NPSS_set1D(), 5-6
NPSS_set1Dentry(), 5-7
NPSS_set2Dentry(), 5-7
NPSS_setD(), 5-6
NPSS_set1D(), 5-6
NPSS_set1Dentry(), 5-7
NPSS_setS(), 5-6
NPSS_set1Dentry(), 5-7
NPSS_variableExists(char* name), 5-7
npssconvert, 7-1

O
ODIR, 4-11

P
PBSJobExec, 2-19
port
  types, 2-2
  ports, 2-2, 3-3
postcreate(), 2-9
Proprietary/Classification Notice, 5-25
provisional errors, 2-11, 3-14
PRNTOUT(), 5-24
PsSecant Attributes, 3-8
PsSecant Functions, 3-8
PsSecant(), 3-8
Python
  installing, 4-3
Python language interpreter, 4-3

R
realScalarAdder, 2-1
RemoteJobExec, 2-19
RESETI(), 5-24
runDiscreteCalcs(), 2-10
RUNMD(), 5-24

S
scramble, 5-7
scramble program, 5-7, 5-9, 5-25, 5-27
SecantSolver, 2-5, 2-6, 3-7
secure objects, 5-1
security ID, 5-2
security model, 5-1, 5-24
setAccess(), 5-2, 5-4, 5-8
setAlias(), 5-4, 5-8
setDefaultAccess(), 5-2, 5-4
setDefaultAccess("PRIV"), 5-8
setMaxDx(), 3-9
setMaxIter(), 3-9
setPerturb(), 3-9
setTolerance(), 3-9
ShellJobExec, 2-20
SimpleWrapper, 2-16
singlefile, 5-8
Socket
  returning pointer to, 3-13
SocketPtr getSocket(), 3-13
Sockets, 2-6, 3-9
solver, 3-5
Solver
  internal, 3-7
  objects, 2-3, 3-5
  Secant, 3-7
Solver object
  adding, 2-4
  solver term
  removing, 3-6
STOREI(), 5-24
subelement
  internal, 3-20
Subelement, 1-1
Subelements, 2-6
switchDes, 3-6
system makefiles
  customizing (UNIX), 4-4

T
Tables, 2-6, 3-11
target Dependents, 3-6
test plan
  sample, 6-2
testing
  internal components, 6-1
  sample output, 6-7
TransHistory, 2-7, 3-3

U
unit conversions, 2-11
unsecure objects, 5-1
updateChildHistories(), 3-5
updateHistory(), 2-10, 3-5
User Defined Classes
  for interpreted code, 2-8

V
ValSequence, 3-2
variable types, 2-1
variableChanged(), 2-10, 3-13
VCInterfacePtr clone(), 3-12
VCopy utility, 7-9, 7-10
verify(), 2-11, 3-13, 3-14
void VCinit(), 2-10