NREL
NATIONAL RENEWABLE ENERGY LABORATORY

NWTC Programmer's Handbook: A Guide for Software Development Within the FAST Computer-Aided Engineering Tool

B.J. Jonkman, J. Michalakes, J.M. Jonkman, M.L. Buhl, Jr., A. Platt, and M.A. Sprague

August 24, 2012
Draft Version
for External Review
Updated July 17, 2013

NREL is a national laboratory of the U.S. Department of Energy, Office of Energy Efficiency & Renewable Energy, operated by the Alliance for Sustainable Energy, LLC.

Technical Report
NREL/TP-xxxx-xxxxx
December 2012

Contract No. DE-AC36-08GO28308

NREL
NATIONAL RENEWABLE ENERGY LABORATORY

NWTC Programmer's Handbook: A Guide for Software Development Within the FAST Computer-Aided Engineering Tool

B.J. Jonkman, J. Michalakes, J.M. Jonkman, M.L. Buhl, Jr., A. Platt, and M.A. Sprague

Prepared under Task Nos. WE110336 and WE115070

August 24, 2012
Draft Version
for External Review
Updated July 17, 2013

National Renewable Energy Laboratory
1617 Cole Boulevard
Golden, Colorado 80401
303-275-3000 • www.nrel.gov

Technical Report
NREL/TP-xxxx-xxxxx
December 2012

Contract No. DE-AC36-08GO28308

NOTICE

Available electronically at http://www.osti.gov/bridge

Available for a processing fee to U.S. Department of Energy
and its contractors, in paper, from:

U.S. Department of Energy
Office of Scientific and Technical Information

P.O. Box 62
Oak Ridge, TN 37831-0062
phone: 865.576.8401
fax: 865.576.5728
email: mailto:reports@adonis.osti.gov

Available for sale to the public, in paper, from:

U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
phone: 800.553.6847
fax: 703.605.6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/help/ordermethods.aspx

Cover Photos: (left to right) PIX 16416, PIX 17423, PIX 16560, PIX 17613, PIX 17436, PIX 17721

Printed on paper containing at least 50% wastepaper, including 10% post consumer waste.

List of Abbreviations

CAE	Computer-aided engineering
DOE	Department of Energy
GPL	General Public License
I/O	Input/output
IDE	Integrated development environment
IVF	Intel® Visual Fortran
MSL	Mean sea level
NREL	National Renewable Energy Laboratory
NWTC	National Wind Technology Center
SI	International System of Units
SVN	Subversion
TOT	Top of trunk

Nomenclature

Δt	Discrete time increment
n	Discrete-step counter
p	System parameters
t	Time
u	System inputs
U	Input-output transformation functions
x	Continuous states
X	Continuous-state functions
X_i, Y_i, Z_i	Axes of the inertial frame
x^d	Discrete states
X^d	Discrete-state functions
y	System outputs
Y	System-output functions
z	Constraint states
Z	Constraint-state functions

Table of Contents

List of Figures

List of Tables

Introduction

Over the past two decades, the U.S. Department of Energy (DOE) has sponsored the National Renewable Energy Laboratory (NREL)'s development of computer-aided engineering (CAE) tools for wind turbine analysis. The tools are developed as free, publicly available, open-source, professional-grade products as a resource for the wind industry. The tools are used by thousands of wind turbine designers, manufacturers, consultants, certifiers, researchers, students, and educators throughout the world. NREL has recently put considerable effort into improving the overall modularity of its core CAE tool, FAST[1,2], to accomplish the following benefits: (1) improved ability to read, implement, and maintain source code; (2) increased module sharing and shared code development across the wind community; (3) improved numerical performance and robustness; and (4) enhanced flexiblity, enabling further development of functionality without the need to recode established modules.

As more individuals and organizations modify FAST (or existing CAE tools) or develop new modules for it, it becomes increasingly important to standardize development activities. It will take concentrated effort to ensure that software being modified simultaneously by independent entities is compatible and can be integrated into one version of quality software. (We do not have resources to support multiple versions of software with different features that have to be separately maintained.) The new modularization structure[3] for FAST provides the standard framework for all new development of tools in the FAST CAE tool framework. Reference 3 explains the features of the new FAST modularization framework, as well as the concepts and mathematical background needed to understand and apply it correctly. This handbook explains the code development requirements and best practices for the FAST modularization framework. Much of the guidance in this document also applies to other tools developed at the National Wind Technology Center (NWTC).[*]

Overview

It takes careful thought to write quality software, and that goal should be kept in mind from the beginning of each development effort. The extra time spent writing quality software will help reduce the amount of time it takes to track down problems in the software, and will allow more people to benefit from the software that has been written.

This document is designed to assist developers in writing software to be used with FAST and other NWTC CAE tools,[†] helping to make the software readable, portable, and robust. The document, however, cannot completely cover everything that is necessary for developing quality software, so, we recommend that developers also reference programming handbooks, the documentation that comes with their compilers, and other resources for writing software.[‡]

[*] Please note that the current FAST source code (v7.01.00a-bjj) does not yet conform to this framework. Effort is being made to convert to this new framework.

[†] The text in this document specifies which guidance in this document is specific to FAST; when not specified, the guidance applies to all of the NWTC CAE tools.

[‡] The section "For Further Reading" at the end of this document gives some suggestions.

Developers are encouraged to read this document in its entirety before planning new development efforts or writing any source code. It may also be helpful to keep the steps for code maintenance specified in Appendix D in mind while reading the rest of the document.

Definitions

Because some of the words used in software development have ambiguous meanings, the following definitions are included to reduce the chance of miscommunication.

Alpha version: Software tested internally and possibly externally by a small number of selected users. The source code may contain comments that indicate how it differs from the previous beta version.

Archive: A collection of many files stored in a single file, often in ZIP or TAR file format. Archives of NWTC CAE tools are used to easily distribute all of the files associated with a particular version of the tool, keeping the directory structure.

Beta version: Software tested by a number of users and distributed externally to a number of users, with any change comments from alpha versions removed. Software in beta form has updated documentation (user's guide and/or theory manual).

Branch: A directory in a Subversion repository that contains development versions of a CAE tool. When branch directories have been sufficiently tested and approved by the CAE tool's primary owner, they are merged back into the repository's trunk directory.

Code: See **Source code**

Coupling interval: The simulation-time increment at which the driver program will make calls to a given module at subsequent simulation times.

Debug mode: A configuration of compiler settings that provides debugging information and disables (or reduces, depending on the compiler) optimizations.

Driver program: An executable (main) program that calls a module. Driver programs are often used to test the functionality of modules (modules do not contain main PROGRAM statements). Appendix I contains an example driver program for the module provided in Appendix E.

Glue code: The driver program that couples (or glues) individual modules together. The "Modular Interface and Coupler" in Figure 1 is the glue code in the FAST framework.

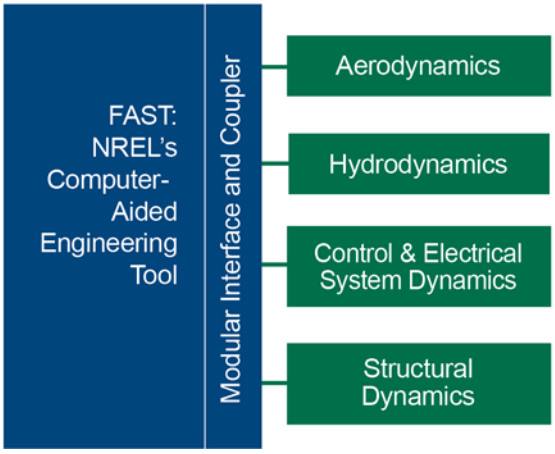


Figure 1. NREL's modular CAE tool, FAST.

Inputs: A set of values supplied to a system that, together with the states, are needed to calculate future states and/or the system's output.

> *Meshed inputs* are inputs that are defined on a discretized boundary characterizing the outer extent of the system.
>
> *Nonmeshed inputs* are inputs that need not be represented on this discretized boundary.

Library: A collection of software resources—such as subroutines, constants, and type specifications—that can be used by other programs. A library itself is not an executable program.

Local variable: A variable accessible only from the subroutine or function in which it is defined.

Loose coupling: A time-integration scheme where two or more modules exchange data (inputs and outputs) through the glue code at coupling intervals, but each module tracks its own states and integrates its own equations with its own solver. (See Figure 2.)

Mesh: A discretization of a domain into a set of discrete sub-domains. A mesh is comprised of a set of nodes (simple points in space) and the connectivity of the nodes (elements).

Module: A separable component of the FAST framework (e.g., the green boxes in Figure 1); a Fortran MODULE is a program unit that contains specifications and definitions that can be used by another unit of the program. Each component of the FAST framework may contain multiple Fortran MODULEs.

Outputs: A set of values calculated by and returned from a system and that depend on the system's states, inputs, and/or parameters.

> *Meshed outputs* are outputs that are defined on a discretized boundary characterizing the outer extent of the system.
>
> *Nonmeshed outputs* are outputs that need not be represented on this discretized boundary.

Parameters: A set of internal system values, independent of the states and inputs, that can be fully defined at initialization (possibly with time dependence that can be fully prescribed at initialization) and characterize a system's state equations (differential, difference, and/or constraint) and output equations.

> *Meshed parameters* are parameters that are defined on the system's discretized domain.
>
> *Nonmeshed parameters* are parameters that need not be represented on this discretized domain.

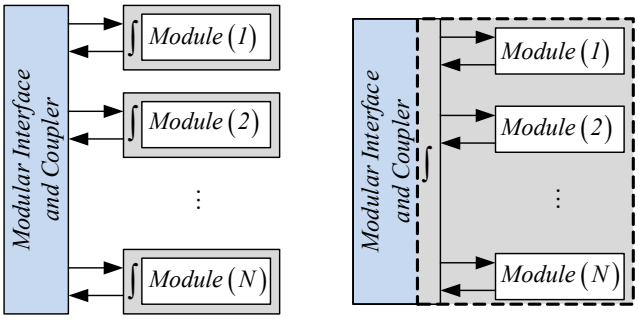


Figure 2. Loose- (left) and tight- (right) coupling schemes.

Primary owner: The primary person responsible for maintaining a particular

CAE tool, assigning development responsibility, and accepting all changes to that code. There may be multiple people developing the same tool, but there is only one owner. The name of each owner for the NREL CAE tools is listed next to the link for its SVN repository on http://wind-dev.

Release mode: A configuration of compiler settings that optimizes the code and does not provide debugging information. Code compiled in release mode almost always runs significantly faster than that compiled in debug mode.

Released version: A version of software that is available for distribution to users, primarily through the NWTC Design Tools web site. Released versions can be either alpha or beta versions, but it is assumed that the version has been sufficiently tested and documented so that nondevelopers can use the tool.

Source code: Text in the format and syntax required by the programming language it is written in. Often used interchangeably with the words "software" or "computer program."

States: A set of internal values of a system used to calculate future state values and/or the system's output if the inputs to the system are supplied.

> *Constraint states* are states that are not differentiated or discrete (i.e., constraint states are algebraic variables) and are characterized by algebraic constraint equations (i.e., equations without time-derivatives of constraint states).
>
> *Continuous states* are states that are differentiable in time and are characterized by continuous-time differential equations.
>
> *Discrete states* are states that only have a value at discrete steps in time and are characterized by discrete-time difference equations.
>
> *Meshed states* are states (continuous, discrete, and/or constraint) that are defined on the system's discretized domain.
>
> *Nonmeshed states* are states (continuous, discrete, and/or constraint) that need not be represented on this discretized domain.

Tight coupling: A time-integration scheme where each module sets up its own equations, but the states are tracked and integrated by a solver common to all of the modules. (See Figure 2.)

Trunk: A directory in a Subversion repository that contains the main development version of a CAE tool. The trunk is assumed to contain code that is stable and has passed all of its version checking.

Type: (also "data type") A classification of data that determines the data's properties. Examples of intrinsic data types include integer, floating-point, Boolean, and character values. Derived data types are user-defined, and are sometimes called "data structures."

Validate: To compare software predictions with test data.

Verify: To compare software predictions with hand calculations or predictions from other software.

Version checking: (also "regression testing") Comparing output from a new software version with output from previous versions.

Copyright and Licensing

Historically, NWTC CAE tools have been distributed under the Data Use Disclaimer Agreement found on our web site: http://wind.nrel.gov/designcodes/disclaimer.html. However, at the time of this writing, we are starting to release NWTC CAE tools under the GNU General Public License (GPL) v3.0 open-source license.[4] Some codes may also be offered under other less restrictive open-source licenses on a per-case basis.

All source code files must have copyright and license agreement information listed at the beginning of the file. See the "LICENSING" section in the module in Appendix E of this document for an example.

Version Naming

Software that is used by more than just one person should have a way of tracking versions. The program should display the version number when it runs and—if possible—include the version number in any output files it creates.

If you make changes to NWTC software, please change the variables containing the version number and date. Alpha versions have the form "v0.00.00a-bjj" where "0.00.00" is the version number, the "a" represents the alpha revision ("a", "b", "c", …), and the "bjj" should be replaced with your initials (or organization name). Beta versions released by the NWTC do not contain initials or alpha revision characters. The number to the left of the first decimal in the version number should be incremented for major rewrites, overhauls, and major upgrades. The number between the decimals should be incremented when features are added, and *must* be incremented when the input file changes. The number to the right of the second decimal is for bug fixes, changes to output, and minor changes. To avoid confusion between the month and day when writing dates, use text instead of numbers for the month. We recommend using "dd-mmm-yyyy" format for all dates (e.g., 08-Aug-2012).

Version Control

Coordinated multi-user development of the NWTC's CAE tools is managed under Subversion[5] (SVN), an automatic software version control and management system in wide use by the open source software development community. The SVN system:

- Provides developers at NREL and at remote locations with network access to source code in a centrally managed and backed-up code repository,
- Maintains a history of the code that allows retrieving and displaying earlier versions of the code, change logs, and differences between versions,

- Supports multiple developers working on the code, automatically detecting and reporting potentially conflicting modifications,
- Permits for read and modify access on a per-project and per-user basis, and
- Provides notification and an audit trail of modifications so that no change can be made to the source code without a record of the change and the individual making the change.

An overview of SVN is provided in Appendix A of this document.

The central data storage in Subversion is called a "repository." Each NWTC CAE tool has its own repository, and each repository is organized into three directories called "trunk," "branches," and "tags." The trunk directory holds the latest stable (working) copy of the CAE tool. The branches directory holds development versions of the trunk, and the tags directory contains copies of the trunk that are tagged with specific version numbers. Appendix B discusses the NWTC's policies for using the repositories.

Documentation

Software is much easier to understand and maintain when it is well documented. Documentation should include comments in the source code, change logs, user's guides, theory manuals, and sample input files and test cases.

Commenting the Code

Software developers must document their source code. Appropriate comments in the source code will aid in modifying existing code, adding new features, and finding errors.

Each variable in the source code must contain comments that indicate its purpose and—if applicable—what its units are (e.g., meters or seconds).

Comments should be used to indicate logical sections within the source code as well as to indicate what the code is intended to do and how it works. The intended functionality might not be obvious to anyone looking at the code later (sometimes not even to the person who wrote the code). For example, if a particular algorithm is being used, the comments should clearly indicate how it works. If it is a new or complicated algorithm, cite a paper or other source where the algorithm can be found with additional documentation.

See the module in Appendix E of this document for an example of how to add comments in source code.

Change Logs

Software must have a separate log file that lists what was changed in each released version. These change log files—typically named "ChangeLog.txt"—provide a history of the code's development and are posted on our web site with released versions of the code to allow users to see what has changed before they download the entire archive.

For each version of the software, the change log should contain the version number, date, and name of the individual(s) making the change(s). Briefly describe changes to the code and reasons behind them. Indicate whether the changes impact any results. Changes to test cases or to other

documents that are released with the code should also be listed in the change log. The log files do not need to include changes that are transparent to the user (e.g., a variable in the source code was renamed).

Subversion Commit Logs

When you commit changes to the SVN repository, you should add comments to the Subversion log. These comments (commit/log messages) should indicate to other developers what you have changed and why (new routines, variable name changes, restructuring, etc.). Good commit comments make it possible to find a previous version if something doesn't work as expected. These comments will also be referenced by the tool's primary owner when SVN branches are merged back into the main trunk.

You may wish to create a developer's change log file to keep track of all your changes as you are working to help you remember what you have done and why the changes were made. You can then copy the text from this file into the SVN log when you commit the changes to the repository.

User's Guides and Theory Manuals

Beta versions of software must have a well written user's guide and theory manual. A user's guide helps people understand how to use the code. It should contain a description of the inputs (input files), outputs (output files), and anything else required to run the code. Figures are especially helpful. Theory manuals should document and explain equations and algorithms that are implemented in the code.

Sample Input Files and Test Cases

Users often rely on sample input files and test cases to learn how to use new software. Sample input files should contain accurate descriptions of the input parameters (including units where appropriate). Many users will use these descriptions to understand the parameter instead of taking the time to look it up in the user's guide. Users also often take the sample test cases and modify them only slightly, which can transfer errors and simplifications from the sample files to many research projects. Please provide adequate documentation about the test cases, document any simplifications you have made (e.g., you reduced the number of analysis points to make the distribution file smaller), and carefully check that the input data does not contain errors.

Writing Source Code

Writing quality software for distribution takes careful thought and planning. You must follow the development policy described in Appendix B and the steps outlined in Appendix D of this document. If you are modifying existing code, take the time to understand what it does before making changes in the source code. Pay attention to details, and try to keep the big picture in mind, especially if you are modifying a complex code. Your changes may work for the case you intend, but think about how it will affect other cases and the other features of the code. Are there cases where the feature you have added does not apply or contradicts other physics of the model? Could numerical problems occur (division by zero, square roots of negative numbers, round-off errors, value outside of valid range, etc.)? Be sure to address these (and other) issues so that your code does not adversely affect the rest of the software (or other codes it links with). It is

important to keep in mind that many people will be using and compiling your software, probably in ways you didn't expect.

General Requirements

Keep the following requirements in mind while you are developing software for the NWTC:

- Modules need to be dynamically allocatable, allowing multiple instances to exist simultaneously (e.g., multiple FAST simulations of turbines linked together in another simulation to model a wind farm).
- NWTC CAE tools should be able to link with codes compiled in Fortran or C/C++.
- NWTC CAE tools should be able to run on Windows® or Linux platforms.
- NWTC CAE tools should be able to be compiled with Intel® Visual Fortran (IVF) and gfortran[§].
- NWTC CAE tools may be linked with multiple driver programs (e.g., FAST can be a standalone program or used in Simulink; AeroDyn can be used in FAST, MSC.Adams, SIMPACK, and FEDEM).
- Modules written for the FAST framework must adhere to the requirements of the template in Appendix E and described in this document.

Planning

Before writing source code, developers must outline a plan (or pseudocode) of the changes and/or additions they intend to make and discuss it with the tool's primary owner and any other developers working on the same CAE tool. The plan should be written in a document that will help create a user's guide or theory manual; the plan should indicate what the code (and each part of it) is supposed to do and how it should interact with other modules.

Decide whether you need to modify existing code or implement a new module. (New theory should be implemented in new modules.) For modules written in the FAST framework, write out all equations and algorithms in the form required for the template in Appendix E and Appendix F of this document. Determine whether the underlying formulation fits the requirements of tight coupling, and choose whether tight and/or loose coupling will be used. If the module can be tightly coupled, we recommend that you implement it in such a way that both loose and tight coupling are available options. For loose coupling, be sure to write out the formulation for the time integration solver you will implement. Clearly define the inputs, outputs, states (continuous, discrete, and constraint), and parameters for the module; make sure that all inputs can be derived from the outputs of other modules, and document the input-output transformation equations (U), not including mapping between meshes.

It may be tempting to skip the planning step in favor of writing code, but ***this step is the most important and probably most time-consuming task in the software development process***. Careful planning will reduce the amount of time spent tracking down bugs and fixing problems in your source code.

[§]We assume that the tools should be able to be compiled with current versions of the IVF and gfortran compilers. Some features may not be available in earlier versions of the compilers.

Module Structure

Software written for the FAST framework must be formulated to fit the template listed in Appendix E of this document. This framework standardizes the interface between modules, and it has been designed to allow both loose and tight coupling with the possibility for linearization (as well as other features)[3]; Table 1 lists the subroutines required for each option.[**]

Each of the subroutines listed in Table 1 operate on data structures (types) defined for the module. These derived data types are listed in Table 2. To avoid potentially circular build dependencies, types defined and used by a module are defined in a separately compiled module named "ModuleName_Types" that will then be used by the "ModuleName" module itself.

The ModuleName_Types module will be generated automatically using the FAST Registry supplied by the NWTC as described in Appendix H. Developers will create a table of the data for each type using the format described in Appendix H, and the FAST Registry will generate the type definitions as well as the system input and output extrapolation/interpolation subroutines, copy/destroy subroutines, and the pack/unpack subroutines. This automation will reduce the chance for errors that may occur in writing the number of subroutines required. The remaining subroutines (initialize/end, time-stepping, and Jacobian) will be written by the developer using the template for the ModuleName module in Appendix E.

Other than replacing the *ModName* text,[**] you should not modify the subroutine statements in the template module when you implement them in your modules. Do not add, remove, or change the order of any subroutine arguments, and do not change the INTENT attributes. Keep all of the subroutines and types defined in the sample modules, even if your module does not have a particular feature. For example, if your module does not have constraint states, do not delete the **ModName_CalcConstrStateResidual** subroutine or the data type definition of **ModName_ConstraintStateType**.

[**] Note that *ModuleName* in the template module would be replaced by the name of the module being developed. *ModName* would be replaced by either the name of the module being developed or an abbreviation of it.

Table 1. Subroutines required for the FAST modular framework.

Template Requirements	Generated by Registry	Loose	Tight (Time Marching)	Tight (Linearization)
Initialize/End Subroutines				
• ModName_Init		✓	✓	✓
• ModName_End		✓	✓	✓
Time-Stepping Subroutines				
• ModName_UpdateStates		✓		
• ModName_CalcOutput		✓	✓	✓
• ModName_CalcContStateDeriv			✓	✓
• ModName_UpdateDiscState			✓	✓
• ModName_CalcConstrStateResidual			✓	✓
Jacobian Subroutines				
• ModName_JacobianPInput			✓	✓
• ModName_JacobianPContState				✓
• ModName_JacobianPDiscState				✓
• ModName_JacobianPConstrState			✓	✓
Input/Output Extrapolation/Interpolation Subroutines				
• ModName_Input_ExtrapInterp	✓	✓	✓	✓
• ModName_Output_ExtrapInterp	✓	✓	✓	✓
Pack/Unpack Subroutines				
• ModName_Pack	✓	✓	✓	✓
• ModName_Pack{*TypeName**}	✓	✓	✓	✓
• ModName_Unpack	✓	✓	✓	✓
• ModName_Unpack{*TypeName**}	✓	✓	✓	✓
Copy/Destroy Subroutines				
• ModName_Copy{*TypeName**}	✓	✓	✓	✓
• ModName_Destroy{*TypeName**}	✓	✓	✓	✓

**TypeName* is the name of the data type to be operated on; it is one of the following values: *Param, Input, Output, ContState, DiscState, ConstrState, OtherState, POutputPInput, PContStatePInput, PDiscStatePInput, PConstrStatePInput, POutputPContState, PContStatePContState, PDiscStatePContState, PConstrStatePContState, POutputPDiscState, PContStatePDiscState, PDiscStatePDiscState, PConstrStatePDiscState, POutputPConstrState, PContStatePConstrState, PDiscStatePConstrState, PConstrStatePConstrState*

Table 2. Derived data types required for the FAST modular framework.

Template Data Types	
Type Name	Purpose
• ModName_InitInputType	Initialization input data
• ModName_InitOutputType	Initialization output data
• ModName_InputType	System inputs
• ModName_OutputType	System outputs
• ModName_ParameterType	System parameters
• ModName_ContinuousStateType	Continuous states
• ModName_DiscreteStateType	Discrete states
• ModName_ConstraintStateType	Constraint states
• ModName_OtherStateType	Other states
• ModName_PartialOutputPInputType	Partial derivative of output equations with respect to inputs
• ModName_PartialContStatePInputType	Partial derivative of continuous state equations with respect to inputs
• ModName_PartialDiscStatePInputType	Partial derivative of discrete state equations with respect to inputs
• ModName_PartialConstrStatePInputType	Partial derivative of constraint state equations with respect to inputs
• ModName_PartialOutputPContStateType	Partial derivative of output equations with respect to continuous states
• ModName_PartialContStatePContStateType	Partial derivative of continuous state equations with respect to continuous states
• ModName_PartialDiscStatePContStateType	Partial derivative of discrete state equations with respect to continuous states
• ModName_PartialConstrStatePContStateType	Partial derivative of constraint state equations with respect to continuous states
• ModName_PartialOutputPDiscStateType	Partial derivative of output equations with respect to discrete state
• ModName_PartialContStatePDiscStateType	Partial derivative of continuous state equations with respect to discrete states
• ModName_PartialDiscStatePDiscStateType	Partial derivative of discrete state equations with respect to discrete states
• ModName_PartialConstrStatePDiscStateType	Partial derivative of constraint state equations with respect to discrete states
• ModName_PartialOutputPConstrStateType	Partial derivative of output equations with respect to constraint states
• ModName_PartialContStatePConstrStateType	Partial derivative of continuous state equations with respect to constraint states
• ModName_PartialDiscStatePConstrStateType	Partial derivative of discrete state equations with respect to constraint states
• ModName_PartialConstrStatePConstrStateType	Partial derivative of constraint state equations with respect to constraint states

The data types and subroutines required for the template modules are described below; the inputs and outputs for subroutines contained in the ModuleName module are also pictured in Appendix F of this document.

Data Types

All of the variables used in a module during time-stepping operations—except local variables—must be placed in one of the user-defined derived data types in Table 2. There are separate derived data types for system inputs, outputs, states, and parameters[††] using the definitions found at the beginning of this document, as well as initialization input and output data and partial derivatives of the state and output functions with respect to the states and inputs. (Variables defined as local subroutine variables that do not retain their values between subroutine calls do not need to be stored in one of these data types.) States must be divided into continuous, discrete, constraint, and other states. The derived data types may themselves contain variables of derived data types, but INTEGER, LOGICAL, or CHARACTER variables are allowed only in the derived data types for initialization input and output, parameters, and other states.

The other states are stored in a data type called "ModName_OtherStateType." In tight coupling, this data type must only be used for code efficiency purposes (e.g., storing an index into a lookup table or previous calculations so they don't have to be redone) and cannot be used to impact the results of the calculations. In loose coupling, the other states data type may also store states that don't fit the definitions of continuous, discrete, or constraint. Another possible use of the ModName_OtherStateType is to store previous values of continuous states if a multistep time-integration method is employed in a loosely coupled approach. In this case, the ModName_OtherStateType data type should include the last k continuous states (i.e., it would likely store an array of the previous k continuous states) where k is the order of the method.

Data of type ModName_OtherStateType can be updated in any subroutine, and the glue code does not associate it with a particular time increment. Unless otherwise specified in this document, we cannot guarantee that the glue code will call subroutines in a particular order or that time will always be advancing. Thus, the module developer must take care when using ModName_OtherStateType, making sure to store any switches or time it needs in the type, and providing appropriate checks for errors.

Each module may define and pass data using its own discretization scheme(s), and separate meshes (schemes) may be used for input, output, parameter, and state data. The mesh structure is discussed further in the "Meshes" section of this document.

All of the derived data types are defined in the ModuleName_Types module, which will be generated using the FAST Registry described in Appendix H.

Input and Output Extrapolation/Interpolation Subroutines

The glue code may keep a time history of the system inputs and outputs of each module, and it is desirable to use these time histories to find values at specific times. Thus, the system input and output derived types for each module must have corresponding subroutines to extrapolate and interpolate based on the time history. These subroutines fit a polynomial through the time-history

[††] Note that system parameters do not need to be defined with the Fortran PARAMETER attribute. System parameters are values that do not change after initialization, but they do not need to be defined at compile time.

data (the order of the polynomial is dictated by the number of time steps in the time history) and evaluate the polynomial to find the system inputs or outputs at a specified time. These subroutines are defined in the ModuleName_Types module, which will be generated using the FAST Registry described in Appendix H.

Copy/Destroy Subroutines

Because the template module allows the developer to use POINTERS, meshes (which have POINTERS in them; see Appendix G of this document), and ALLOCATABLE arrays in user-defined data structures, each derived type must have corresponding subroutines to copy and destroy the derived data type. These subroutines are defined in the ModuleName_Types module, which will be generated using the FAST Registry described in Appendix H.

Pack/Unpack Subroutines

It is sometimes necessary to stop large simulations and restart them later. The subroutines **ModName_Pack{*TypeName*}** and **ModName_Unpack{*TypeName*}**[‡‡] allow codes developed in the FAST framework to have this capability. Each derived data type has a corresponding subroutine to pack (save) its data into separate arrays of type REAL(ReKi), REAL(DbKi) and INTEGER(IntKi), and a subroutine to unpack (retrieve) the derived type from the three arrays. All of the pack and unpack subroutines are defined in the ModuleName_Types module, which will be generated using the FAST Registry described in Appendix H. These subroutines may also be useful in mixed-language programming.

Initialize/End Subroutines

The **ModName_Init** subroutine—defined in the ModuleName module—is designed to be called one time at the beginning of each simulation for each instance of the module (e.g., individual turbine). It is the first subroutine that will be called from the ModuleName module. It performs initialization tasks for the module, including reading input files and setting up any meshes.[§§] The subroutine returns the system parameters, an initial guess for the input to the system, the initial states (the constraint states should be calculated using the initial guess for the system inputs, thus the constraint states are also considered a guess), and time increment for loose coupling and discrete states. The glue code (driver program) will supply a suggestion for the module's coupling interval, but the initialization routine may override this value.

The **ModName_End** subroutine is designed to be called one time only for each instance of the module, at the end of a simulation. It is the last subroutine that will be called from the ModuleName module. Its task is to release memory and close files.

Time-Stepping Subroutines

All of the time-stepping subroutines are defined in the ModuleName module. The driver program sends the current simulation time—stored as a double-precision real number—to all of the time-stepping subroutines.

[‡‡] *TypeName* is any one of a list of data types as defined in the footnote of Table 1. For example there will be subroutines named **ModName_PackParam**, **ModName_PackInput**, **ModName_PackOutput**, etc.

[§§] Meshes that follow meshes from other modules should be initialized in an undeflected position to allow the mapping between meshes to be set up properly. See the "Coupling Modules Together" section of this document.

The **ModName_UpdateStates** subroutine is called in a loose coupling scheme at the coupling interval defined in the module initialization. It is given the simulation time and step, system parameters, and a time history of system inputs along with an array of the times associated with those inputs. The states are input with values of continuous, discrete, and constraint states at the simulation time. The subroutine solves for the constraint states and updates the continuous and discrete states to their values at the next coupling interval. If the system inputs are needed for these calculations, subroutine **ModName_UpdateStates** should call **ModName_Input_ExtrapInterp** to access the system inputs at desired times within the time interval.

The **ModName_CalcOutput** subroutine is given the current simulation time, the system inputs and states at the current simulation time, and system parameters. It computes the system outputs at the current simulation time. In loose coupling, **ModName_CalcOutput** subroutine is called at the coupling interval defined in the module initialization, but in tight coupling it may be called at other times.

The **ModName_CalcContStateDeriv** subroutine calculates the time derivatives of the continuous states at the current simulation time. It operates on the inputs and states defined at the current simulation time and the system parameters.

The **ModName_UpdateDiscState** subroutine is called at the coupling interval defined in the module initialization. It is given the simulation time and step, the inputs and states defined at the simulation time, and system parameters. It returns the discrete states updated to their values at the next coupling interval.

The **ModName_CalcConstrStateResidual** subroutine solves for the residual of the constraint state functions (which should be zero when the constraint-state guess is correct) at the current simulation time. It is given the system's parameters and its inputs and states (continuous and discrete states and a guess for the constraint states) at the current simulation time; it returns the residual of the constraint-state functions.

If the module can be tightly coupled, we recommend that you implement it in such a way that both loose and tight coupling are available options (e.g., to minimize coding redundancy, the loose coupling **ModName_UpdateStates** subroutine would call the tight-coupling routines **ModName_CalcConstrStateResidual**, **ModName_CalcContStateDeriv**, and **ModName_UpdateDiscState**).

Jacobian Subroutines

There are four subroutines to compute Jacobians, which are not used by the glue code in loose-coupling schemes. Each subroutine is given the current simulation time, parameters, and system inputs and states at the current simulation time. The subroutines calculate up to four partial derivatives (which are optional arguments): the output (Y), continuous- (X), constraint- (Z), and discrete- (X^d) state functions with respect to a common variable. The **ModName_JacobianPContState** subroutine calculates the Jacobians with respect to the continuous states (x); the **ModName_JacobianPDiscState** subroutine calculates the Jacobians with respect to the discrete states (x^d); the **ModName_JacobianPConstrState** subroutine calculates the Jacobians with respect to the constraint states (z); and the

ModName_JacobianPInput subroutine calculates the Jacobians with respect to the system inputs (u).

For tightly coupled time marching simulation, four Jacobians must be defined: $\partial Z/\partial z$, $\partial Z/\partial u$, $\partial Y/\partial z$, and $\partial Y/\partial u$. To allow linearization in tightly coupled schemes, all sixteen Jacobians must be defined.[***] We recommend that tightly coupled modules always define all sixteen Jacobians available in the template so that the module can be linearized.

If possible, Jacobians should be derived and implemented analytically to give the best numerical-convergence performance. However, if that is impractical, numerical implementations of the Jacobians are acceptable. All of the Jacobian subroutines are defined in the ModuleName module.

Meshes

Where applicable, software components in the FAST framework that use spatial discretizations (meshes) may define them using the ModMesh and ModMesh_Types modules. Inputs, outputs, and states can all be stored as meshed data; however, input and output data on discretized spatial boundaries *must* be stored using the ModMesh and ModMesh_Types modules. These modules contain the data structures and methods used to define meshes and meshed data communicated between components of the software. A mesh is comprised of a set of *nodes* (simple points in space) and their connectivity as specified by their membership as vertices in an *element* (the space between nodes; see Appendix G for examples). No single mesh may contain elements of mixed spatial dimensions (e.g., a mesh cannot contain both a line [1-dimensional] element and a surface [2-dimensional] element). Instead, separate meshes must be created to accommodate these discretizations. One mesh will contain the elements of one dimension; a separate mesh will contain the elements of another dimension. Using separate meshes allows the units of the mesh fields (forces, moments, and added mass) to be defined consistently, which will be necessary for spatial interpolation and mapping of the meshes.

A mesh is associated with one or more *fields* that represent the values of variables or "degrees of freedom" at each node. A mesh always allocates one fields named **Position** and **RefOrientation**; **Position** specifies the location in three-dimensional space as an X_i,Y_i,Z_i triplet of each node, and **RefOrientation** defines the node's reference orientation (as a direction cosine matrix). In addition, the ModMesh module predefines a number of other fields of triples representing velocities, forces, and moments as well as a field of 9 values representing a direction cosine matrix. These fields may be allocated by the module developer at initialization. Detailed descriptions of the ModMesh and ModMesh_Types modules are provided in Appendix G of this document.

The operations on meshes defined in the ModMesh module are considered low-level and include basic operations such as creation, spatio-location of nodes, construction, committing the mesh definition, initialization of fields, accessing and updating field data, copying, deallocating, and destroying meshes. Higher level operations on meshes—such as interpolation and remapping—are being developed in another module.

[***] In equation form: $\partial Y/\partial u$, $\partial Y/\partial x$, $\partial Y/\partial z$, $\partial Y/\partial x^d$, $\partial X/\partial u$, $\partial X/\partial x$, $\partial X/\partial z$, $\partial X/\partial x^d$, $\partial Z/\partial u$, $\partial Z/\partial x$, $\partial Z/\partial z$, $\partial Z/\partial x^d$, $\partial X^d/\partial u$, $\partial X^d/\partial x$, $\partial X^d/\partial z$, and $\partial X^d/\partial x^d$.

The ModMesh routines for creation, construction, committing the mesh definition, and initialization of fields are intended to be called from the **ModName_Init** routine in the FAST framework. Once a mesh is committed and initialized, it may be used in the time-stepping and Jacobian subroutines in the FAST framework. Except for passing meshes between different components and other modules for inter-component mapping of meshes, the driver program (glue code) does not generally know about or use the contents of the MeshType data structures it is forwarding between the components.

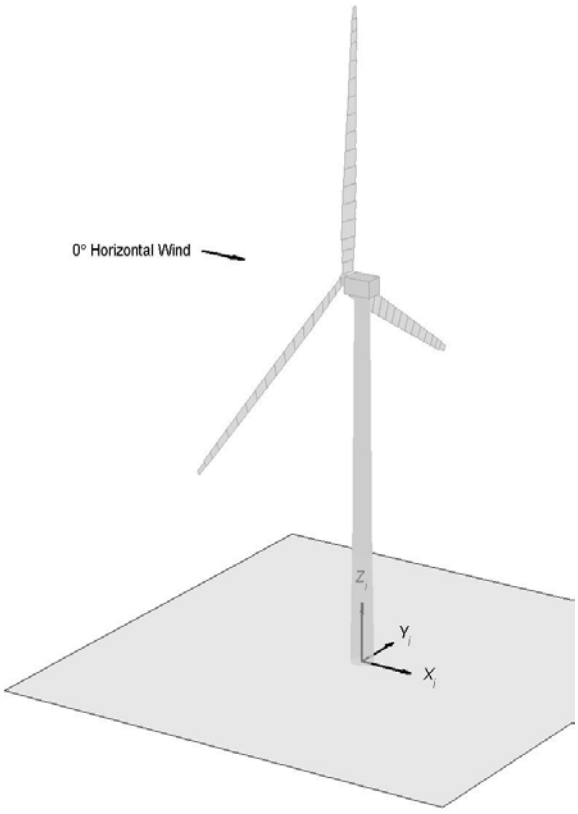


Figure 3. The global coordinate system used for interfaces in the FAST modular framework

Units

Data passed through the module interface in the FAST framework must be stored in the International System of Units (SI)—specifically the SI base units, which includes kilograms, meters, seconds, and radians.

Mesh fields of forces, moments, and added mass must have units that are *per unit length, per unit area,* and *per unit volume* for line, surface, and volume elements, respectively. The forces, moments, and added mass are lumped for point elements.

Coordinate Systems

The coordinate positions and loads passed between modules (through the module interface) in the FAST framework are assumed to be relative to a global coordinate system. The system, denoted X_i,Y_i,Z_i, is the right-handed set of orthogonal axes of the inertial reference frame shown in Figure 3, and is described as follows:

Origin:
For land-based systems: The point where the undeflected tower centerline intersects with the ground.

For offshore systems: The point where the undeflected support structure centerline intersects with mean sea level (MSL).

***X**$_i$* **axis**: Pointing horizontally in the nominal downwind direction (aligned along the 0° horizontal wind direction).

***Y**$_i$* **axis**: Pointing to the left when looking in the nominal downwind direction (i.e., looking along the positive X_i axis).

***Z_i* axis**: Pointing vertically upward opposite to gravity; for vertical towers, along the centerline of the undeflected tower (when the support platform is undisplaced for offshore systems).

Coupling Modules Together

The code that couples individual modules together is referred to as the "glue code." The glue code interconnects the individual modules, derives inputs from outputs (including coordinate transformations and mapping between different discretizations in space and time), and drives the overall solution forward. In tight coupling, the glue code has the added tasks of integrating the coupled system equations using one of its own solvers and driving the linearization calculations.

The task of developing the glue code for NWTC CAE tools is the responsibility of NREL developers; other developers may choose to create their own glue code—particularly for testing—but it is not required. Module developers must, however, take care to specify the exact form of the inputs and outputs (e.g., displacement, velocity, force) and document the input-output transformation equations (including functions *U*) (not including the mapping of meshes) so that NREL can develop the glue code correctly. For the coupling to work properly, it is important for developers to follow the framework provided in Appendix E and the guidelines listed in this document. Module developers should be aware that unless otherwise specified in this document, we cannot guarantee that the glue code will call subroutines in a particular order or that time will always be advancing.

The glue code has two responsibilities regarding meshes: (1) to couple interface meshes (that may be non-matching) by determining nearest-neighbor locations for transfer of data (mapping), and (2) to interpolate data based on the mapping for transfer between modules. The interface region associated with a module must be spatially discretized (meshed) with one or more of the element types described in Appendix G (points, lines, surfaces, and/or volumes) and with the desired location of inputs and outputs specified for each element. As stated above, the meshes need not be "matching;" the interface mesh of one module may be more refined than that of the coupled module.

The mapping between module interface meshes, as described above, is performed at any time step when the "RemapFlag" variable is set to TRUE in the ModMesh module. In cases where the interface meshes do not move relative to each other, the mapping should only be done once at initialization. Alternatively, either module might request a new mapping in the event of significant mesh distortion or significant relative motion between interface meshes. In cases where an interface mesh in a module will follow an interface mesh of another module, both meshes should be initialized in the undeflected position. RemapFlag must be set to FALSE after initialization to ensure that the "follower" module updates the interface mesh such that there is no relative motion between the interface meshes.

Handling Errors

Modules in the FAST modularization framework must never end the calling (driver) program. Intrinsic routines called from modules must not terminate program execution, either.[†††] Instead, each subroutine in the framework contains arguments called "ErrStat" and "ErrMsg," which

[†††] Many intrinsic Fortran routines (ALLOCATE, DEALLOCATE, READ, etc.) have optional error status variables that can be used to prevent the routine from terminating program execution.

allow the module to tell the driver program if an error occurred. The ErrStat argument is an integer value that should be one of the NWTC_Library's error-level parameters: ErrID_None, ErrID_Info, ErrID_Warn, ErrID_Severe, or ErrID_Fatal (see Appendix J). The ErrMsg argument is a string of characters that describes the error (it should be empty if no error occurred). The calling (driver) program will handle the errors as it sees fit (possibly writing to the screen or to a log file).

When a module calls other routines with ErrStat and ErrMsg arguments, it must check that the error level hasn't reached "AbortErrLev" before continuing. One way to handle this error checking is listed below:

```
IF (ErrStat >= AbortErrLev) THEN
   ! clean up local variables/files as appropriate and then exit this routine
   ErrMsg = 'Error in NameOfThisRoutine: '//TRIM(ErrMsg)
   RETURN
ELSE IF ( ErrStat2 /= ErrID_None ) THEN
   CALL WrScr( 'Error in NameOfThisRoutine: '//TRIM(ErrMsg)
END IF
```

Handling I/O

We recommend that developers handle I/O for their modules in a way consistent with FAST's current look and feel. That is, modules read their own input files and have the option to write their own output files. If a module developer wishes to have the driver code collect its output and write it to a single master output file, you should add an array variable called "WriteOutput" to **ModName_OutputType**. This array should contain only the data that the module wants written to the file. The glue code may have to interpolate this array to obtain output at time intervals consistent with other modules. The names and units associated with each column of the "WriteOutput" array should be specified at initialization using array variables called "WriteOutputHdr" and "WriteOutputUnt," which are contained in **ModName_InitOutputType** (see Appendix E for an example).

Module developers are encouraged to keep their I/O in separate subroutines and/or MODULES that can easily be separated out and modified, should a new method be desired. It may be desirable to define and use a derived data type that contains all of the information from the module's input file.[‡‡‡]

Source Code

Source code must be written in either Fortran, C, or C++, taking care to strictly adhere to the programming standard you are using (i.e., do not use a particular compiler's extension to the standard). NWTC CAE tools have traditionally been written in Fortran, so most of our guidelines are specific to Fortran; however, C/C++ programmers are encouraged to read the guidelines and apply them to their own situations. Also note that for simplification purposes, this document will discuss compiler options using IVF for Windows® syntax. Linux and gfortran users should use the table in Appendix K for comparable options.

[‡‡‡] A derived data type containing all input file information could be passed to the module inside the ModName_InitInputType data structure, allowing additional options for reading the module's input data.

NWTC Subroutine Library

The NWTC Subroutine Library[6] is a collection of many general-use routines and constants used in most NWTC CAE tools. The library contains many constants (like π) as well as routines for I/O and numerical operations. A more detailed description is provided in Appendix J of this document. Use this library whenever possible; if you need modifications or a new routine added to it, please contact the developers of the NWTC Subroutine Library.

Fortran 2003 Standard

If you are programming in Fortran, write code that adheres to the Fortran 2003 standard[7]. If that is not possible, isolate all of the nonstandard code into a small subroutine in a separate source file (such as the NWTC Subroutine Library's Sys*.f90 file). To check that your Fortran code adheres to the Fortran 2003 standard, you should compile your code using the **/stand:f03** compiler option.

Guidelines

The following guidelines will help you write code that meets the general requirements listed above. They will also help you avoid some of the more common deviations from the Fortran 2003 standard and some undesirable programming practices that are allowed by the standard:

- Use Fortran's free source form; do not use fixed or tab source forms.
- Do not let your Fortran source code exceed 132 characters per line. Comments exceeding this limit are okay, as long as the comment indicator (!) occurs before column 132, but keep in mind that it is difficult to read long lines in some text editors.
- Do not include any tab characters in the source code. If you need a tab character in a string, use the NWTC Subroutine Library's TAB parameter (ASCII character 9). Be sure to tell your source code editor to convert tabs to spaces. We suggest 3 spaces per tab.
- Do not write IF statements that require short-circuit evaluation. Fortran compilers are not required to evaluate comparisons in a particular order or to any level of completeness (as long as they are logically equivalent), thus we cannot say,

  ```
  IF ( INDEX > 0 .AND. CharVar(INDEX) == 'Y' ) THEN
  ```

  without getting an error if the first condition is FALSE. In this example, you should instead say,

  ```
  IF ( INDEX > 0 ) THEN
     IF ( CharVar(INDEX) == 'Y' ) THEN
  ```

- Variables must be explicitly defined using meaningful variable names.
  - Use IMPLICIT NONE statements.
  - Use INTENT() specifications on variables passed as subroutine and function arguments.
  - Declare variable KINDs using the parameters in the NWTC Subroutine Library's "Precision" module. (This makes it easier to port to other systems.) For example, write

```
REAL(DbKi)      :: DummyDouble ! Example real number
REAL(ReKi)      :: DummyReal   ! Example real number
INTEGER(IntKi)  :: DummyInt    ! Example integer number
```

instead of

```
REAL(8)         :: DummyDouble ! Example real number
REAL(4)         :: DummyReal   ! Example real number
INTEGER(4)      :: DummyInt    ! Example integer number
```

  - Use the same name for variables in input files as they are used in the source code (e.g., if air density is a variable called "AirDens" in the source code, call it "AirDens" in the input file as well).
  - Declare only one variable per line, followed by a comment describing the variable's purpose and a description of its units.
  - Group variable declarations in an intuitive manner.
  - Use ALLOCATABLE arrays instead of pointers whenever possible. (Pointers are harder to maintain, they often cause memory loss, and compilers have a hard time optimizing them.)
  - If you must use the POINTER attribute in a derived data type, you must also provide subroutines to copy and destroy the derived data type.

- Store data in the main program and pass it through subroutine arguments. (It is not thread-safe to use global variables.)
  - Do not use COMMON blocks.
  - Do not store data in modules (i.e., do not declare variables in the specification part of a module; it is acceptable to declare constants with the PARAMETER attribute and to specify type definitions).
  - Do not use the SAVE statement or attribute. (Watch out for variables that are given the SAVE attribute by default.)
  - Do not write code that depends on the **/Qsave** or **/automatic** options to apply the SAVE attribute automatically.
  - Do not initialize local variables in their declaration statements. (Intel Fortran gives these variables the SAVE attribute by default.)
- Explicitly initialize your variables (but not in their declaration statements). Do not rely on the compiler to initialize everything to zero. Your code must not depend on the **/Qzero** compiler option.
- Use generic instances of subroutines/functions if they exist (e.g., use the ABS() function instead of DABS() or IABS(); use the NWTC Library's ReadVar() routine instead of ReadRVar() or ReadIVar()).
- Avoid using EQUIVALENCE statements. (It is very hard to maintain code with these statements, and compilers have trouble optimizing it.)

- DEALLOCATE arrays and pointers when they are no longer needed, including at the end of a program. (Do not rely on the executable program to automatically deallocate them; memory leaks occur in programs like MATLAB® if you fail to do this step.)
- Provide error checking in your code so that it does not crash ungracefully or return nonsensical data. This is especially important on I/O and mathematical operations (read, write, open files, division by zero, logarithms of negative numbers, value outside of valid range, etc.). See the "Handling Errors" section of this document for more details.
    - Use the error checking provided in the NWTC Subroutine Library routines whenever possible.
    - If an error occurs, provide a descriptive error message.
    - Do not use STOP or ABORT statements. Instead, return appropriate error codes so that the calling program can close gracefully (without locked files or leaked memory). The NWTC Subroutine Library includes specific error code parameters that you can use.
    - Consider using the **/traceback** compiler option so that users know where the error occurred if the program does crash.
- Include the PRIVATE statement in your modules if possible. This statement makes the routines and data defined in a module inaccessible to other modules unless they are explicitly marked PUBLIC, reducing the chances of naming conflicts.
- Do not use GOTO statements.
- Do not use numeric statement labels.
- Do not use CONTINUE statements.
- Indent source lines in standard, logical ways (e.g., make sure IF and END IF statements line up). We want to be able to easily read your code.
- Be very careful dealing with input and output files and writing messages to the screen. Such operations are not thread safe.
    - Do not use the `PRINT *` or `WRITE(*,*)` commands. Instead, call the WrScr() subroutine in the NWTC Subroutine Library when you write to the screen. (This keeps all of the writing to the screen in one place that can easily be modified.)
    - If you work with files in your code, use parameters or variable names to specify external I/O unit specifiers. This makes it easier to change the value if there is a conflict with another module's choice of unit specifiers. For example, use
      ```
      READ(MyUn, *, IOSTAT=IOS)  VarName
      ```
      instead of
      ```
      READ(15, *, IOSTAT=IOS)  VarName
      ```
    - To minimize duplication of I/O unit specifiers, use the NWTC Subroutine Library's GetNewUnit() subroutine.
- Be careful when comparing real numbers. Remember that real numbers are approximated on computer systems and some numbers cannot be exactly represented. For example, do not write

```
IF ( a == 1.0 ) THEN
```

because it won't be true if "a" is stored as 0.999999 (which likely represents 1.0). Instead, use the NWTC Subroutine Library's EqualRealNos() function to compare two real numbers:

```
IF ( EqualRealNos( a, 1.0 ) ) THEN
```

This will ensure real numbers are compared in a consistent way that can be easily modified if necessary.

- Whenever possible, loop through arrays in the correct order for your programming language for optimized code. In Fortran, you should vary the left-most index fastest; for example:

```
DO J = 1,10
   DO I = 1,5
      Ary(I,J) = 10*J + I
   END DO
END DO
```

Mixed Languages

Developers may choose to write modules for the FAST framework in C or C++, however the glue code is written in Fortran. The glue code expects the subroutines defined in Table 1 to be written in Fortran, but these subroutines can be "wrapper" routines that call the developer's C or C++ routines.

Derived data types should not be passed through the Fortran/C/C++ interface. The wrapper routines have the responsibility of converting Fortran derived data types into a form that can be passed to and understood by the C/C++ routines. It is left to the developer to decide the best manner to do this within the wrapper code and the contributed modules.

One suggested method is to write the wrapper routines in Fortran, convert the derived data types into buffers (one-dimensional arrays), and then pass these buffers into the corresponding C/C++ routine[§§§]. The buffers should be passed "by reference", i.e., passed as addresses of the first element of the array. (This is the default in Fortran.) On the C/C++ side, the corresponding arguments should be defined as pointers. The C/C++ code may then populate its own data structures from the data in the buffers. The automatically-generated pack/unpack subroutines may be useful for packing and unpacking buffers within the Fortran wrapper subroutines.

Testing

As new modules are developed or existing NWTC CAE tools are modified, it is important to verify results, check that unexpected side effects have been avoided, and validate (if possible). When possible, individual parts of a code should be tested separately before integrating them into a much larger code (particularly if you are working on a complicated algorithm). When the software is complete, it is important to provide new tests so that future updates can test the new features or modules as well.

[§§§] Developers may wish to use Fortran's ISO_C_BINDING intrinsic module.

Verification
When a new feature is added to software, the results must be compared to hand calculations, results of other software, or other known solutions. This verification procedure builds confidence that the developer has implemented the feature correctly. We recommend that verification results be documented and published.

Validation
Validation is a difficult subject. It is subjective, probably expensive, and not always possible; however, it can add value to predictive software. When validating predictive software against test data, the answers will never agree. Errors in the code, simplifications in the model, the use of inaccurate model properties, and even errors in the test data can cause discrepancies. Deciding how close is good enough is a matter of personal opinion. The complexity of the software and the level of its distribution should determine the necessary degree of accuracy, the number of cases to compare, and the level of documentation.

While validation can be useful, *it is not a substitute for verification*. Validation is used to show that the theory is valid; verification shows that the theory was implemented correctly in the software. If you are not implementing new theory, the main focus of your testing should be verification.

Version Checking
New versions of NWTC CAE tools and new modules must not harm existing capabilities. One way to test this is to compare the results of the new version with the results from previous versions for many different test cases. Researchers at the NWTC have developed DOS batch files named "CertTest.bat" to perform this version checking (certification testing) for most of the NWTC CAE tools.[****] The CertTest.bat file runs the CAE tool for many test cases, compares the new results with the results stored from a previous version, and writes differences to a file called "CertTest.out." Any differences must either be corrected or explained and verified (e.g. you fixed a bug).

If you are modifying an existing NWTC CAE tool and believe that it is working properly, run the CertTest.bat file. If you have made changes to any input file(s), you will first have to modify the input files that CertTest.bat runs. We recommend that you first run the CertTest.bat file using an executable compiled in debug mode with subscript bounds checking (this is usually enabled only for debugging and not for final releases) to ensure that the sample cases never cause arrays to exceed their limits. If the CertTest.bat results are satisfactory with the debugging version of the code, run it again with code compiled in release mode.

Testing New Features
Each new module in the FAST framework must have a driver program that demonstrates how the module should be used and that tests this smaller portion of the code. The driver program should call every public subroutine in the module being tested, and it should have its own test cases to compare with previous results, similar to the certification tests provided with NWTC CAE tools.

[****] The NWTC currently does not have scripts for Linux systems.

If you have modified an existing NWTC CAE tool with a new output or feature, you should provide capability to test your feature in future versions. Either modify an existing test case to include the feature (making sure not to remove a test of any other existing feature), or create a new test case. New test cases should be documented and added to the CertTest.bat file and distributed with the software (added to the archive) for future version checking.

Software Distribution

When software has been sufficiently tested, verified, and documented, the tool's primary owner may choose to distribute the software on the NWTC Design Tools web site (or other means). The primary owner of the tool will determine the schedule for releasing a new version. See Appendix D of this document for the steps that lead up to a release.

Software distributed to users as either alpha or beta versions must include the following information:

- All of the source files (if you link with other software that is easily available on the internet or that has licensing restrictions prohibiting redistribution, give specific instructions for obtaining the necessary files instead)
- The input file(s) for the FAST Registry.
- Executable code or library (if applicable) for Windows® (Code must be able to be compiled on Linux, but we do not require an executable code for Linux.)
- Name(s) and version(s) of all other codes the software uses and what compiler was used to generate the executable code
- A file that indicates the order of compilation for the source files
- A driver program (see the "Testing New Features" section of this document for details)
- A change log that contains the full history of previously released versions (see the "Documentation" section of this document for details)
- A user's guide and theory manual (useful for alpha version, but *required* for beta versions of software)
- Sample input files
- Sample test cases, including all files necessary to run them
- Output generated from the sample test cases
- A certification testing program (e.g., CertTest.bat) to compare user's results from the sample test cases with the output included in the software distribution. See the "Version Checking" section of this document for details.
- A list of all files being distributed
- A list of known bugs or limitations
- Any other files that are useful to run, understand, or maintain the software

We recommend that you build an archive to save your results for distribution. We include a file called "Archive.bat" in our distributions, and the script can be run from a command prompt in a Windows® environment to create an archive using WinZip, the WinZip Self-Extractor, and the WinZip Command Line Support Add-On for Windows.[††††]

You should check that the archive runs properly and contains all the necessary files. We recommend that you also check that the sample test cases run on another computer.

After posting new software on the NWTC Design Tools web site, the tool's primary owner (or his/her designee) should announce the release on the NWTC Computer-Aided Engineering Software Tools forum and may wish to send an email to people who have downloaded previous versions of the software. (Appendix D contains specific details for announcing the release.) We recommend that users subscribe to the forum's announcement topic so they are aware of new releases.

Ongoing Improvements to This Document

At the time of this writing, several items are incomplete or are a work in progress. Items that are currently incomplete may cause this document to be updated.

- Existing FAST source code is not fully converted to this framework.
- The ModMesh and ModMesh_Types module have not yet been fully tested, and the code to map the meshes has not been implemented.
- The unpack routines may need to be modified to accommodate the ModMesh data structures.
- Restrictions on SIBLING meshes and relationships between meshes and Jacobians may be added.
- An appendix to aid module developers in categorizing their data into inputs, outputs, states (continuous, discrete, constraint, and other), and parameters is being developed.
- Recommendations for FAST input and output files may be updated.
- Guidance on coupling FAST modules to commercial software (e.g., commercial software as the driver program) may be provided.
- Guidance on time integration methods may be provided.
- The Licensing may change to GPL 2.0 instead of GPL 3.0.

[††††] Because almost all of our development to date has been on Windows®, we do not currently have scripts for Linux. We plan to also distribute files in the TAR (or tar.gz) format for Linux/Mac users.

References

[1] Jonkman, J. M. and Buhl Jr., M. L. *FAST User's Guide*. NREL/EL-500-38230. Golden, CO: National Renewable Energy Laboratory, August 2005. Accessed June 13, 2012: http://wind.nrel.gov/designcodes/simulators/fast/FAST.pdf

[2] Jonkman, J. M. *FAST Theory Manual*. NREL/TP-500-32449. Golden, CO: National Renewable Energy Laboratory (forthcoming).

[3] Jonkman, J. M. "The New Modularization Framework for the FAST Wind Turbine CAE Tool." *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, 7–10 January 2013, Grapevine (Dallas/Ft. Worth Region), TX* [online proceedings]. URL: http://arc.aiaa.org/doi/pdf/10.2514/6.2013-202. AIAA-2013-0202. Reston, VA: American Institute of Aeronautics and Astronautics, January 2013; NREL/CP-5000-57228. Golden, CO: National Renewable Energy Laboratory.

[4] "The GNU General Public License v3.0" Free Software Foundation, Inc. Accessed June 23, 2012: http://www.gnu.org/licenses/gpl-3.0.html

[5] "Apache Subversion." The Apache Software Foundation, 2012. Accessed June 18, 2012: http://subversion.apache.org

[6] Buhl, M.L., Jr. "NWTC Design Codes (NWTC Subroutine Library)." National Renewable Energy Laboratory, Last modified February 21, 2012. Accessed June 16 2012: http://wind.nrel.gov/designcodes/miscellaneous/nwtc_subs

[7] INCITS/ISO/IEC 1539-1:2004 "Information Technology – Programming Languages – Fortran – Part 1: Base Language."

8"RabbitVCS." The RabbitVCS Team, 2011. Accessed June 19, 2012: http://rabbitvcs.org

[9] "TortoiseSVN." The TortoiseSVN Team, 2012. Accessed June 19, 2012: http://tortoisesvn.net

For Further Reading

Adams, J. C.; Brainerd, W. S.; Hendrickson, R. A.; Maine, R. E.; Martin, J. T.; and Smith, B. T. *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*. 1st edition. Springer, 2009.

Brainerd, W. S. *Guide to Fortran 2003 Programming*. 1st edition. Springer, 2009.

Buhl, M. L., Jr.; Green, H. J. *Software Quality-Control Guidelines for Codes Developed for the NWTC*. NREL/TP-500-26207. Golden, CO: National Renewable Energy Laboratory, 1999. Accessed June 13, 2012: http://wind.nrel.gov/designcodes/papers/TP-500-26207.pdf

Buhl, M. L., Jr. *Code Maintenance*. Golden, CO: National Renewable Energy Laboratory, 2005. Accessed June 13, 2012: http://windinternal.nrel.gov/tips/CodeMaintenance.pdf (internal only)

Chapman, S. J. *Fortran 95/2003 for Scientists and Engineers*. 3rd edition. McGraw-Hill, 2008.

Collins-Sussman, B.; Fitzpatrick, B.W.; and Pilato, C.M. *Version Control with Subversion*. O'Reilly Media, 2004. Accessed June 18. 2012: http://svnbook.red-bean.com

Gasmi, A.; Sprague, M. A.; Jonkman, J. M.; and Jones, W. B. "Numerical Stability and Accuracy of Temporally Coupled Multi-Physics Modules in Wind-Turbine CAE Tools." *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, 7–10 January 2013, Grapevine (Dallas/Ft. Worth Region), TX* [online proceedings]. URL: http://arc.aiaa.org/doi/pdf/10.2514/6.2013-203. AIAA-2013-0203. Reston, VA: American Institute of Aeronautics and Astronautics, January 2013; NREL/CP-500-57298. Golden, CO: National Renewable Energy Laboratory.

Appendix A: Working With Subversion

The two key objects that SVN manages for a developer are the *repository* itself and the developer's *working copy* of the code.

SVN Repositories

The SVN repository stores files and directories for a project and is like an ordinary file system except that SVN also remembers every change that was made to the repository, stretching back to when the project was first imported into SVN. Each revision of the code is recorded and tagged with a unique sequentially increasing revision number, which allows a user to refer to and—if necessary—retrieve any specific revision of the code. Revisions may also be referred to using dates or user-specified tags, a more easily remembered string that can be assigned to refer to a particular revision. The NWTC uses tags to label specific revisions of the code that correspond to releases of a code. Where no revision number is specified, the SVN operation will apply to the most recent revision of the code, sometimes called the "head of the repository" or "top-of-trunk" (TOT).

The repository is maintained by an SVN server program, which runs on the computer that stores the SVN code repository. The server is set up and managed by system administrators. Developers never work with the SVN server directly; rather, developers use an SVN client program that communicates over the network with the SVN server. The SVN client on Linux systems is the **`svn`** command with its options and arguments. (RabbitVCS[8] also provides free graphical tools to access SVN on Linux.) TortoiseSVN[9] is a free SVN client for Windows; it provides a graphical interface that can be accessed by right-clicking the Desktop background or specific files and directories being managed under SVN. The underlying SVN operations are the same under both Linux and Windows clients.

Working Copies

A working copy is an SVN-managed mirror copy of a project's files and directories in the repository on the user's computer. These look just like other files and directories except that they're managed by SVN. Most SVN commands must refer to an SVN working copy to have any effect. Issuing an SVN command in a directory that is not a working copy will generate an error message:

```
% svn status
svn: warning: '.' is not a working copy
```

Each directory of a working copy will contain a subdirectory named **`.svn`**[*]. The contents of these **`.svn`** directories is not important to the user (do not modify them), but their existence indicates that a directory and its contents are part of an SVN working copy and not just a normal set of files on the computer. On Windows, the TortoiseSVN client will mark the icons for working copies with a check in a green dot or an exclamation point in red dot, which also

[*] The dot at the beginning of the directory name usually prevents it from being listed. To list these directories under Linux, use **`ls`** with the **`-a`** option; under Windows, set the folder view options to show hidden files.

indicates whether the working copy has been modified with respect to the repository on the SVN server (Figure 4[†]).

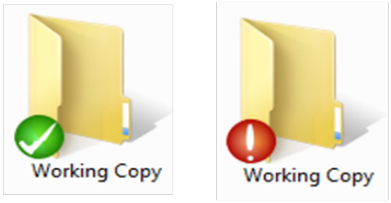


Figure 4. Example Windows® icons modified by TortoiseSVN to indicate the status of working directories

Workflow

The process for using SVN is "copy-modify-merge." That is, a user checks out a working copy of the project files from the repository, works on this local copy, and then, when ready, merges the resulting changes back into the repository. In the meantime, merges by other users working on the project may have altered the repository. SVN handles the mechanics of the version control process, such as detecting changes that conflict.

SVN has no knowledge of the contents or meaning of the files and directories it manages. Other aspects of version control, such as how to actually resolve conflicting merges and deciding when a code change is allowed to be merged back in to the repository, require human participation. The NWTC has developed some policies to deal with some of these issues. Please refer to Appendix B and Appendix C of this document for further details.

Developers use an SVN client program on their local machine to *check out* all or part of the repository as a *working copy*, which they can modify as desired without any effect on the code in the repository. The repository is only modified when a developer *commits* part or all of the working copy back into the repository.

Your changes go back into the repository when you *commit* part or all of your working copy back to the repository. It is only at this point that the repository is modified and a new TOT is created with a new and unique revision number.

In the meantime, you can (and should) *update* your working copy to keep it current with changes other developers are committing to the repository. You can update with respect to the TOT (the current state of the repository) or you can supply a date or revision number in the past and SVN will update your working copy to reflect the state of the repository then. It may occur that another developer's commit changes a line or set of lines in the repository that you have also changed in your working copy. When you update your working copy, SVN checks for and reports any such conflicts. SVN will not allow you to commit the conflicted part of the working copy. It is up to you (and the other developer) to resolve the conflict, perhaps by editing your working copy to accept the other developer's changes, or by working out a different solution with the other developer.

[†] The names of the folders shown in Figure 4 are incidental. Working copies do not have to be named "working copy".

Typical SVN Operations

Developers use an SVN client program on their local machine to access and make changes to the repository. There are a large number of SVN operations, but a typical user developing and contributing code would use this small and relatively simple subset:

- **`svn checkout`** creates a working copy of a repository on the user's local file system. It is necessary to check out a particular working copy only once.
- **`svn info`** provides basic information about a working copy such as the revision number of the working copy and the URL (web address) for the repository in which the working copy is versioned.
- **`svn status`** provides the state of the working copy with respect to the repository. This command will show any files that have been modified in the working copy.
- **`svn diff`** shows the differences between a file or files in the working copy relative to the repository.
- **`svn update`** brings the working copy up to date with respect to the repository, or reports conflicts that need to be resolved before that can happen.
- **`svn add`** marks a file in the working copy to be added to the repository on the next commit, adding it only to the latest revision (and onward).
- **`svn delete`** marks a file in the working copy to be removed from the repository on the next commit, removing the file only from the latest revision (and onward).
- **`svn commit`** causes the non-conflicting changes to the user's working copy to be stored in the repository. The revision number of the repository will be incremented by one, reflecting this new top of the trunk.
- **`svn copy`** is used to tag the top of the trunk with an easy-to-remember name that can be used to retrieve the revision.

The copy command has other uses, along with other SVN commands for managing and merging branches of the repository. For this and other more advanced topics, consult SVN documentation and tutorials.

The First Time Through

The work you do in developing a CAE tool will be on a working copy you have checked out from the SVN repository. If you are checking out a working copy for the first time, you will issue the SVN checkout command[‡]. On Linux, this will be at the command line using **svn checkout** (Figure 5).

On Windows using TortoiseSVN, right-click on the Desktop and select **<SVN Checkout …>** from the pop-up menu. The Checkout window (Figure 6) will allow you to enter the URL for the trunk of the repository and to enter the name to give to the top-level directory of the working copy (if you want a different name than trunk). Click OK and the working copy will appear on your local machine with the name and location you specify (Figure 7).

[‡] The **checkout** command can be shortened to **co**.

```
$ svn co https://windsvn.nrel.gov/HydroDyn/svn/trunk  HydDyn
       <output omitted>
Checked out revision 1.
$ ls -ld HydDyn
drwxr-xr-x 6 jmichala jmichala 4096 Jan 16 15:29 HydDyn
$ ls -la HydDyn
total 80
drwxr-xr-x  6 jmichala jmichala  4096 Jan 16 15:29 .
drwxr-xr-x 77 jmichala jmichala  4096 Jan 16 15:29 ..
-rw-r--r--  1 jmichala jmichala   572 Jan 16 15:29 ArcFiles.txt
-rw-r--r--  1 jmichala jmichala  1161 Jan 16 15:29 Archive.bat
-rw-r--r--  1 jmichala jmichala  7384 Jan 16 15:29 ChangeLog.txt
-rw-r--r--  1 jmichala jmichala   117 Jan 16 15:29 Disclaimer.txt
-rw-r--r--  1 jmichala jmichala 36656 Jan 16 15:29 HydroDynOutListParameters.xlsx
drwxr-xr-x  4 jmichala jmichala  4096 Jan 16 15:29 Samples
drwxr-xr-x  3 jmichala jmichala  4096 Jan 16 15:29 Source
drwxr-xr-x  6 jmichala jmichala  4096 Jan 16 15:29 .svn
drwxr-xr-x  5 jmichala jmichala  4096 Jan 16 15:29 UtilityCodes
```

Figure 5. Example results from the SVN checkout command on Linux. Notice that the files and directories are no different from what you normally see except that SVN has included its own .svn directory as a hidden file.

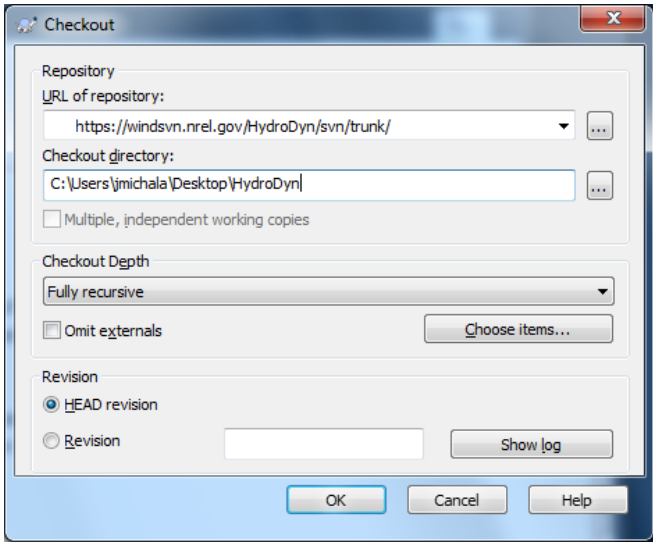


Figure 6. The TortoiseSVN checkout window.

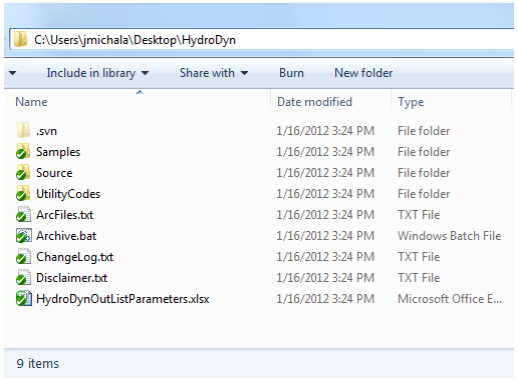


Figure 7. Example results on Windows® after checkout using TortoiseSVN.

In an Existing Working Copy

Until you are ready to start integrating your copy of the code with what is in the SVN repository, working with the SVN Working Copy of the code is no different from working with any local set of files and directories. The version control process comes into play once you have made changes and are ready to incorporate these back into the repository.

Get the Go Ahead.

The first step is determining that you are at this point. That is, are your changes ready to go into the repository? If the code worked before, does it still (however that's determined)? Have you

consulted with other developers and those responsible for the module you are working on and do they concur? These are more policy issues than SVN mechanics. (See Appendix B).

Assuming the answers are yes, the best practice steps for putting changes from a working copy back into the repository are update, resolve, update again, commit. The commit should never occur until an update comes back showing only your proposed changes without any conflicts. SVN is good at ensuring that commits are done atomically—that is, by only one user at a time. Even so, it is best if other developers working on the code in the repository know you are committing and wait for an all clear from you before making any commits of their own.

Status of the Working Copy.
Find out what it is you will be committing. At the beginning of a work cycle, before you have made any changes to code in the repository, there will not be anything to commit. The svn status command will return without generating any output, meaning that the working copy is up to date with respect to the repository. Here, the **svn status** command is issued in the source directory of a clean (unmodified) HydroDyn working copy:

```
$ svn status Source
$
```

On Windows with TortoiseSVN, the icon for the Source folder will have a green circle with a check mark, as shown in Figure 4. However, if changes have been made to files in the directory, the status command will report them:

```
[jmichala@rrlogin1 HydDyn]$ svn status Source
M      Source/Waves.f90
```

The 'M' to the left of the file name in the status listing means the file has been modified. Files might also be listed with a '?', which means that the file is unknown to SVN (it is not a copy of anything in the repository). Or it might be listed with a '!', meaning that there's a file stored in the repository that is missing in the working copy. On Windows, modifications are indicated with a red circle on the icon (see Figure 4).

To see what has changed in the code, you can use the **svn diff** command:

```
$ svn diff Source
Index: Source/Waves.f90
===================================================================
--- Source/Waves.f90    (revision 1)
+++ Source/Waves.f90    (working copy)
@@ -1,5 +1,6 @@
 MODULE Waves

+! a sample change, adding a line to MODULE Waves in Waves.f90

    ! This MODULE stores variables and routines associated with incident
    ! waves and current.
```

The output of the SVN diff command is similar to the output of the Unix diff utility. The added line in the change above is indicated with a '+'. A few lines before and after the change are included for context. On Windows, right-clicking and selecting **<Diff>** from the pull-down menu displays the differences graphically in a TortoiseMerge window (Figure 8). Note that

TortoiseMerge does not difference whole directories, so it was necessary to change into the source folder and right-click the Waves.f90 file that changed (also indicated with a red circle in its icon).

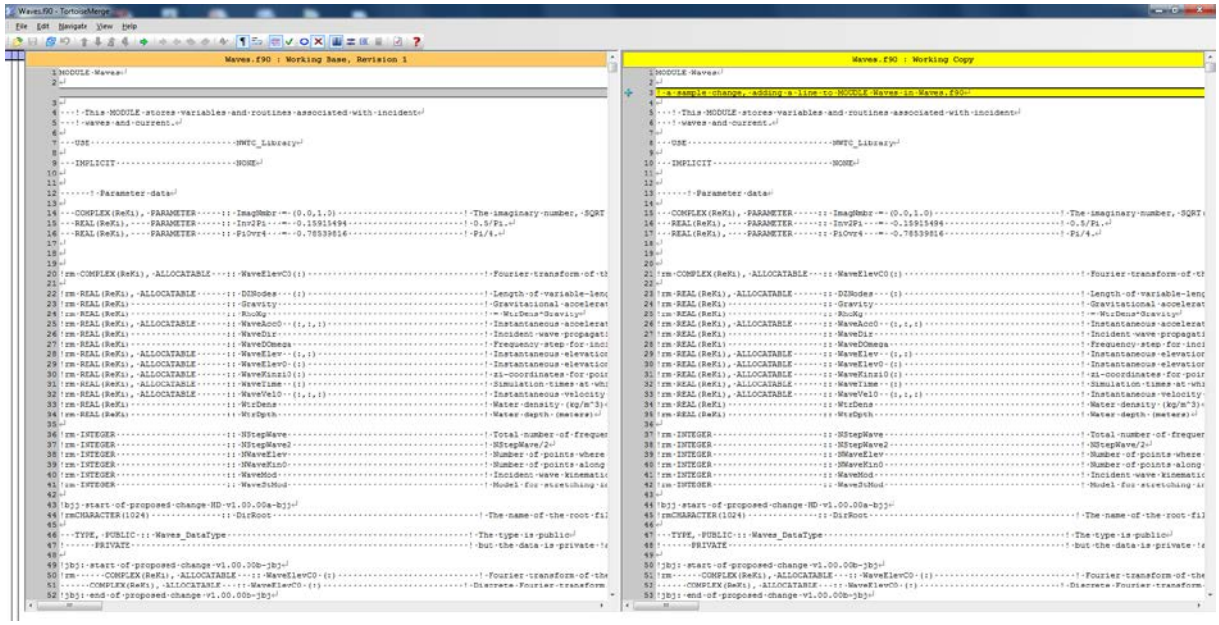


Figure 8. TortoiseSVN > Diff… **opens TortoiseMerge to compare and merge changes in text files.**

What is New in the Repository?

While you were working in your working copy, other developers were busy in their working copies. If they have committed their changes to the repository before you, the repository will not be the same as when you started with your clean working copy. The possibility of this situation is one of the main reasons you are using a version control system in the first place. The SVN update command brings your working copy up to date and checks to make sure that none of your changes will conflict with a change another developer has made in the meantime.

Gotchas

Avoid moving SVN working copies between Windows and Linux development platforms. End of line markers in text files of a working copy checked out from Subversion on a Windows system will differ from those in files checked out on Linux system. In addition to creating formatting problems when editing, this will appear as many spurious modifications to the working copy when updating, differencing, or committing with SVN from the other system. To move code to a different platform, check in your changes (to a branch, if necessary) and then check out again on the other system.

After importing, the copy of the code you just imported into SVN is *not* a working copy. Working copies are only ever created by the **svn checkout** command and will contain a **.svn** bookkeeping subdirectory in each of the working copy subdirectories (Linux) or a green or red dot on their icon (TortoiseSVN on Windows).

Working with Branches

SVN allows users to make copies of the development trunk called *branches* and separately maintain those branches under revision control until the work is ready to be merged back onto the trunk. This allows developers to work with and commit to their own SVN working copy of the trunk as they wish without interference or encumbrance from the policy for the main trunk (see Appendix B). The policy would apply when the branch copy is merged back to the main trunk. Under the Linux SVN client, creating a branch is done with the **svn copy** command. Under TortoiseSVN, there are separate Branch/Tag and Merge options in the menu that appears when right-clicking a file or directory in an SVN working copy. A working copy's association with the trunk version or a branch copy on the server is controlled using the **svn switch** command (same for both Linux and TortoiseSVN clients).

Additional Information

SVN tutorial information and links to additional documentation may be found on http://wind-dev.nrel.gov (internal) or http://www.michalakes.us/SVN_presentation.pdf (external). The "For Further Reading" section of this document also lists some resources.

Appendix B: NWTC CAE Tool Development Policy

The following policy outlines what is expected of all NWTC CAE tool developers.

- CAE tools under development are stored on NREL servers that can be accessed using Subversion. The repository for each tool has three directories: trunk, branches, and tags.
- The trunk must always be stable. No commit or merge shall break its functionality.
- Each tool has a primary owner who is responsible for the trunk version. The trunk may not be modified without the consent of the primary code owner.
- Development must be done in branches (each developer may make a branch copy of the most recent trunk version for making their changes).
- Branches must pass all certification tests prior to being merged back into the trunk. The developer must create additional tests if existing tests do not adequately evaluate features being developed.
- The tool's primary owner is responsible for reviewing changes from the branches and merging approved branches back into the trunk. The primary owner may delegate this responsibility if desired.
- After the trunk is updated (branches merged into the trunk), all certification tests must be rerun. This step is to check that changes made in the review/approval process and any conflicts in the merge process were resolved appropriately.
- Certification tests must be updated to test new features and to test issues that have been found while debugging code.
- Each time a tool is released—either as an alpha or beta version—a copy of the trunk is tagged using the NWTC version number. This provides a snapshot of the tool in the form it was released.

Appendix C: Recommended Practices for Code Development Using Subversion

Keep the following recommended practices in mind while you are developing code:

- Check your code back into Subversion branches on a regular basis (daily or multiple times a week when doing full time development). This way you can take advantage of Subversion for rolling back to previous versions of the code when something you just added doesn't work as expected.
- Developers should partition major code development efforts into reasonably sized projects that can be checked into the trunk separately. We do not want several years' worth of code to be merged back into the trunk, because the trunk may have changed dramatically during that time.
- Code should be merged into the trunk whenever problems have been fixed or new features have been added *and* the code is stable.
- Keep regular backups of your code. While our subversion repositories are backed up daily, you should not count on that alone. In the unlikely event that something happened to the repositories, it could be days before they are back online.
- After a branch is merged back into the trunk, it should be deleted. This will remove clutter from the repository and will allow other developers to see what branches are currently being developed. If you need to make more changes after the branch has been merged into the trunk, start a new branch.
- Regularly communicate with other developers. Usually, the earlier you spot coding conflicts in the development process, the easier it is to resolve them. Be considerate of other developers' programming goals.

Appendix D: Steps for Maintaining and Developing Software

Follow the steps below for maintaining and developing software:

1. **Plan**: Create a plan of the changes and/or additions you intend to make. Write this information in a document (you can use this when writing the theory manual in step 10). *Discuss the plan with the CAE tool's primary owner* and other developers that may be affected. See the "Planning" section of this document for details.
2. **Create a branch in the Subversion repository**: All development work must be done in branches to ensure that the trunk is always stable and that the primary owner has control of the state of the trunk.
3. **Change the version number**: The first thing you should change is the version number and date.
4. **Make changes**: Make sure you comment your changes well, both in the source code and in the change log. See the "Documentation" section of this document for details.
5. **Debug**: Create an executable program in debug mode, making the compiler option to check array and string bounds is enabled (use **/check:bounds**).
6. **Test**: Run CertTest.bat and other tests to verify that your changes have done only what they were supposed to do. Create new test cases to check new features. Return to step 4 and fix the source code as necessary.
7. **Compile the source code for release**: Compile the code with optimizations and without all the debugging options (in Intel Visual Fortran, you can use Release mode), but keep the **/traceback** option so that users have some information on the error if your program crashes ungracefully.
8. **Test again**: Run CertTest.bat and other tests to verify that the new compiled code did not cause any differences. Return to step 4 if it did.
9. **Update the test results**: When the optimized executable works satisfactorily, update the test results. You can do this by running the Update.bat script included with NWTC software. (Results from your new version will overwrite the results from the old version.)
10. **Update the user's guide and/or theory manual**: This step may be completed at a different stage in the process, but it *must* be completed before the code is released as a beta version.
11. **Commit changes to the Subversion branch**: We recommend that you update and commit your changes to the SVN branch on a regular basis. However, it must be checked back into the branch before step 12.
12. **Notify the CAE tool's primary owner**: Before accepting your changes, the owner may request that you modify your branch to comply with NWTC Programming Policies or to avoid conflicts with other development.
    A. Once accepted, the tool owner (or his/her designee) will merge the branch back into the trunk.

B. After the merge, the tool owner will rerun the certification tests to ensure it was completed successfully. If successful, the test results on the trunk will be updated with new results.

C. Your branch may be deleted after it is merged back into the trunk.

13. **Create an archive from the SVN trunk**: See the "Software Distribution" section in this document for more details. NWTC CAE tools are distributed with a file called "Archive.bat" that can create an archive from a list of files.
14. **Distribute the archive**: Send your archive to desired recipients. Ask for their feedback and return to step 1 with any modifications in a new alpha version.
15. **Update the web site**: The primary owner of each tool is responsible for updating the NWTC web site with new versions of the code. The timing (and content) of this step is up to the tool's primary owner.
16. **Announce the release**: The primary owner of each tool (or his/her designee) should send an email to people who have downloaded previous version of the tool (or similar NWTC software, if this is a new tool)[*] and should post an announcement at http://wind.nrel.gov/forum/wind/viewtopic.php?f=4&t=652.[†]

[*] Please see Bonnie Jonkman for a list of email addresses of people who have downloaded NWTC CAE tools (note that it may take several days to respond to your request). This list is unavailable to outside contributors.

[†] This topic is locked to prevent unauthorized posting, so you will have to see an account administrator to unlock it before posting your announcement.

Appendix E: Module Template

```
!**********************************************************************************************************************************
! The ModuleName and ModuleName_Types modules make up a template for creating user-defined calculations in the FAST Modularization
! Framework. ModuleNames_Types will be auto-generated based on a description of the variables for the module.
!
! "ModuleName" should be replaced with the name of your module. Example: HydroDyn
! "ModName" (in ModName_*) should be replaced with the module name or an abbreviation of it. Example: HD
!..................................................................................................................................
! LICENSING
! Copyright (C) 2012, 2013  National Renewable Energy Laboratory
!
!    This file is part of ModuleName.
!
!    ModuleName is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as
!    published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
!
!    This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty
!    of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more details.
!
!    You should have received a copy of the GNU General Public License along with ModuleName.
!    If not, see <http://www.gnu.org/licenses/>.
!
!**********************************************************************************************************************************
MODULE ModuleName

   USE ModuleName_Types
   USE NWTC_Library

   IMPLICIT NONE

   PRIVATE

   TYPE(ProgDesc), PARAMETER            :: ModName_Ver = ProgDesc( 'ModuleName', 'v1.02.00', '22-March-2013' )


      ! ..... Public Subroutines ...................................................................................................

   PUBLIC :: ModName_Init                           ! Initialization routine
   PUBLIC :: ModName_End                            ! Ending routine (includes clean up)

   PUBLIC :: ModName_UpdateStates                   ! Loose coupling routine for solving for constraint states, integrating
                                                    !   continuous states, and updating discrete states
   PUBLIC :: ModName_CalcOutput                     ! Routine for computing outputs

   PUBLIC :: ModName_CalcConstrStateResidual        ! Tight coupling routine for returning the constraint state residual
   PUBLIC :: ModName_CalcContStateDeriv             ! Tight coupling routine for computing derivatives of continuous states
   PUBLIC :: ModName_UpdateDiscState                ! Tight coupling routine for updating discrete states
```

```
   PUBLIC :: ModName_JacobianPInput               ! Routine to compute the Jacobians of the output (Y), continuous- (X), discrete-
                                                  !   (Xd), and constraint-state (Z) functions all with respect to the inputs (u)
   PUBLIC :: ModName_JacobianPContState           ! Routine to compute the Jacobians of the output (Y), continuous- (X), discrete-
                                                  !   (Xd), and constraint-state (Z) functions all with respect to the continuous
                                                  !   states (x)
   PUBLIC :: ModName_JacobianPDiscState           ! Routine to compute the Jacobians of the output (Y), continuous- (X), discrete-
                                                  !   (Xd), and constraint-state (Z) functions all with respect to the discrete
                                                  !   states (xd)
   PUBLIC :: ModName_JacobianPConstrState         ! Routine to compute the Jacobians of the output (Y), continuous- (X), discrete-
                                                  !   (Xd), and constraint-state (Z) functions all with respect to the constraint
                                                  !   states (z)

CONTAINS
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_Init( InitInp, u, p, x, xd, z, OtherState, y, Interval, InitOut, ErrStat, ErrMsg )
! This routine is called at the start of the simulation to perform initialization steps.
! The parameters are set here and not changed during the simulation.
! The initial states and initial guess for the input are defined.
!..................................................................................................................................

      TYPE(ModName_InitInputType),       INTENT(IN   )  :: InitInp     ! Input data for initialization routine
      TYPE(ModName_InputType),           INTENT(  OUT)  :: u           ! An initial guess for the input; input mesh must be defined
      TYPE(ModName_ParameterType),       INTENT(  OUT)  :: p           ! Parameters
      TYPE(ModName_ContinuousStateType), INTENT(  OUT)  :: x           ! Initial continuous states
      TYPE(ModName_DiscreteStateType),   INTENT(  OUT)  :: xd          ! Initial discrete states
      TYPE(ModName_ConstraintStateType), INTENT(  OUT)  :: z           ! Initial guess of the constraint states
      TYPE(ModName_OtherStateType),      INTENT(  OUT)  :: OtherState  ! Initial other/optimization states
      TYPE(ModName_OutputType),          INTENT(  OUT)  :: y           ! Initial system outputs (outputs are not calculated;
                                                                       !   only the output mesh is initialized)
      REAL(DbKi),                        INTENT(INOUT)  :: Interval    ! Coupling interval in seconds: the rate that
                                                                       !   (1) ModName_UpdateStates() is called in loose coupling &
                                                                       !   (2) ModName_UpdateDiscState() is called in tight coupling.
                                                                       !   Input is the suggested time from the glue code;
                                                                       !   Output is the actual coupling interval that will be used
                                                                       !   by the glue code.
      TYPE(ModName_InitOutputType),      INTENT(  OUT)  :: InitOut     ! Output for initialization routine
      INTEGER(IntKi),                    INTENT(  OUT)  :: ErrStat     ! Error status of the operation
      CHARACTER(*),                      INTENT(  OUT)  :: ErrMsg      ! Error message if ErrStat /= ErrID_None

         ! local variables

      INTEGER(IntKi)                                    :: NumOuts


         ! Initialize variables

      ErrStat = ErrID_None
      ErrMsg  = ""
```

```
NumOuts = 2


   ! Initialize the NWTC Subroutine Library

CALL NWTC_Init( )

   ! Display the module information

CALL DispNVD( ModName_Ver )


   ! Define parameters here:

p%DT  = Interval


   ! Define initial system states here:

x%DummyContState           = 0
xd%DummyDiscState          = 0
z%DummyConstrState         = 0
OtherState%DummyOtherState = 0


   ! Define initial guess for the system inputs here:

u%DummyInput = 0


   ! Define system output initializations (set up mesh) here:
ALLOCATE( y%WriteOutput(NumOuts), STAT = ErrStat )
IF ( ErrStat/= 0 ) THEN
   ErrStat = ErrID_Fatal
   ErrMsg  = 'Error allocating output header and units arrays in ModName_Init'
   RETURN
END IF

y%DummyOutput = 0
y%WriteOutput = 0


   ! Define initialization-routine output here:
ALLOCATE( InitOut%WriteOutputHdr(NumOuts), InitOut%WriteOutputUnt(NumOuts), STAT = ErrStat )
IF ( ErrStat/= 0 ) THEN
   ErrStat = ErrID_Fatal
   ErrMsg  = 'Error allocating output header and units arrays in ModName_Init'
   RETURN
END IF
```

```
      InitOut%WriteOutputHdr = (/ 'Time   ', 'Column2' /)
      InitOut%WriteOutputUnt = (/ '(s)',     '(-)'     /)


         ! If you want to choose your own rate instead of using what the glue code suggests, tell the glue code the rate at which
         !   this module must be called here:

       !Interval = p%DT


END SUBROUTINE ModName_Init
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_End( u, p, x, xd, z, OtherState, y, ErrStat, ErrMsg )
! This routine is called at the end of the simulation.
!..................................................................................................................................

      TYPE(ModName_InputType),           INTENT(INOUT)  :: u           ! System inputs
      TYPE(ModName_ParameterType),       INTENT(INOUT)  :: p           ! Parameters
      TYPE(ModName_ContinuousStateType), INTENT(INOUT)  :: x           ! Continuous states
      TYPE(ModName_DiscreteStateType),   INTENT(INOUT)  :: xd          ! Discrete states
      TYPE(ModName_ConstraintStateType), INTENT(INOUT)  :: z           ! Constraint states
      TYPE(ModName_OtherStateType),      INTENT(INOUT)  :: OtherState  ! Other/optimization states
      TYPE(ModName_OutputType),          INTENT(INOUT)  :: y           ! System outputs
      INTEGER(IntKi),                    INTENT(  OUT)  :: ErrStat     ! Error status of the operation
      CHARACTER(*),                      INTENT(  OUT)  :: ErrMsg      ! Error message if ErrStat /= ErrID_None


         ! Initialize ErrStat

      ErrStat = ErrID_None
      ErrMsg  = ""


         ! Place any last minute operations or calculations here:


         ! Close files here:



         ! Destroy the input data:

      CALL ModName_DestroyInput( u, ErrStat, ErrMsg )


         ! Destroy the parameter data:
```

```
      CALL ModName_DestroyParam( p, ErrStat, ErrMsg )


         ! Destroy the state data:

      CALL ModName_DestroyContState(   x,           ErrStat, ErrMsg )
      CALL ModName_DestroyDiscState(   xd,          ErrStat, ErrMsg )
      CALL ModName_DestroyConstrState( z,           ErrStat, ErrMsg )
      CALL ModName_DestroyOtherState(  OtherState,  ErrStat, ErrMsg )


         ! Destroy the output data:

      CALL ModName_DestroyOutput( y, ErrStat, ErrMsg )



END SUBROUTINE ModName_End
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_UpdateStates( t, n, Inputs, InputTimes, p, x, xd, z, OtherState, ErrStat, ErrMsg )
! Loose coupling routine for solving constraint states, integrating continuous states, and updating discrete states.
! Continuous, constraint, and discrete states are updated to values at t + Interval.
!..................................................................................................................................

      REAL(DbKi),                         INTENT(IN   ) :: t              ! Current simulation time in seconds
      INTEGER(IntKi),                     INTENT(IN   ) :: n              ! Current step of the simulation: t = n*Interval
      TYPE(ModName_InputType),            INTENT(IN   ) :: Inputs(:)      ! Inputs at InputTimes
      REAL(DbKi),                         INTENT(IN   ) :: InputTimes(:)  ! Times in seconds associated with Inputs
      TYPE(ModName_ParameterType),        INTENT(IN   ) :: p              ! Parameters
      TYPE(ModName_ContinuousStateType),  INTENT(INOUT) :: x              ! Input: Continuous states at t;
                                                                          !   Output: Continuous states at t + Interval
      TYPE(ModName_DiscreteStateType),    INTENT(INOUT) :: xd             ! Input: Discrete states at t;
                                                                          !   Output: Discrete states at t + Interval
      TYPE(ModName_ConstraintStateType),  INTENT(INOUT) :: z              ! Input: Constraint states at t;
                                                                          !   Output: Constraint states at t + Interval
      TYPE(ModName_OtherStateType),       INTENT(INOUT) :: OtherState     ! Other/optimization states
      INTEGER(IntKi),                     INTENT(  OUT) :: ErrStat        ! Error status of the operation
      CHARACTER(*),                       INTENT(  OUT) :: ErrMsg         ! Error message if ErrStat /= ErrID_None

         ! Local variables

      TYPE(ModName_ContinuousStateType)               :: dxdt           ! Continuous state derivatives at t
      TYPE(ModName_DiscreteStateType)                 :: xd_t           ! Discrete states at t (copy)
      TYPE(ModName_ConstraintStateType)               :: z_Residual     ! Residual of the constraint state functions (Z)
      TYPE(ModName_InputType)                         :: u              ! Instantaneous inputs
      INTEGER(IntKi)                                  :: ErrStat2       ! Error status of the operation (secondary error)
      CHARACTER(LEN(ErrMsg))                          :: ErrMsg2        ! Error message if ErrStat2 /= ErrID_None
```

```
   ! Initialize variables

ErrStat   = ErrID_None           ! no error has occurred
ErrMsg    = ""


! This subroutine contains an example of how the states could be updated. Developers will
! want to adjust the logic as necessary for their own situations.


   ! Get the inputs at time t, based on the array of values sent by the glue code:

CALL ModName_Input_ExtrapInterp( Inputs, InputTimes, u, t, ErrStat, ErrMsg )
IF ( ErrStat >= AbortErrLev ) RETURN



   ! Get first time derivatives of continuous states (dxdt):

CALL ModName_CalcContStateDeriv( t, u, p, x, xd, z, OtherState, dxdt, ErrStat, ErrMsg )
IF ( ErrStat >= AbortErrLev ) THEN
   CALL ModName_DestroyContState( dxdt, ErrStat2, ErrMsg2)
   RETURN
END IF



   ! Update discrete states:
   !   Note that xd [discrete state] is changed in ModName_UpdateDiscState() so xd will now contain values at t+Interval
   !   We'll first make a copy that contains xd at time t, which will be used in computing the constraint states
CALL ModName_CopyDiscState( xd, xd_t, MESH_NEWCOPY, ErrStat, ErrMsg )

CALL ModName_UpdateDiscState( t, n, u, p, x, xd, z, OtherState, ErrStat, ErrMsg )
IF ( ErrStat >= AbortErrLev ) THEN
   CALL ModName_DestroyConstrState( Z_Residual, ErrStat2, ErrMsg2)
   CALL ModName_DestroyContState(   dxdt,       ErrStat2, ErrMsg2)
   CALL ModName_DestroyDiscState(   xd_t,       ErrStat2, ErrMsg2)
   RETURN
END IF


   ! Solve for the constraint states (z) here:

   ! Iterate until the value is within a given tolerance.

! DO

   CALL ModName_CalcConstrStateResidual( t, u, p, x, xd_t, z, OtherState, Z_Residual, ErrStat, ErrMsg )
```

```
         IF ( ErrStat >= AbortErrLev ) THEN
            CALL ModName_DestroyConstrState( Z_Residual, ErrStat2, ErrMsg2)
            CALL ModName_DestroyContState(   dxdt,       ErrStat2, ErrMsg2)
            CALL ModName_DestroyDiscState(   xd_t,       ErrStat2, ErrMsg2)
            RETURN
         END IF

         !  z =

      ! END DO



         ! Integrate (update) continuous states (x) here:

      !x = function of dxdt and x


         ! Destroy local variables before returning

      CALL ModName_DestroyConstrState( Z_Residual, ErrStat2, ErrMsg2)
      CALL ModName_DestroyContState(   dxdt,       ErrStat2, ErrMsg2)
      CALL ModName_DestroyDiscState(   xd_t,       ErrStat2, ErrMsg2)


END SUBROUTINE ModName_UpdateStates
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_CalcOutput( t, u, p, x, xd, z, OtherState, y, ErrStat, ErrMsg )
! Routine for computing outputs, used in both loose and tight coupling.
!..................................................................................................................................

      REAL(DbKi),                        INTENT(IN   )  :: t           ! Current simulation time in seconds
      TYPE(ModName_InputType),           INTENT(IN   )  :: u           ! Inputs at t
      TYPE(ModName_ParameterType),       INTENT(IN   )  :: p           ! Parameters
      TYPE(ModName_ContinuousStateType), INTENT(IN   )  :: x           ! Continuous states at t
      TYPE(ModName_DiscreteStateType),   INTENT(IN   )  :: xd          ! Discrete states at t
      TYPE(ModName_ConstraintStateType), INTENT(IN   )  :: z           ! Constraint states at t
      TYPE(ModName_OtherStateType),      INTENT(INOUT)  :: OtherState  ! Other/optimization states
      TYPE(ModName_OutputType),          INTENT(INOUT)  :: y           ! Outputs computed at t (Input only so that mesh con-
                                                                       !   nectivity information does not have to be recalculated)
      INTEGER(IntKi),                    INTENT(  OUT)  :: ErrStat     ! Error status of the operation
      CHARACTER(*),                      INTENT(  OUT)  :: ErrMsg      ! Error message if ErrStat /= ErrID_None


         ! Initialize ErrStat

      ErrStat = ErrID_None
      ErrMsg  = ""
```

```
      ! Compute outputs here:
   y%DummyOutput    = 2.0_ReKi

   y%WriteOutput(1) = REAL(t,ReKi)
   y%WriteOutput(2) = 1.0_ReKi


END SUBROUTINE ModName_CalcOutput
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_CalcContStateDeriv( t, u, p, x, xd, z, OtherState, dxdt, ErrStat, ErrMsg )
! Tight coupling routine for computing derivatives of continuous states
!..................................................................................................................................

   REAL(DbKi),                        INTENT(IN   )  :: t           ! Current simulation time in seconds
   TYPE(ModName_InputType),           INTENT(IN   )  :: u           ! Inputs at t
   TYPE(ModName_ParameterType),       INTENT(IN   )  :: p           ! Parameters
   TYPE(ModName_ContinuousStateType), INTENT(IN   )  :: x           ! Continuous states at t
   TYPE(ModName_DiscreteStateType),   INTENT(IN   )  :: xd          ! Discrete states at t
   TYPE(ModName_ConstraintStateType), INTENT(IN   )  :: z           ! Constraint states at t
   TYPE(ModName_OtherStateType),      INTENT(INOUT)  :: OtherState  ! Other/optimization states
   TYPE(ModName_ContinuousStateType), INTENT(  OUT)  :: dxdt        ! Continuous state derivatives at t
   INTEGER(IntKi),                    INTENT(  OUT)  :: ErrStat     ! Error status of the operation
   CHARACTER(*),                      INTENT(  OUT)  :: ErrMsg      ! Error message if ErrStat /= ErrID_None


      ! Initialize ErrStat

   ErrStat = ErrID_None
   ErrMsg  = ""


      ! Compute the first time derivatives of the continuous states here:

   dxdt%DummyContState = 0


END SUBROUTINE ModName_CalcContStateDeriv
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_UpdateDiscState( t, n, u, p, x, xd, z, OtherState, ErrStat, ErrMsg )
! Tight coupling routine for updating discrete states
!..................................................................................................................................

   REAL(DbKi),                        INTENT(IN   )  :: t           ! Current simulation time in seconds
   INTEGER(IntKi),                    INTENT(IN   )  :: n           ! Current step of the simulation: t = n*Interval
   TYPE(ModName_InputType),           INTENT(IN   )  :: u           ! Inputs at t
   TYPE(ModName_ParameterType),       INTENT(IN   )  :: p           ! Parameters
   TYPE(ModName_ContinuousStateType), INTENT(IN   )  :: x           ! Continuous states at t
   TYPE(ModName_DiscreteStateType),   INTENT(INOUT)  :: xd          ! Input: Discrete states at t;
```

```
                                                                  !   Output: Discrete states at t + Interval
      TYPE(ModName_ConstraintStateType), INTENT(IN   )  :: z           ! Constraint states at t
      TYPE(ModName_OtherStateType),      INTENT(INOUT)  :: OtherState  ! Other/optimization states
      INTEGER(IntKi),                    INTENT(  OUT)  :: ErrStat     ! Error status of the operation
      CHARACTER(*),                      INTENT(  OUT)  :: ErrMsg      ! Error message if ErrStat /= ErrID_None


         ! Initialize ErrStat

      ErrStat = ErrID_None
      ErrMsg  = ""


         ! Update discrete states here:

      xd%DummyDiscState = 0.0

END SUBROUTINE ModName_UpdateDiscState
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_CalcConstrStateResidual( t, u, p, x, xd, z, OtherState, Z_residual, ErrStat, ErrMsg )
! Tight coupling routine for solving for the residual of the constraint state functions
!..................................................................................................................................

      REAL(DbKi),                        INTENT(IN   )  :: t           ! Current simulation time in seconds
      TYPE(ModName_InputType),           INTENT(IN   )  :: u           ! Inputs at t
      TYPE(ModName_ParameterType),       INTENT(IN   )  :: p           ! Parameters
      TYPE(ModName_ContinuousStateType), INTENT(IN   )  :: x           ! Continuous states at t
      TYPE(ModName_DiscreteStateType),   INTENT(IN   )  :: xd          ! Discrete states at t
      TYPE(ModName_ConstraintStateType), INTENT(IN   )  :: z           ! Constraint states at t (possibly a guess)
      TYPE(ModName_OtherStateType),      INTENT(INOUT)  :: OtherState  ! Other/optimization states
      TYPE(ModName_ConstraintStateType), INTENT(  OUT)  :: Z_residual  ! Residual of the constraint state functions using
                                                                       !     the input values described above
      INTEGER(IntKi),                    INTENT(  OUT)  :: ErrStat     ! Error status of the operation
      CHARACTER(*),                      INTENT(  OUT)  :: ErrMsg      ! Error message if ErrStat /= ErrID_None


         ! Initialize ErrStat

      ErrStat = ErrID_None
      ErrMsg  = ""


         ! Solve for the residual of the constraint state functions here:

      Z_residual%DummyConstrState = 0

END SUBROUTINE ModName_CalcConstrStateResidual
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! WE ARE NOT YET IMPLEMENTING THE JACOBIANS...
```

```
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_JacobianPInput( t, u, p, x, xd, z, OtherState, dYdu, dXdu, dXddu, dZdu, ErrStat, ErrMsg )
! Routine to compute the Jacobians of the output (Y), continuous- (X), discrete- (Xd), and constraint-state (Z) functions
! with respect to the inputs (u). The partial derivatives dY/du, dX/du, dXd/du, and DZ/du are returned.
!..................................................................................................................................

      REAL(DbKi),                                INTENT(IN   )           :: t          ! Current simulation time in seconds
      TYPE(ModName_InputType),                   INTENT(IN   )           :: u          ! Inputs at t
      TYPE(ModName_ParameterType),               INTENT(IN   )           :: p          ! Parameters
      TYPE(ModName_ContinuousStateType),         INTENT(IN   )           :: x          ! Continuous states at t
      TYPE(ModName_DiscreteStateType),           INTENT(IN   )           :: xd         ! Discrete states at t
      TYPE(ModName_ConstraintStateType),         INTENT(IN   )           :: z          ! Constraint states at t
      TYPE(ModName_OtherStateType),              INTENT(INOUT)           :: OtherState ! Other/optimization states
      TYPE(ModName_PartialOutputPInputType),     INTENT(  OUT), OPTIONAL :: dYdu       ! Partial derivatives of output functions
                                                                                       !   (Y) with respect to the inputs (u)
      TYPE(ModName_PartialContStatePInputType),  INTENT(  OUT), OPTIONAL :: dXdu       ! Partial derivatives of continuous state
                                                                                       !   functions (X) with respect to inputs (u)
      TYPE(ModName_PartialDiscStatePInputType),  INTENT(  OUT), OPTIONAL :: dXddu      ! Partial derivatives of discrete state
                                                                                       !   functions (Xd) with respect to inputs (u)
      TYPE(ModName_PartialConstrStatePInputType),INTENT(  OUT), OPTIONAL :: dZdu       ! Partial derivatives of constraint state
                                                                                       !   functions (Z) with respect to inputs (u)
      INTEGER(IntKi),                            INTENT(  OUT)           :: ErrStat    ! Error status of the operation
      CHARACTER(*),                              INTENT(  OUT)           :: ErrMsg     ! Error message if ErrStat /= ErrID_None


         ! Initialize ErrStat

      ErrStat = ErrID_None
      ErrMsg  = ""


      IF ( PRESENT( dYdu ) ) THEN

         ! Calculate the partial derivative of the output functions (Y) with respect to the inputs (u) here:

         dYdu%DummyOutput%DummyInput = 0

      END IF

      IF ( PRESENT( dXdu ) ) THEN

         ! Calculate the partial derivative of the continuous state functions (X) with respect to the inputs (u) here:

         dXdu%DummyContState%DummyInput = 0

      END IF

      IF ( PRESENT( dXddu ) ) THEN
```

```
         ! Calculate the partial derivative of the discrete state functions (Xd) with respect to the inputs (u) here:

         dXddu%DummyDiscState%DummyInput = 0

      END IF

      IF ( PRESENT( dZdu ) ) THEN

         ! Calculate the partial derivative of the constraint state functions (Z) with respect to the inputs (u) here:

         dZdu%DummyConstrState%DummyInput = 0

      END IF


END SUBROUTINE ModName_JacobianPInput
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_JacobianPContState( t, u, p, x, xd, z, OtherState, dYdx, dXdx, dXddx, dZdx, ErrStat, ErrMsg )
! Routine to compute the Jacobians of the output (Y), continuous- (X), discrete- (Xd), and constraint-state (Z) functions
! with respect to the continuous states (x). The partial derivatives dY/dx, dX/dx, dXd/dx, and DZ/dx are returned.
!..................................................................................................................................

      REAL(DbKi),                                  INTENT(IN   )           :: t          ! Current simulation time in seconds
      TYPE(ModName_InputType),                     INTENT(IN   )           :: u          ! Inputs at t
      TYPE(ModName_ParameterType),                 INTENT(IN   )           :: p          ! Parameters
      TYPE(ModName_ContinuousStateType),           INTENT(IN   )           :: x          ! Continuous states at t
      TYPE(ModName_DiscreteStateType),             INTENT(IN   )           :: xd         ! Discrete states at t
      TYPE(ModName_ConstraintStateType),           INTENT(IN   )           :: z          ! Constraint states at t
      TYPE(ModName_OtherStateType),                INTENT(INOUT)           :: OtherState ! Other/optimization states
      TYPE(ModName_PartialOutputPContStateType),   INTENT(  OUT), OPTIONAL :: dYdx       ! Partial derivatives of output functions
                                                                                         !   (Y) with respect to the continuous
                                                                                         !   states (x)
      TYPE(ModName_PartialContStatePContStateType), INTENT(  OUT), OPTIONAL :: dXdx      ! Partial derivatives of continuous state
                                                                                         !   functions (X) with respect to
                                                                                         !   the continuous states (x)
      TYPE(ModName_PartialDiscStatePContStateType), INTENT(  OUT), OPTIONAL :: dXddx     ! Partial derivatives of discrete state
                                                                                         !   functions (Xd) with respect to
                                                                                         !   the continuous states (x)
      TYPE(ModName_PartialConstrStatePContStateType),INTENT(  OUT), OPTIONAL :: dZdx     ! Partial derivatives of constraint state
                                                                                         !   functions (Z) with respect to
                                                                                         !   the continuous states (x)
      INTEGER(IntKi),                              INTENT(  OUT)           :: ErrStat    ! Error status of the operation
      CHARACTER(*),                                INTENT(  OUT)           :: ErrMsg     ! Error message if ErrStat /= ErrID_None


         ! Initialize ErrStat

      ErrStat = ErrID_None
```

```
      ErrMsg  = ""


      IF ( PRESENT( dYdx ) ) THEN
         ! Calculate the partial derivative of the output functions (Y) with respect to the continuous states (x) here:
         dYdx%DummyOutput%DummyContState = 0
      END IF
      IF ( PRESENT( dXdx ) ) THEN
         ! Calculate the partial derivative of the continuous state functions (X) with respect to the continuous states (x) here:
         dXdx%DummyContState%DummyContState = 0
      END IF
      IF ( PRESENT( dXddx ) ) THEN
         ! Calculate the partial derivative of the discrete state functions (Xd) with respect to the continuous states (x) here:
         dXddx%DummyDiscState%DummyContState = 0
      END IF
      IF ( PRESENT( dZdx ) ) THEN

         ! Calculate the partial derivative of the constraint state functions (Z) with respect to the continuous states (x) here:
         dZdx%DummyConstrState%DummyContState = 0
      END IF

   END SUBROUTINE ModName_JacobianPContState
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_JacobianPDiscState( t, u, p, x, xd, z, OtherState, dYdxd, dXdxd, dXddxd, dZdxd, ErrStat, ErrMsg )
! Routine to compute the Jacobians of the output (Y), continuous- (X), discrete- (Xd), and constraint-state (Z) functions
! with respect to the discrete states (xd). The partial derivatives dY/dxd, dX/dxd, dXd/dxd, and DZ/dxd are returned.
!..................................................................................................................................

      REAL(DbKi),                                INTENT(IN   )          :: t          ! Current simulation time in seconds
      TYPE(ModName_InputType),                   INTENT(IN   )          :: u          ! Inputs at t
      TYPE(ModName_ParameterType),               INTENT(IN   )          :: p          ! Parameters
      TYPE(ModName_ContinuousStateType),         INTENT(IN   )          :: x          ! Continuous states at t
```

```
TYPE(ModName_DiscreteStateType),              INTENT(IN   )           :: xd         ! Discrete states at t
TYPE(ModName_ConstraintStateType),            INTENT(IN   )           :: z          ! Constraint states at t
TYPE(ModName_OtherStateType),                 INTENT(INOUT)           :: OtherState ! Other/optimization states
TYPE(ModName_PartialOutputPDiscStateType),    INTENT(  OUT), OPTIONAL :: dYdxd      ! Partial derivatives of output functions
                                                                                    !  (Y) with respect to the discrete
                                                                                    !  states (xd)
TYPE(ModName_PartialContStatePDiscStateType), INTENT(  OUT), OPTIONAL :: dXdxd      ! Partial derivatives of continuous state
                                                                                    !   functions (X) with respect to the
                                                                                    !   discrete states (xd)
TYPE(ModName_PartialDiscStatePDiscStateType), INTENT(  OUT), OPTIONAL :: dXddxd     ! Partial derivatives of discrete state
                                                                                    !   functions (Xd) with respect to the
                                                                                    !   discrete states (xd)
TYPE(ModName_PartialConstrStatePDiscStateType),INTENT(  OUT), OPTIONAL :: dZdxd     ! Partial derivatives of constraint state
                                                                                    !   functions (Z) with respect to the
                                                                                    !   discrete states (xd)
INTEGER(IntKi),                               INTENT(  OUT)           :: ErrStat    ! Error status of the operation
CHARACTER(*),                                 INTENT(  OUT)           :: ErrMsg     ! Error message if ErrStat /= ErrID_None


   ! Initialize ErrStat

ErrStat = ErrID_None
ErrMsg  = ""


IF ( PRESENT( dYdxd ) ) THEN

   ! Calculate the partial derivative of the output functions (Y) with respect to the discrete states (xd) here:

   dYdxd%DummyOutput%DummyDiscState = 0

END IF

IF ( PRESENT( dXdxd ) ) THEN

   ! Calculate the partial derivative of the continuous state functions (X) with respect to the discrete states (xd) here:

   dXdxd%DummyContState%DummyDiscState = 0

END IF

IF ( PRESENT( dXddxd ) ) THEN

   ! Calculate the partial derivative of the discrete state functions (Xd) with respect to the discrete states (xd) here:

   dXddxd%DummyDiscState%DummyDiscState = 0

END IF
```

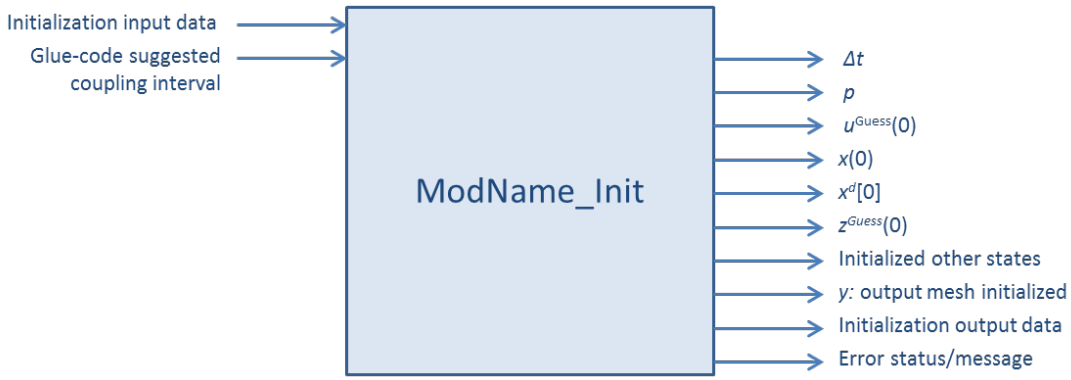
```
      IF ( PRESENT( dZdxd ) ) THEN

         ! Calculate the partial derivative of the constraint state functions (Z) with respect to the discrete states (xd) here:

         dZdxd%DummyConstrState%DummyDiscState = 0

      END IF


END SUBROUTINE ModName_JacobianPDiscState
!----------------------------------------------------------------------------------------------------------------------------------
SUBROUTINE ModName_JacobianPConstrState( t, u, p, x, xd, z, OtherState, dYdz, dXdz, dXddz, dZdz, ErrStat, ErrMsg )
! Routine to compute the Jacobians of the output (Y), continuous- (X), discrete- (Xd), and constraint-state (Z) functions
! with respect to the constraint states (z). The partial derivatives dY/dz, dX/dz, dXd/dz, and DZ/dz are returned.
!..................................................................................................................................

      REAL(DbKi),                                    INTENT(IN   )           :: t          ! Current simulation time in seconds
      TYPE(ModName_InputType),                       INTENT(IN   )           :: u          ! Inputs at t
      TYPE(ModName_ParameterType),                   INTENT(IN   )           :: p          ! Parameters
      TYPE(ModName_ContinuousStateType),             INTENT(IN   )           :: x          ! Continuous states at t
      TYPE(ModName_DiscreteStateType),               INTENT(IN   )           :: xd         ! Discrete states at t
      TYPE(ModName_ConstraintStateType),             INTENT(IN   )           :: z          ! Constraint states at t
      TYPE(ModName_OtherStateType),                  INTENT(INOUT)           :: OtherState ! Other/optimization states
      TYPE(ModName_PartialOutputPConstrStateType),   INTENT(  OUT), OPTIONAL :: dYdz       ! Partial derivatives of output
                                                                                           !  functions (Y) with respect to the
                                                                                           !  constraint states (z)
      TYPE(ModName_PartialContStatePConstrStateType), INTENT(  OUT), OPTIONAL :: dXdz       ! Partial derivatives of continuous
                                                                                           !  state functions (X) with respect to
                                                                                           !  the constraint states (z)
      TYPE(ModName_PartialDiscStatePConstrStateType), INTENT(  OUT), OPTIONAL :: dXddz      ! Partial derivatives of discrete state
                                                                                           !  functions (Xd) with respect to the
                                                                                           !  constraint states (z)
      TYPE(ModName_PartialConstrStatePConstrStateType),INTENT(  OUT), OPTIONAL :: dZdz       ! Partial derivatives of constraint
                                                                                           ! state functions (Z) with respect to
                                                                                           !  the constraint states (z)
      INTEGER(IntKi),                                INTENT(  OUT)           :: ErrStat    ! Error status of the operation
      CHARACTER(*),                                  INTENT(  OUT)           :: ErrMsg     ! Error message if ErrStat /= ErrID_None


         ! Initialize ErrStat

      ErrStat = ErrID_None
      ErrMsg  = ""

      IF ( PRESENT( dYdz ) ) THEN

            ! Calculate the partial derivative of the output functions (Y) with respect to the constraint states (z) here:

         dYdz%DummyOutput%DummyConstrState = 0
```

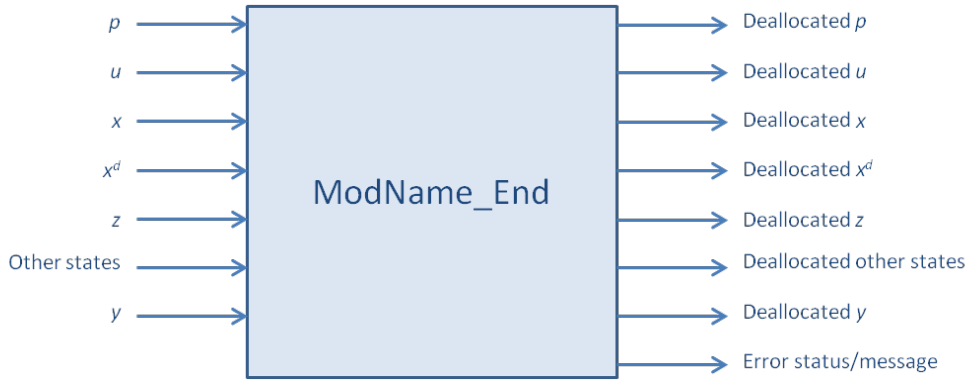
```
      END IF

      IF ( PRESENT( dXdz ) ) THEN

            ! Calculate the partial derivative of the continuous state functions (X) with respect to the constraint states (z) here:

         dXdz%DummyContState%DummyConstrState = 0

      END IF

      IF ( PRESENT( dXddz ) ) THEN

            ! Calculate the partial derivative of the discrete state functions (Xd) with respect to the constraint states (z) here:

         dXddz%DummyDiscState%DummyConstrState = 0

      END IF

      IF ( PRESENT( dZdz ) ) THEN

            ! Calculate the partial derivative of the constraint state functions (Z) with respect to the constraint states (z) here:

         dZdz%DummyConstrState%DummyConstrState = 0

      END IF


END SUBROUTINE ModName_JacobianPConstrState
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

END MODULE ModuleName
!**********************************************************************************************************************************
```

Appendix F: Subroutine Inputs and Outputs for Modules Developed for the FAST CAE Tool Framework

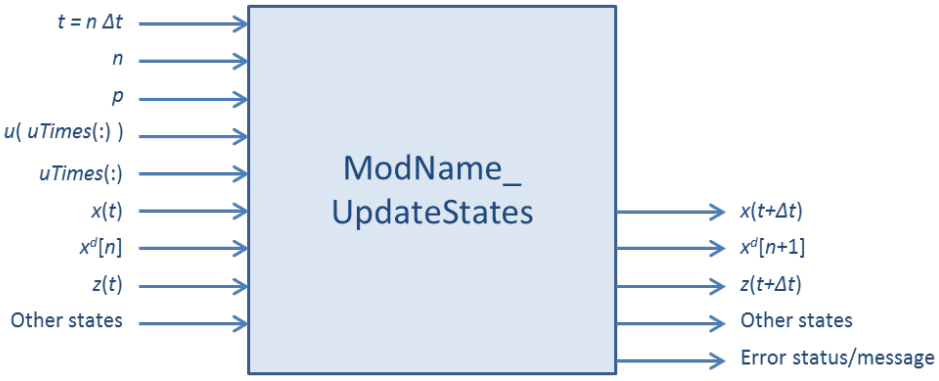
The following boxes with arrows indicate the inputs and outputs required for the subroutines developers will have to provide in the FAST modular framework. Arrows going into the box are INTENT(IN); arrows going out of a box are INTENT(OUT); when an output arrow lines up horizontally with an input arrow, it indicates a single argument that is INTENT(INOUT). The Fortran template for these subroutines is provided in Appendix E. Subroutines not pictured here will be generated automatically using the FAST Registry described in Appendix H.

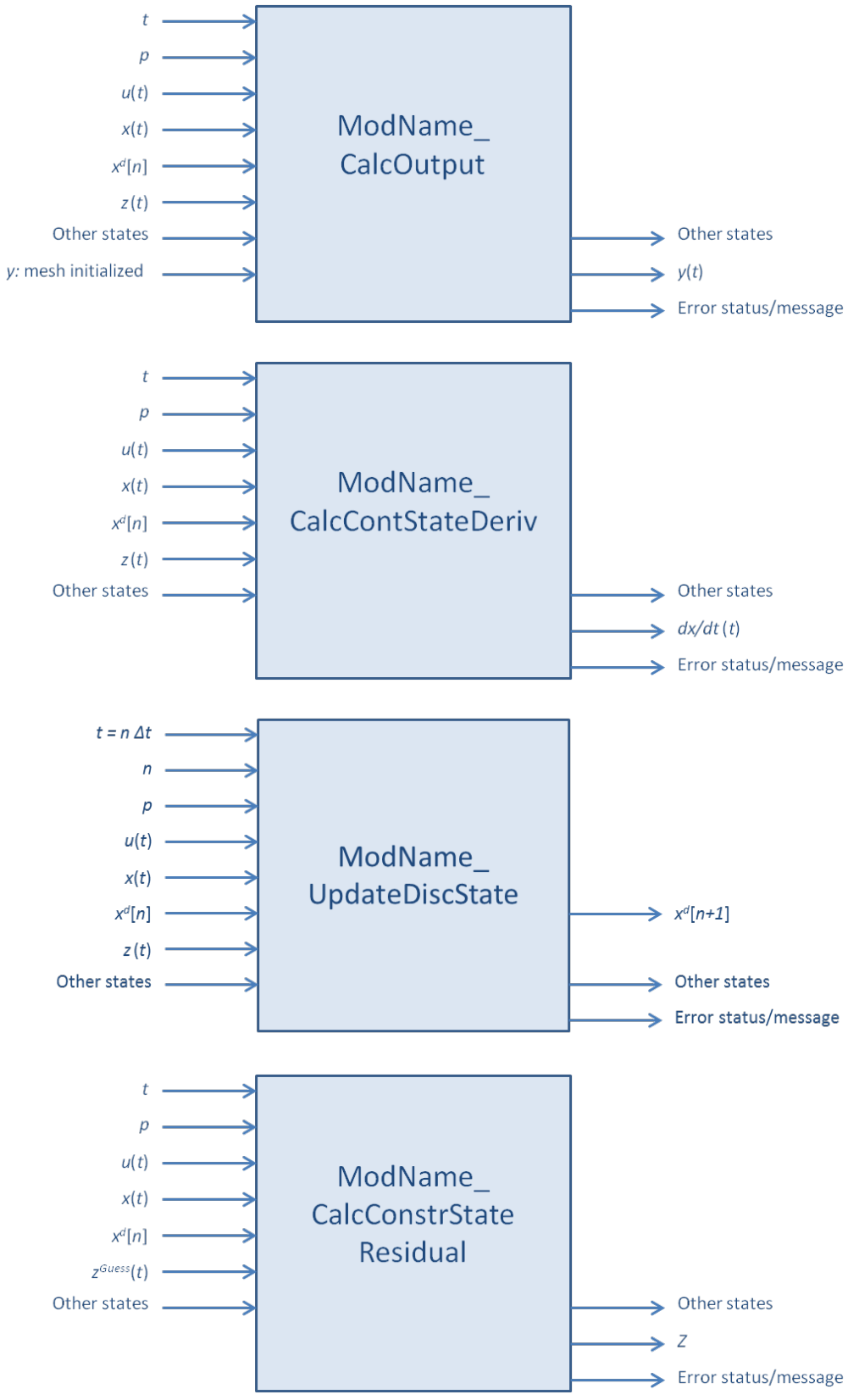
Initialization/End Subroutines:



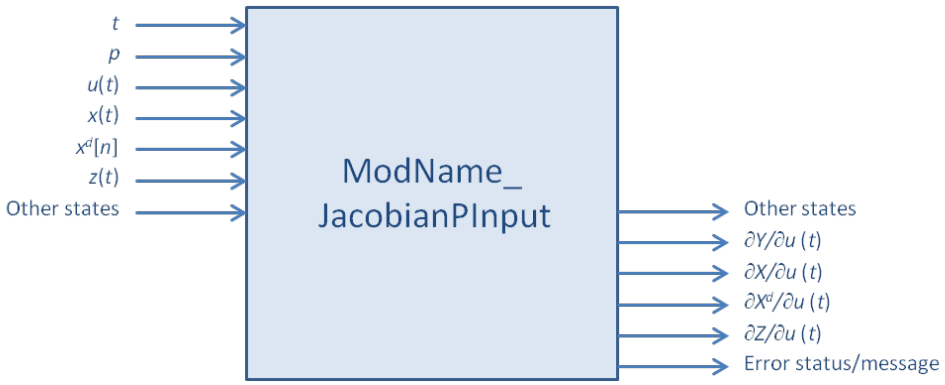


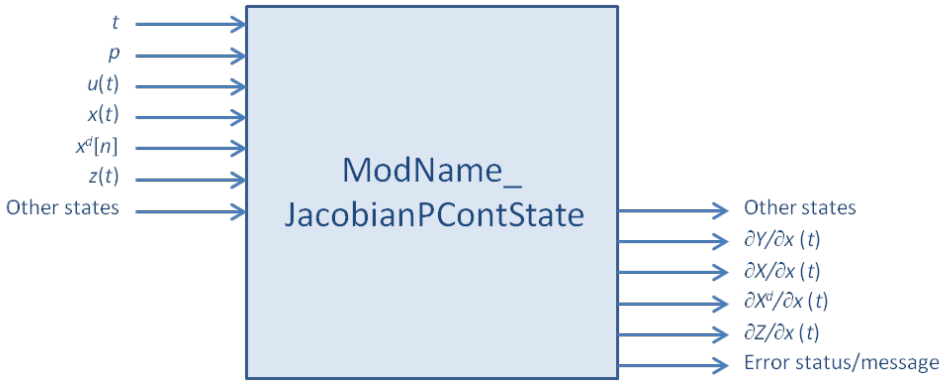
Time-Stepping Subroutines:

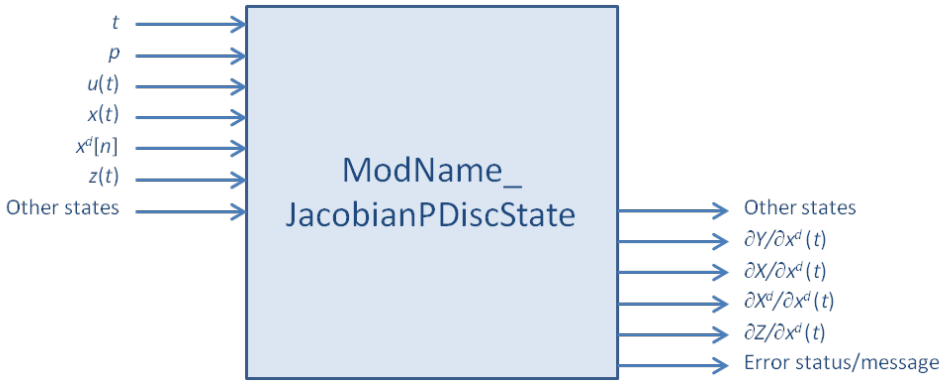


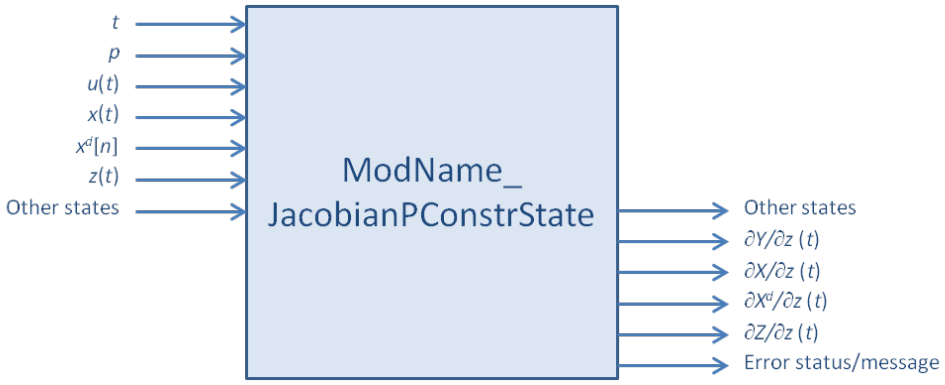

t
p
u(t)
x(t)
x^d[n]
z(t)
Other states
y: mesh initialized
ModName_
CalcOutput
Other states
y(t)
Error status/message
t
p
u(t)
x(t)
x^d[n]
z(t)
Other states
ModName_
CalcContStateDeriv
Other states
dx/dt (t)
Error status/message
t = n Δt
n
p
u(t)
x(t)
x^d[n]
z(t)
Other states
ModName_
UpdateDiscState
x^d[n+1]
Other states
Error status/message
t
p
u(t)
x(t)
x^d[n]
z^Guess(t)
Other states
ModName_
CalcConstrState
Residual
Other states
Z
Error status/message

Jacobian Subroutines:









Appendix G: Mesh Module and Types

The modules ModMesh and ModMesh_Types provide data structures and subroutines for representing and manipulating meshes and meshed data in the FAST modular framework. Inputs, outputs, parameters, and states can all be stored as meshed data; however, input and output data on discretized spatial boundaries *must* be stored using the ModMesh and ModMesh_Types modules. A mesh is comprised of a set of "nodes" (simple points in space) together with information specifying how they are connected to form "elements" (Figure 9) representing spatial boundaries between components. ModMesh and ModMesh_Types define point, line, surface, and volume elements in a standard isoparametric mapping from finite element analysis. Each element (except points) has optional midside nodes that allow quadradically defined curved lines or surfaces. Otherwise, lines are straight and surfaces are flat. The ModMesh and ModMesh_Types modules may be USE associated explicitly or as part of NWTC_Library.

Associated with a mesh are one or more "fields" that represent the values of variables or "degrees of freedom" at each node. A mesh always has a named "Position" that specifies the location in three-dimensional space as an X_i,Y_i,Z_i triplet of each node and a field named "RefOrientation" that specifies the orientation (as a direction cosine matrix) of the node. The ModMesh_Types module predefines a number of other fields of triples representing velocities, forces, and moments as well as a field of nine values representing a direction cosine matrix.

The operations on meshes defined in the ModMesh module are creation, spatio-location of nodes, construction, committing the mesh definition, initialization of fields, accessing field data, updating field data, copying, deallocating, and destroying meshes. Higher level operations on meshes—such as interpolation, remapping, subsetting, and joining—are being developed in another module.

Creating a mesh entails defining an instance of a mesh structure of TYPE(MeshType) to store a

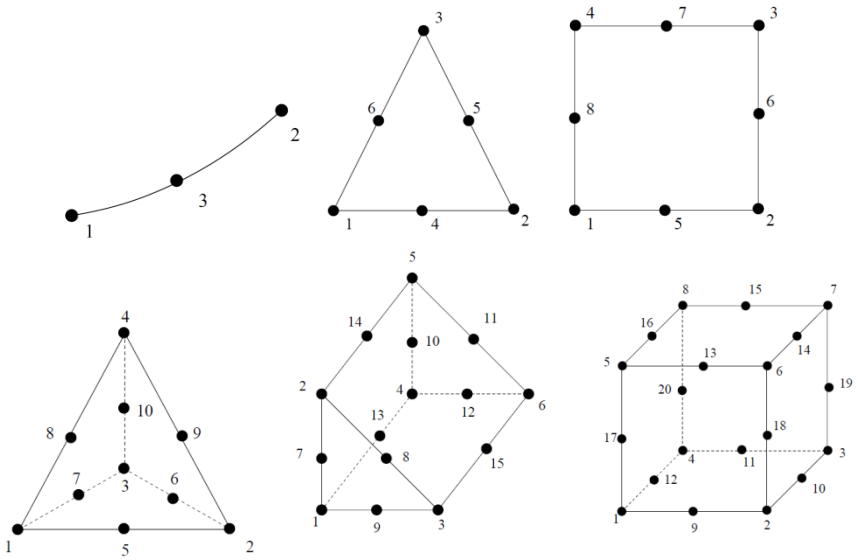


Figure 9. Line, surface, and volume elements defined and managed by ModMesh_Types and ModMesh modules: lines, triangles, quadrilaterals, wedges, and tetrahedra.

mesh of a given size (number of nodes). *Spatio-location* involves providing the spatial location (and orientation) information of each node of the mesh within the FAST global coordinate system. *Constructing* a mesh involves connecting the nodes together into elements and, in turn, connecting the elements. Once a mesh has been created, constructed, and spatio-located it must be *committed*, which tells the mesh library that the mesh has been completely defined and ready to be initialized. *Initializing* the mesh involves setting the variables at each node in the mesh to their initial values, if any.

Requiring that a mesh be explicitly committed before it can be used allows the library to precompute traversal information, and other information about the mesh to construct neighbor lists to facilitate efficient access. It is an error to initialize meshed data or use a mesh until it is committed. Likewise, and with the exception of changing spatio-location of nodes, it is an error to redefine the mesh after it has been committed.

The ModMesh routines for creation, spatio-location, construction, committing and initialization operations on a mesh are intended to be called from the **ModName_Init** routine provided in the FAST modularization framework. The dirver program calls the component's **ModName_Init** routine, which creates, constructs, and initializes and then passes back the ready-to-use meshes. Once a mesh is constructed and initialized, it may be used. The access and update operations on a mesh are intended for use by the routines provided by the developer of a FAST module.

The remainder of this appendix describes the MeshType data type and the subroutines that operate on the type.

Type: MeshType

The MeshType stores information about the nodes, elements, and fields of a mesh. The members of the MeshType derived data type shown below are all public and may be accessed as needed by the developer. However, only certain members may be modified once the mesh is committed: those that store values of fields on the mesh (indicated in comments below).* The RemapFlag may also be altered directly by component code to, for example, indicate to the Glue code that positions of nodes of the mesh have changed. The public definition of MeshType is:

```
TYPE, PUBLIC :: MeshType
  LOGICAL :: initialized                          ! Indicate whether this mesh is initialized
  LOGICAL :: fieldmask(FIELDMASK_SIZE)            ! Number of allocatable fields, below
  LOGICAL, POINTER :: RemapFlag                   ! false=no action/ignore; true=remap required
  INTEGER :: ios                                  ! Intended use:  COMPONENT_INPUT,
                                                  !   COMPONENT_OUTPUT, or COMPONENT_STATE
  INTEGER :: Nnodes                               ! Number of nodes (vertices) in mesh

 ! Keeping track of siblings:
  TYPE(MeshType),POINTER :: SiblingMesh           ! Pointer to mesh's (only) sibling

 ! Keeping track of elements:
  TYPE(ElemTabType), POINTER :: ElemTable(:)      ! Elements in the mesh, by type

 ! Mesh traversal:
  INTEGER :: nelemlist                            ! Number of elements in the list (ElemList)
  INTEGER :: nextelem                             ! Next element in the list
```

* There is no way to actually prevent anyone from changing the values that shouldn't be changed after the mesh is committed, so it is up to the module developer to adhere to this requirement.

```
  TYPE(ElemListType), POINTER :: ElemList(:)      ! All of the elements in the mesh

 ! Node position and reference orientation, which are always allocated (shared between siblings):
  REAL(ReKi), POINTER :: Position(:,:)            ! XYZ coordinate of node (3,NNodes)
  REAL(ReKi), POINTER :: RefOrientation(:,:,:)    ! Original/reference orientation (Direction
                                                  !    Cosine Matrix [DCM]) (3,3,NNodes)

 ! Fields available on the nodes (not shared between siblings):
! the last dimension of each of these has range 1:nnodes for the mesh being represented
! only some of these would be allocated, depending on what's being represented on the mesh.
  REAL(ReKi), ALLOCATABLE :: Force(:,:)           ! Field: Force vectors (3,NNodes)
  REAL(ReKi), ALLOCATABLE :: Moment(:,:)          ! Field: Moment vectors (3, NNodes)
  REAL(ReKi), ALLOCATABLE :: Orientation(:,:,:)   ! Field: Direction Cosine Matrix (3,3, NNodes)
  REAL(ReKi), ALLOCATABLE :: TranslationDisp(:,:) ! Field: Translational displacements (3, NNodes)
  REAL(ReKi), ALLOCATABLE :: RotationVel(:,:)     ! Field: Rotational velocities (3, NNodes)
  REAL(ReKi), ALLOCATABLE :: TranslationVel(:,:)  ! Field: Translational velocities (3, NNodes)
  REAL(ReKi), ALLOCATABLE :: RotationAcc(:,:)     ! Field: Rotational accelerations (3, NNodes)
  REAL(ReKi), ALLOCATABLE :: TranslationAcc(:,:)  ! Field: Translational accelerations (3, NNodes)
  REAL(ReKi), ALLOCATABLE :: Scalars(:,:)         ! Scalars (nScalars,Nnodes)
  INTEGER                 :: nScalars             ! Stores value of nScalars when created

 ! Fields available on the elements (NOT IMPLEMENTED, YET):
! REAL(ReKi), ALLOCATABLE :: ElementScalars(:,:)  ! Scalars associated with elements
! INTEGER                 :: nElementScalars      ! Stores value of nElementScalars when created

END TYPE MeshType

! Properties of a given element:
TYPE, PUBLIC :: ElemRecType
  INTEGER                    :: Xelement              ! Which kind of element
  INTEGER                    :: Nneighbors            ! How many neighbors
  INTEGER, POINTER           :: ElemNodes(:)          ! List of the element's nodes
  TYPE(ElemRecType), POINTER :: Neighbors(:)          ! List of the element's neighbors
END TYPE ElemRecType

! Table of all elements of a particular type:
TYPE, PUBLIC :: ElemTabType
  INTEGER                    :: nelem,                ! Number of elements of this type
  INTEGER                    :: Xelement              ! Which kind of elements
  TYPE(ElemRecType), POINTER :: Elements(:)           ! List of all elements of this type
END TYPE ElemTabType

! List of all elements in mesh (may be different element types, but must be same spatial dimension):
TYPE, PUBLIC :: ElemListType
  TYPE(ElemRecType), POINTER :: Element
END TYPE ElemListType
```

Subroutine: MeshCreate

Takes a blank, uninitialized instance of Type(MeshType) and defines the number of nodes in the mesh. Optional arguments indicate the fields that will be allocated and associated with the nodes of the mesh. The fields that may be associated with the mesh nodes are Force, Moment, Orientation, Rotation, TranslationDisp, RotationVel, TranslationVel, RotationAcc, TranslationAcc, and an arbitrary number of Scalars. See the definition of ModMeshType for descriptions of these fields.

Required arguments:

```
    TYPE(MeshType), INTENT(INOUT) :: BlankMesh ! Mesh to be created
    INTEGER,        INTENT(IN   ) :: IOS       ! Intended use:
                                               !   COMPONENT_INPUT
                                               !   COMPONENT_OUTPUT
```

```
                                          !   COMPONENT_STATE
INTEGER,        INTENT(IN    ) :: Nnodes   ! Number of nodes in mesh
INTEGER,        INTENT(   OUT) :: ErrStat  ! Error status/level
CHARACTER(*),   INTENT(   OUT) :: ErrMess  ! Error message
```

Optional arguments:

```
LOGICAL,INTENT(IN):: Force           ! If present and true, allocate Force field
LOGICAL,INTENT(IN):: Moment          ! If present and true, allocate Moment field
LOGICAL,INTENT(IN):: Orientation     ! If present and true, allocate Orientation field
LOGICAL,INTENT(IN):: TranslationDisp ! If present and true, allocate TranslationDisp field
LOGICAL,INTENT(IN):: TranslationVel  ! If present and true, allocate TranslationVel field
LOGICAL,INTENT(IN):: RotationVel     ! If present and true, allocate RotationVel field
LOGICAL,INTENT(IN):: TranslationAcc  ! If present and true, allocate TranslationAcc field
LOGICAL,INTENT(IN):: RotationAcc     ! If present and true, allocate RotationAcc field
INTEGER,INTENT(IN):: nScalars        ! If present and > 0, allocate nScalars Scalars
```

Subroutine: MeshPositionNode

For a given node in a mesh, assign the coordinates of the node in the global coordinate space. If an Orient argument is included, the node will also be assigned the specified orientation (orientation is assumed to be the identity matrix if omitted). Returns a non-zero value in ErrStat if Inode is outside the range 1..Nnodes.

Required arguments:

```
TYPE(MeshType),  INTENT(INOUT) :: Mesh        ! Mesh being spatio-located
INTEGER(IntKi),  INTENT(IN   ) :: Inode       ! Number of node being located
REAL(ReKi),      INTENT(IN   ) :: Pos(3)      ! Xi,Yi,Zi, coordinates of node
INTEGER(IntKi),  INTENT(  OUT) :: ErrStat     ! Error code
CHARACTER(*),    INTENT(  OUT) :: ErrMess     ! Error message
```

Optional arguments:

```
REAL(ReKi),      INTENT(IN   ) :: Orient(3,3) ! Orientation (direction cosine matrix) of
                                              !    node; identity by default
```

Subroutine: MeshConstructElement

Given a mesh and an element name, construct an element whose vertices are the node indices listed as the remaining arguments of the call to MeshConstructElement. The adjacency of elements is implied when elements are created that share some of the same nodes. Returns a non-zero value on error.

[Implementation note: the routine is defined as an F90 module procedure to allow variable length list of node indices.]

Required arguments:

```
TYPE(MeshType),       INTENT(INOUT) :: Mesh        ! Mesh being constructed
INTEGER(IntKi),       INTENT(IN)    :: Xelement    ! See Element Names
INTEGER(IntKi),       INTENT(OUT)   :: ErrStat     ! Error code
CHARACTER(*),         INTENT(OUT)   :: ErrMess     ! Error message
INTEGER,              INTENT(IN   ) :: P1, P2, ... ! Node index list (variable length)
```

Subroutine: MeshCommit

Given a mesh that has been created, spatio-located, and constructed, commit the definition of the mesh, making it ready for initialization and use. Explicitly committing a mesh provides the opportunity to precompute traversal information, neighbor lists and other information about the mesh. Returns non-zero in value of ErrStat on error.

Required arguments:

```
TYPE(MeshType),              INTENT(INOUT) :: Mesh    ! Mesh being committed
INTEGER(IntKi),              INTENT(OUT)   :: ErrStat ! Error code
CHARACTER(*),                INTENT(OUT)   :: ErrMess ! Error message
```

Subroutine: MeshCopy

Given an existing mesh and a destination mesh, create a completely new copy, a sibling, or update the fields of a second existing mesh from the first mesh. When CtrlCode is MESH_NEWCOPY or MESH_SIBLING, the destination mesh must be a blank, uncreated mesh.

If CtrlCode is MESH_NEWCOPY, an entirely new copy of the mesh is created, including all fields, with the same data values as the original, but as an entirely separate copy in memory. The new copy is in the same state as the original—if the original has not been committed, neither is the copy; in this case, an all-new copy of the mesh must be committed separately.

If CtrlCode is MESH_SIBLING, the destination mesh is created with the same mesh and position/reference orientation information of the source mesh, and this new sibling is added to the end of the list for the set of siblings. Siblings may have different fields (other than Position and RefOrientation). Therefore, for a sibling, it is necessary, as with MeshCreate, to indicate the fields the sibling will have using optional arguments. *Sibling meshes should not be created unless the original mesh has been committed first.*

If CtrlCode is MESH_UPDATECOPY, all of the allocatable fields of the destination mesh are updated with the values of the fields in the source. (The underlying mesh is untouched.) The mesh and field definitions of the source and destination meshes must match and both must have been already committed. The destination mesh may be an entirely different copy or it may be a sibling of the source mesh.

Required arguments:

```
TYPE(MeshType), INTENT(INOUT) :: SrcMesh  ! Mesh being copied
TYPE(MeshType), INTENT(INOUT) :: DestMesh ! Copy of mesh
INTEGER(IntKi), INTENT(IN   ) :: CtrlCode ! MESH_NEWCOPY, MESH_SIBLING, or MESH_UPDATECOPY
INTEGER(IntKi), INTENT(  OUT) :: ErrStat  ! Error code
CHARACTER(*),   INTENT(  OUT) :: ErrMess  ! Error message
```

Optional arguments (used only if CtrlCode is MESH_SIBLING):

```
LOGICAL,INTENT(IN):: Force           ! If present and true, allocate Force field
LOGICAL,INTENT(IN):: Moment          ! If present and true, allocate Moment field
LOGICAL,INTENT(IN):: Orientation     ! If present and true, allocate Orientation field
LOGICAL,INTENT(IN):: TranslationDisp ! If present and true, allocate TranslationDisp field
LOGICAL,INTENT(IN):: TranslationVel  ! If present and true, allocate TranslationVel field
```

```
LOGICAL,INTENT(IN):: RotationVel     ! If present and true, allocate RotationVel field
LOGICAL,INTENT(IN):: TranslationAcc  ! If present and true, allocate TranslationAcc field
LOGICAL,INTENT(IN):: RotationAcc     ! If present and true, allocate RotationAcc field
INTEGER,INTENT(IN):: nScalars        ! If present and > 0, allocate nScalars Scalars
```

Subroutine: MeshNextElement (mesh traversal)

Given a control code and a mesh that has been committed, retrieve the next element in the mesh. Used to traverse mesh element by element. On entry, the CtrlCode argument contains a control code: zero indicates start from the beginning, an integer between 1 and Mesh%Nelemlist returns that element, and MESH_NEXT means return the next element in traversal. On exit, CtrlCode contains the status of the traversal in (zero or MESH_NOMOREELEMS). The routine optionally outputs the index of the element in the mesh's element list, the name of the element (see "Element Names"), and a pointer to the element.

The returned index of the element can be used to access the indices of the nodes making up the element from the MeshType. For example, the returned value of Ielement can be used to determine the indices for the element: Mesh%ElemList%Element(Ielement)%ElemNodes. See also the section titled, "Accessing a Specific Element and Node Value in a Mesh" in this appendix.

Required arguments:

```
TYPE(MeshType),           INTENT(INOUT) :: Mesh      ! Mesh being constructed
INTEGER(IntKi),           INTENT(INOUT) :: CtrlCode  ! CtrlCode
INTEGER(IntKi),           INTENT(OUT)   :: ErrStat   ! Error code
CHARACTER(*),             INTENT(OUT)   :: ErrMess   ! Error message
```

Optional arguments:

```
INTEGER(IntKi),           INTENT(OUT)   :: Ielement  ! Element index
INTEGER(IntKi),           INTENT(OUT)   :: Xelement  ! See Element Names
TYPE(ElemRecType),POINTER,INTENT(INOUT) :: ElemRec   ! Return element
```

Subroutine: MeshElemNumNeighbors (not yet implemented)

Given a mesh and the name and index of an element, return the number of neighboring elements of the element.

Required arguments:

```
TYPE(MeshType),INTENT(INOUT) :: Mesh         ! Mesh being referenced
INTEGER(IntKi),INTENT(IN)    :: Thiselement  ! Element index of home element
INTEGER(IntKi),INTENT(IN)    :: ThisXelement ! Name of home element
INTEGER(IntKi),INTENT(OUT)   :: Nneighbors   ! Number of neighboring elements
INTEGER(IntKi),INTENT(OUT)   :: ErrStat     ! Error code
CHARACTER(*),INTENT(OUT)     :: ErrMess     ! Error message
```

Subroutine: MeshNextElemNeighbor (neighbor traversal) (not yet implemented)

Given a mesh and the name and index of an element, return the next neighbor of the element. On entry, the Istat argument contains a control code: zero indicates start from the beginning, an integer between 1 and the number of neighbors (as returned by MeshElemNumNeighbors) returns that neighbor, and MESH_NEXT means return the next element to be returned in traversal. Returns the index of the element in the home element's neighbor list, the name of the

neighbor element (see the "Element Names" section of this appendix), the number of vertices in the neighbor element, and the status of the traversal in Istat (zero or MESH_NOMOREELEMS). See also the section titled, "Accessing a Specific Element and Node Value in a Mesh" in this appendix.

Required arguments:

```
TYPE(MeshType),INTENT(INOUT) :: Mesh         ! Mesh being committed
INTEGER(IntKi),INTENT(INOUT) :: Istat        ! Control/status of traversal
INTEGER(IntKi),INTENT(IN)    :: Thiselement  ! Element index of home element
INTEGER(IntKi),INTENT(IN)    :: ThisXelement ! Name of home element
INTEGER(IntKi),INTENT(OUT)   :: Ielement     ! Element index of neighbor
INTEGER(IntKi),INTENT(OUT)   :: Xelement     ! Name of neighbor element
INTEGER(IntKi),INTENT(OUT)   :: Nvertices    ! Number of nodes neighbor element
INTEGER(IntKi),INTENT(OUT)   :: ErrStat      ! Error code
CHARACTER(*),INTENT(OUT)     :: ErrMess      ! Error message
```

Subroutine: MeshPack

Given a mesh and allocatable buffers of type INTEGER(IntKi), REAL(ReKi), and REAL(DbKi), return the mesh information compacted into consecutive elements of the corresponding buffers. This would be done to allow subsequent writing of the buffers to a file for restarting later. The sense of the name is "pack the data from the mesh into buffers". IMPORTANT: **MeshPack** allocates the three buffers. It is incumbent upon the calling program to deallocate the buffers when they are no longer needed. For sibling meshes, **MeshPack** should be called separately for each sibling, because the fields allocated with the siblings are separate and unique to each sibling.

Required arguments:

```
TYPE(MeshType),              INTENT(INOUT) :: Mesh      ! Mesh being packed/saved
REAL(ReKi),     ALLOCATABLE, INTENT(  OUT) :: ReBuf(:)  ! Real buffer
REAL(DbKi),     ALLOCATABLE, INTENT(  OUT) :: DbBuf(:)  ! Double buffer
INTEGER(IntKi), ALLOCATABLE, INTENT(  OUT) :: IntBuf(:) ! Integer buffer
INTEGER(IntKi),              INTENT(  OUT) :: ErrStat   ! Error status
CHARACTER(*),                INTENT(  OUT) :: ErrMess   ! Error message
```

Subroutine: MeshUnpack

Given a blank, uncreated mesh and buffers of type INTEGER(IntKi), REAL(ReKi), and REAL(DbKi), unpack the mesh information from the buffers. This would be done to recreate a mesh after reading in the buffers on a restart of the program. The sense of the name is "unpack the mesh from buffers." The resulting mesh will be returned in the exact state as when the data in the buffers was packed using **MeshPack**. If the mesh has an already recreated sibling mesh from a previous call to **MeshUnpack**, specify the existing sibling as an optional argument so that the sibling relationship is also recreated.

Required arguments:

```
TYPE(MeshType),              INTENT(INOUT) :: Mesh
REAL(ReKi),     ALLOCATABLE, INTENT(IN   ) :: Re_Buf(:)
REAL(DbKi),     ALLOCATABLE, INTENT(IN   ) :: Db_Buf(:)
INTEGER(IntKi), ALLOCATABLE, INTENT(IN   ) :: Int_Buf(:)
INTEGER(IntKi),              INTENT(  OUT) :: ErrStat
CHARACTER(*),                INTENT(  OUT) :: ErrMess
```

Optional argument (not yet implemented):

```
TYPE(MeshType),INTENT(IN)         ::  Sibling  ! Existing sibling for reunion
```

Subroutine: MeshDestroy

Destroy the given mesh and deallocate all of its data. If the optional IgnoreSibling argument is set to TRUE, destroying a sibling in a set has no effect on the other siblings other than to remove the victim from the list of siblings. If IgnoreSibling is omitted or is set to FALSE, all of the other siblings in the set will be destroyed as well.

Required arguments:

```
TYPE(MeshType),INTENT(INOUT) :: Mesh          ! Mesh to destroy
INTEGER(IntKi),INTENT(OUT)   :: ErrStat       ! Error code
CHARACTER(*),  INTENT(OUT)   :: ErrMess       ! Error message
```

Optional argument:

```
LOGICAL,       INTENT(IN)    :: IgnoreSibling   ! true=keep sibling; false=destroy it
```

Accessing a Specific Element and Node Value in a Mesh

There is no subroutine provided to access a specific element in a mesh. Rather, one may accesses the desired element from the mesh itself. For example:
Mesh%ElemTable(ELEMENT_LINE2)(1:2,*i*) contains the node indices for the i^{th} 2-node line element in the mesh. Likewise, the fields associated with the nodes of the element are accessible using the node indices, assuming the field is allocated for the nodes of the mesh. For example, the positions of the two nodes of the i^{th} ELEMENT_LINE2 element, above, are accessible as:

```
mesh%Position(1:3, mesh%ElemTable(ELEMENT_LINE2)%Elements(i)%ElemNodes(1))  ! XYZ of node 1
mesh%Position(1:3, mesh%ElemTable(ELEMENT_LINE2)%Elements(i)%ElemNodes(2))  ! XYZ of node 2
```

The Position and RefOrientation fields are always defined for a mesh. The availability of other fields depends on how the mesh was created using MeshCreate. When in doubt, test with ALLOCATED(*mesh%field*) before accessing a field.

Element Names:

Parameters for each kind of element allowed in ModMesh are named as follows:

```
ELEMENT_POINT
ELEMENT_LINE2  ELEMENT_LINE3
ELEMENT_TRI3   ELEMENT_TRI6
ELEMENT_QUAD4  ELEMENT_QUAD8
ELEMENT_TET4   ELEMENT_TET10
ELEMENT_HEX8   ELEMENT_HEX20
ELEMENT_WEDGE6 ELEMENT_WEDGE15
```

Note that only ELEMENT_POINT and ELEMENT_LINE2 are currently implemented.

Appendix H: FAST Registry for Automatic Code Generation

This appendix describes the FAST Registry, a utility available to component developers for table-based automatic generation of code and data types for the FAST component module interface.

Description

The Registry consists of tables stored in a text file called "Registry.txt," and a program (distributed with and compiled with FAST) that reads the Registry when FAST is compiled. This program automatically generates Fortran code for the ModuleName_Types module, which includes the module's derived data types (see Table 2) and implementations of the system input and output extrapolation/interpolation, Pack{*TypeName*}/Unpack{*TypeName*}, and Copy{*TypeName*}/Destroy{*TypeName*} routines (see Table 1) that operate on these types.

The Registry allows the developer to specify the data types for a module once and in a single location, automating the time consuming and error-prone task of generating the code. The tables in the Registry serve as a data dictionary for improving understandability and maintainability of the code.

The FAST Registry is borrowed from a mechanism that was originally developed at NCAR for the Weather Research and Forecast (WRF) model software[*]. Under the current implementation, the FAST software uses a single Registry file that contains entries for all the modules comprising the suite; however, each module in the FAST framework contains its own Registry "sub-file" (a Registry file containing the data types for only that module), and the sub-files for all of the modules will be included together to form a single FAST Registry file. When new modules are added to FAST, new entries must be added to the table (i.e., a new Registry sub-file for the module must be created and included in the main Registry file).

Syntax

The Registry file syntax allows continuation lines, comments, conditional compilation directives, and include statements to bring in Registry statements from other files. Lines that end with a "\" character continue to the next line. Comments in the registry begin with a "#" character and extend to the right to the end of the line. Certain elements of an entry are allowed to contain spaces but then must be enclosed in double quotes ("). Elements of a registry may not contain quote characters. A hyphen "-" may be used for an entry if the value does not matter or is the default. Hyphens may be omitted if they appear as the last element(s) of a line.

Logically, the Registry file is a collection of tables that describe the derived types and the types and dimensions of the fields that make up the derived types. Each entry in the Registry is one line (possibly continued). The first element of a Registry line specifies which table the line is a part of. Currently, the FAST Registry has only two tables, "dimspec" and "typedef."

Dimspec Entries

Dimspec entries are used to define dimensions that will be used subsequently in defining multi-dimensional arrays as fields within a type. (The dimensions may be defined in-line in the entry

[*] http://www.mmm.ucar.edu/wrf/WG2/software_2.0/registry_schaffer.pdf

that defines the array as shown in the typedef entry below; however, one may wish to use dimspec if a dimension is to be used repeatedly). The form of a dimspec entry is:

- *Table membership:* The keyword "dimspec" (no quotes)
- *DimName:* The name of the dimension
- *HowDefined:* Specification of how the range of the dimension is defined

The *DimName* is a character-string name that is used in typedef entries to indicate a dimension of an array in a typedef entry in the registry.

HowDefined specifies how the ranges of the dimension will be defined when the array is allocated in the running FAST program. The *HowDefined* entry can be one of:

- constant=[<start constant>:]<end constant>
- the word "deferred" or the colon character ":"

The "constant=" indicates that the integer constant is specified on the right hand side of the equal sign "=" will be used to dimension the array. "Deferred" or ":" means that the field will be allocated by the program and that the registry should simply generate this field as allocatable. The "constant=" syntax allows a starting index for the range to be specified, separated from the ending index by a colon (for generating Fortran code only). If no starting index is specified, it defaults to 1.

Typedef Entries

Typedef entries are used to specify types that will appear in the ModuleName_Types module for a contributed module. There will be a typedef entry for each field of a derived data type. The form of a typedef entry is:

- *Table membership:* The keyword "typedef" (no quotes)
- *ModuleName*: The name of the FAST module the entry is defined for
- *TypeName:* The name of the derived type this field is a member of
- *Type:* The type of the field
- *Name:* The name of the field
- *Dims:* A string denoting the dimensionality of the field or a hypen (-) if scalar
- *IO:* Not used, currently. A placeholder for specifying I/O on the field later
- *DataName:* The name of the field as it is known outside the program
- *Description:* A short description of the field
- *Units:* Units for the variable

Fields are added to a type by listing additional entries. *ModuleName* defines the name of the FAST module the entry pertains to. *TypeName* is the name of the derived type being defined with a new field named *Name.*

The *Type* element specifies the type of the field being added to the derived type. It may be a simple type (ReKi, IntKi), a derived type previously defined in the Registry, or MeshType.

Dims specifies the dimensionality of the field being added to the derived type. The entry is a string of dimension names specified using dimspec entries above. If the developer prefers, constant ranges may be specified inline. The colon character ":" may be used to indicate a deferred dimension. Use " {" and "}" to avoid ambiguous groupings.

The *IO* entry is not currently used but is included as a placeholder for future functionality. Use a hypen for this field.

DataName is also not currently used, but is intended to allow a variable to be known by a different name outside the program to facilitate off-line coupling to other programs. Use a hypen if the *DataName* is the same as *Name.*

Description is a character string provided to document what the variable is for someone reading the Registry file and could also be output as metadata for self-describing datasets in the future. The string may contain spaces if enclosed by double quotes (").

Units is a character string provided to document the units of the variable. May be output as metadata for self-describing datasets. Use a hyphen if there are no units.

Registry Output

A typedef entry for a given *TypeName* and *ModuleName* defines a field (a scalar, array, mesh or other defined type) as a member of type *TypeName* in the *ModuleName*_Types module. Each *ModuleName*_Types module is defined in a file named "*ModuleName*_Types.f90."

The first typedef entry for a given *ModuleName* in the registry causes the *ModuleName*_Types.f90 file to be opened. The first typedef entry for a given *TypeName* causes that derived type to be defined in the module.

The registry will generate as many *ModuleName*_Types.f90 files as there are different *ModuleName* values used in the Registry file. Developers should not make changes directly to an automatically generated *ModuleName*_Types.f90 file because changes will be lost the next time the code is compiled and the file overwritten.

Example Registry File

The following page contains an example set of Registry file entries that define the user-defined data types listed in Table 2 for a module named "ModuleName."

When FAST is compiled, the registry program processes this Registry file line-by-line; the types defined in the Registry and the accompanying subroutines for each type are automatically generated in a module named "ModuleName" that is contained in a file called "ModuleName_Types.f90."

```
###################################################################################################################################
# Registry for ModuleName in the FAST Modularization Framework
# This Registry file is used to create MODULE ModuleName_Types, which contains all of the user-defined types needed in ModuleName.
# It also contains copy, destroy, pack, and unpack routines associated with each defined data types.
# Entries are of the form
# keyword <ModuleName/ModName> <TypeName>    <FieldType>    <FieldName>  <Dims>  <IO> <DNAME> <DESCRIP> <UNITS>
#
# Use ^ as a shortcut for the value from the previous line.
###################################################################################################################################

# ..... Initialization data .......................................................................................................
# Define inputs that the initialization routine may need here:
#   e.g., the name of the input file, the file root name, etc.
typedef ModuleName/ModName InitInputType CHARACTER(1024) InputFile  - - -   "Name of the input file; remove if there is no file" -

# Define outputs from the initialization routine here:
typedef   ^              InitOutputType CHARACTER(10) WriteOutputHdr  {:} - -   "Names of the output-to-file channels" -
typedef   ^              InitOutputType CHARACTER(10) WriteOutputUnt  {:}- -   "Units of the output-to-file channels" -

# ..... States ....................................................................................................................
# Define continuous (differentiable) states here:
typedef    ^             ContinuousStateType ReKi     DummyContState - - -  "Remove this variable if you have continuous states" -

# Define discrete (nondifferentiable) states here:
typedef   ^              DiscreteStateType   ReKi     DummyDiscState - - -  "Remove this variable if you have discrete states" -

# Define constraint states here:
typedef   ^              ConstraintStateType ReKi     DummyConstrState - - - "Remove this variable if you have constraint states" -

# Define any data that are not considered actual states here:
#   e.g. data used only for efficiency purposes (indices for searching in an array, copies of previous calculations of output
#   at a given time, etc.)
typedef   ^              OtherStateType      IntKi    DummyOtherState - - - "Remove this variable if you have other states" -

# ..... Parameters ................................................................................................................
# Define parameters here:
# Time step for integration of continuous states (if a fixed-step integrator is used) and update of discrete states:
typedef  ^              ParameterType       DbKi     DT - - - "Time step for cont. state integration & disc. state update" seconds

# ..... Inputs ....................................................................................................................
# Define inputs that are contained on the mesh here:
#typedef   ^               InputType          MeshType  MeshedInput - - -     "Meshed data" -
# Define inputs that are not on this mesh here:
typedef   ^              InputType           ReKi     DummyInput  - - -     "Remove this variable if you have input data" -

# ..... Outputs ...................................................................................................................
# Define outputs that are contained on the mesh here:
#typedef ModuleName ModName_OutputType         MeshType MeshedOutput - - -    "Meshed data" -
# Define outputs that are not on this mesh here:
typedef   ^              OutputType          ReKi     DummyOutput  -  - -   "Remove this variable if you have output data" -
typedef   ^                ^                 ReKi     WriteOutput {:} - -   "Example of data to be written to an output file" "s,-"
```

Appendix I: Example Driver Program for Modules in the FAST Modular Framework

```
!**********************************************************************************************************************************
! ModuleName_DriverCode: This code tests the template modules
!..................................................................................................................................
! LICENSING
! Copyright (C) 2012  National Renewable Energy Laboratory
!
!    This file is part of ModuleName.
!
!    ModuleName is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as
!    published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
!
!    This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty
!    of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more details.
!
!    You should have received a copy of the GNU General Public License along with ModuleName.
!    If not, see <http://www.gnu.org/licenses/>.
!
!**********************************************************************************************************************************
PROGRAM TestTemplate

   USE NWTC_Library
   USE ModuleName
   USE ModuleName_Types

   IMPLICIT NONE

   INTEGER(IntKi), PARAMETER                       :: NumInp = 1          ! Number of inputs sent to ModName_UpdateStates

      ! Program variables

   REAL(DbKi)                                      :: Time                ! Variable for storing time, in seconds
   REAL(DbKi)                                      :: TimeInterval        ! Interval between time steps, in seconds
   REAL(DbKi)                                      :: InputTime(NumInp)   ! Variable for storing time associated with inputs, in seconds

   TYPE(ModName_InitInputType)                     :: InitInData          ! Input data for initialization
   TYPE(ModName_InitOutputType)                    :: InitOutData         ! Output data from initialization

   TYPE(ModName_ContinuousStateType)               :: x                   ! Continuous states
   TYPE(ModName_DiscreteStateType)                 :: xd                  ! Discrete states
   TYPE(ModName_ConstraintStateType)               :: z                   ! Constraint states
   TYPE(ModName_ConstraintStateType)               :: Z_residual          ! Residual of the constraint state functions (Z)
   TYPE(ModName_OtherStateType)                    :: OtherState          ! Other/optimization states

   TYPE(ModName_ParameterType)                     :: p                   ! Parameters
   TYPE(ModName_InputType)                         :: u(NumInp)           ! System inputs
   TYPE(ModName_OutputType)                        :: y                   ! System outputs
```

```
TYPE(ModName_ContinuousStateType)                :: dxdt               ! First time derivatives of the continuous states

TYPE(ModName_PartialOutputPInputType)            :: dYdu               ! Partial derivatives of the output functions
                                                                       !  (Y) with respect to the inputs (u)
TYPE(ModName_PartialContStatePInputType)         :: dXdu               ! Partial derivatives of the continuous state
                                                                       !  functions (X) with respect to the inputs (u)
TYPE(ModName_PartialDiscStatePInputType)         :: dXddu              ! Partial derivatives of the discrete state
                                                                       !  functions (Xd) with respect to the inputs (u)
TYPE(ModName_PartialConstrStatePInputType)       :: dZdu               ! Partial derivatives of the constraint state
                                                                       !  functions (Z) with respect to the inputs (u)
TYPE(ModName_PartialOutputPContStateType)        :: dYdx               ! Partial derivatives of the output functions (Y)
                                                                       !  with respect to the continuous states (x)
TYPE(ModName_PartialContStatePContStateType)     :: dXdx               ! Partial derivatives of the continuous state funct-
                                                                       !  ions (X) with respect to the continuous states (x)
TYPE(ModName_PartialDiscStatePContStateType)     :: dXddx              ! Partial derivatives of the discrete state funct-
                                                                       !  ions (Xd) with respect to continuous states (x)
TYPE(ModName_PartialConstrStatePContStateType)   :: dZdx               ! Partial derivatives of the constraint state funct-
                                                                       !  ions (Z) with respect to the continuous states (x)
TYPE(ModName_PartialOutputPDiscStateType)        :: dYdxd              ! Partial derivatives of the output functions (Y)
                                                                       !  with respect to the discrete states (xd)
TYPE(ModName_PartialContStatePDiscStateType)     :: dXdxd              ! Partial derivatives of the continuous state funct-
                                                                       !  ions (X) with respect to the discrete states (xd)
TYPE(ModName_PartialDiscStatePDiscStateType)     :: dXddxd             ! Partial derivatives of the discrete state funct-
                                                                       !  ions (Xd) with respect to the discrete states (xd)
TYPE(ModName_PartialConstrStatePDiscStateType)   :: dZdxd              ! Partial derivatives of the constraint state funct-
                                                                       !  ions (Z) with respect to the discrete states (xd)
TYPE(ModName_PartialOutputPConstrStateType)      :: dYdz               ! Partial derivatives of the output functions (Y)
                                                                       !  with respect to the constraint states (z)
TYPE(ModName_PartialContStatePConstrStateType)   :: dXdz               ! Partial derivatives of the continuous state funct-
                                                                       !  ions (X) with respect to the constraint states (z)
TYPE(ModName_PartialDiscStatePConstrStateType)   :: dXddz              ! Partial derivatives of the discrete state funct-
                                                                       !  ions (Xd) with respect to constraint states (z)
TYPE(ModName_PartialConstrStatePConstrStateType) :: dZdz               ! Partial derivatives of the constraint state funct-
                                                                       !  ions (Z) with respect to the constraint states (z)


INTEGER(IntKi)                                   :: n                  ! Loop counter (for time step)
INTEGER(IntKi)                                   :: ErrStat            ! Status of error message
CHARACTER(1024)                                  :: ErrMsg             ! Error message if ErrStat /= ErrID_None


REAL(ReKi), ALLOCATABLE                          :: Re_SaveAry  (:)    ! Array to store reals in packed data structure
REAL(DbKi), ALLOCATABLE                          :: Db_SaveAry  (:)    ! Array to store doubles in packed data structure
INTEGER(IntKi), ALLOCATABLE                      :: Int_SaveAry (:)    ! Array to store integers in packed data structure

!.....................................................................................................
! Routines called in initialization
!.....................................................................................................
```

```
      ! Populate the InitInData data structure here:

InitInData%InputFile = 'MyInputFileName.inp'

      ! Set the driver's request for time interval here:

TimeInterval = 0.25                       ! Glue code's request for delta time (likely based on information from other modules)


      ! Initialize the module

CALL ModName_Init( InitInData, u(1), p,  x, xd, z, OtherState, y, TimeInterval, InitOutData, ErrStat, ErrMsg )
IF ( ErrStat /= ErrID_None ) THEN         ! Check if there was an error and do something about it if necessary
   CALL WrScr( ErrMsg )
END IF


      ! Destroy initialization data

CALL ModName_DestroyInitInput(  InitInData,  ErrStat, ErrMsg )
CALL ModName_DestroyInitOutput( InitOutData, ErrStat, ErrMsg )


!.....................................................................................................
! Routines called in loose coupling -- the glue code may implement this in various ways
!.....................................................................................................


DO n = 0,2

   Time = n*TimeInterval
   InputTime(1) = Time

      ! Modify u (likely from the outputs of another module or a set of test conditions) here:


      ! Calculate outputs at n

   CALL ModName_CalcOutput( Time, u(1), p, x, xd, z, OtherState, y, ErrStat, ErrMsg )
   IF ( ErrStat /= ErrID_None ) THEN         ! Check if there was an error and do something about it if necessary
      CALL WrScr( ErrMsg )
   END IF


      ! Get state variables at next step: INPUT at step n, OUTPUT at step n + 1

   CALL ModName_UpdateStates( Time, n, u, InputTime, p, x, xd, z, OtherState, ErrStat, ErrMsg )
   IF ( ErrStat /= ErrID_None ) THEN         ! Check if there was an error and do something about it if necessary
```

```
         CALL WrScr( ErrMsg )
      END IF


   END DO


   !.....................................................................................................
   ! Routines called in tight coupling -- time marching only
   !.....................................................................................................

   DO n = 0,2

      Time = n * TimeInterval   ! Note that the discrete states must be updated only at the TimeInterval defined in initialization

         ! set inputs (u) here:
!      u =

         ! Update constraint states at Time

      ! DO

         CALL ModName_CalcConstrStateResidual( Time, u(1), p, x, xd, z, OtherState, Z_residual, ErrStat, ErrMsg )
         IF ( ErrStat /= ErrID_None ) THEN      ! Check if there was an error and do something about it if necessary
            CALL WrScr( ErrMsg )
         END IF

         ! z =

      ! END DO



         ! Calculate the outputs at Time

      CALL ModName_CalcOutput( Time, u(1), p, x, xd, z, OtherState, y, ErrStat, ErrMsg )
      IF ( ErrStat /= ErrID_None ) THEN          ! Check if there was an error and do something about it if necessary
         CALL WrScr( ErrMsg )
      END IF


         ! Calculate the continuous state derivatives at Time

      CALL ModName_CalcContStateDeriv( Time, u(1), p, x, xd, z, OtherState, dxdt, ErrStat, ErrMsg )
      IF ( ErrStat /= ErrID_None ) THEN          ! Check if there was an error and do something about it if necessary
         CALL WrScr( ErrMsg )
      END IF
```

```
      ! Update the discrete state from step n to step n+1
      ! Note that the discrete states must be updated only at the TimeInterval defined in initialization

   CALL ModName_UpdateDiscState( Time, n, u(1), p, x, xd, z, OtherState, ErrStat, ErrMsg )
   IF ( ErrStat /= ErrID_None ) THEN          ! Check if there was an error and do something about it if necessary
      CALL WrScr( ErrMsg )
   END IF


      ! Driver should integrate (update) continuous states here:

   !x = function of dxdt, x


      ! Jacobians required:

   CALL ModName_JacobianPInput( Time, u(1), p, x, xd, z, OtherState, dYdu=dYdu, dZdu=dZdu, ErrStat=ErrStat, ErrMsg=ErrMsg )
   IF ( ErrStat /= ErrID_None ) THEN          ! Check if there was an error and do something about it if necessary
      CALL WrScr( ErrMsg )
   END IF

   CALL ModName_JacobianPConstrState( Time, u(1), p, x, xd, z, OtherState, dYdz=dYdz, dZdz=dZdz, &
                                      ErrStat=ErrStat, ErrMsg=ErrMsg )
   IF ( ErrStat /= ErrID_None ) THEN          ! Check if there was an error and do something about it if necessary
      CALL WrScr( ErrMsg )
   END IF


END DO


   ! Destroy Z_residual and dxdt because they are not necessary anymore

CALL ModName_DestroyConstrState( Z_residual, ErrStat, ErrMsg )
IF ( ErrStat /= ErrID_None ) THEN   ! Check if there was an error and do something about it if necessary
   CALL WrScr( ErrMsg )
END IF

CALL ModName_DestroyContState( dxdt, ErrStat, ErrMsg )
IF ( ErrStat /= ErrID_None ) THEN   ! Check if there was an error and do something about it if necessary
   CALL WrScr( ErrMsg )
END IF

!.....................................................................................................
! Jacobian routines called in tight coupling
!.....................................................................................................

CALL ModName_JacobianPInput( Time, u(1), p, x, xd, z, OtherState, dYdu, dXdu, dXddu, dZdu, ErrStat, ErrMsg )
IF ( ErrStat /= ErrID_None ) THEN          ! Check if there was an error and do something about it if necessary
```

```
   CALL WrScr( ErrMsg )
END IF

CALL ModName_JacobianPContState( Time, u(1), p, x, xd, z, OtherState, dYdx, dXdx, dXddx, dZdx, ErrStat, ErrMsg )
IF ( ErrStat /= ErrID_None ) THEN          ! Check if there was an error and do something about it if necessary
   CALL WrScr( ErrMsg )
END IF

CALL ModName_JacobianPDiscState( Time, u(1), p, x, xd, z, OtherState, dYdxd, dXdxd, dXddxd, dZdxd, ErrStat, ErrMsg )
IF ( ErrStat /= ErrID_None ) THEN          ! Check if there was an error and do something about it if necessary
   CALL WrScr( ErrMsg )
END IF

CALL ModName_JacobianPConstrState( Time, u(1), p, x, xd, z, OtherState, dYdz, dXdz, dXddz, dZdz, ErrStat, ErrMsg )
IF ( ErrStat /= ErrID_None ) THEN          ! Check if there was an error and do something about it if necessary
   CALL WrScr( ErrMsg )
END IF


!.....................................................................................................................
! Routines to pack data (to restart later)
!.....................................................................................................................
CALL ModName_Pack(Re_SaveAry, Db_SaveAry, Int_SaveAry, u(1), p, x, xd, z, OtherState, y, ErrStat, ErrMsg)

IF ( ErrStat /= ErrID_None ) THEN
   CALL WrScr( ErrMsg )
END IF


!.....................................................................................................................
! Routine to terminate program execution
!.....................................................................................................................
CALL ModName_End( u(1), p, x, xd, z, OtherState, y, ErrStat, ErrMsg )

IF ( ErrStat /= ErrID_None ) THEN
   CALL WrScr( ErrMsg )
END IF


!.....................................................................................................................
! Routines to retreive packed data (unpack for restart)
!.....................................................................................................................
CALL ModName_Unpack( Re_SaveAry, Db_SaveAry, Int_SaveAry, u(1), p, x, xd, z, OtherState, y, ErrStat, ErrMsg )

IF ( ErrStat /= ErrID_None ) THEN
   CALL WrScr( ErrMsg )
END IF
```

```
   !.....................................................................................................
   ! Routines to copy data (not already tested)
   !.....................................................................................................



   !.....................................................................................................
   ! Routines to destroy data (not already tested)
   !.....................................................................................................

   IF ( ALLOCATED( Re_SaveAry  ) ) DEALLOCATE( Re_SaveAry )
   IF ( ALLOCATED( Db_SaveAry  ) ) DEALLOCATE( Db_SaveAry )
   IF ( ALLOCATED( Int_SaveAry ) ) DEALLOCATE( Int_SaveAry )

!   CALL ModName_DestroyPartialOutputPInput ( )  ! Jacobian Routine not yet implemented


   !.....................................................................................................
   ! Routine to terminate program execution (again)
   !.....................................................................................................

   CALL ModName_End( u(1), p, x, xd, z, OtherState, y, ErrStat, ErrMsg )
   IF ( ErrStat /= ErrID_None ) THEN
      CALL WrScr( ErrMsg )
   END IF


END PROGRAM TestTemplate
```

Appendix J: Using the NWTC Subroutine Library

The following documentation provides a general overview of how to use the NWTC Subroutine Library and what functionality it provides. It is valid for NWTC Subroutine Library version 2.02.00; for other versions of the software, please refer to the library's change log file for a list of modifications.

The NWTC Subroutine Library consists of several Fortran MODULES, each contained in its own source file. If you wish to access any of the parameters or routines from the library, add the statement

```
USE NWTC_Library
```

to the program unit requiring access. All of the modules in the NWTC Subroutine Library will be accessible through this statement (including ModMesh). The library's **NWTC_Init** subroutine must also be called before using any of the other routines or parameters in the library; it is not a problem to call **NWTC_Init** multiple times in the same program. To compile the library, see the instructions and source-file compilation order described in the heading of the "NWTC_Library.f90" file.

An overview of the library's most common parameters and routines is presented below. For a complete and more detailed list, please see the documentation provided with the NWTC Subroutine Library; the headings of the library's source files also list all available routines.

Parameters

The library provides many constants for precision, which are used to set KIND attributes for variables (i.e., to specify how many bytes each variable type contains). The SingPrec.f90 source file contains the constants set for single-precision compilation, and DoubPrec.f90 contains the constants for double-precision compilation. The three constants for default precision listed in Table 3 should be used for the majority of your variable declarations. If you have variable types that must not change between single- and double-precision (e.g., the code is reading a specific number of bytes from a binary file), use the constants for specific precision listed in Table 3.

The error codes in the library are designed to provide a standardize way to categorize the severity of warnings or errors encountered in program execution. This method allows the developer or user to stop the program (in the driver program—never stop the code in a module!) at a specified error level. You must check the error code after calling routines with ErrStat/ErrMsg arguments so that errors can be dealt with appropriately.

The library also includes several common mathematical constants involving π as well as factors to convert between degrees and radians and to convert between revolutions per minute (RPM) and radians per second (RPS).

Table 3. Some of the most commonly used parameters in the NWTC Subroutine Library.

Parameter	Description
Constants for Default Precision	
IntKi	Default KIND parameter for INTEGER (whole number) values
ReKi	Default KIND parameter for REAL (floating-point) values
DbKi	Default KIND parameter for DOUBLE (floating-point) values
Constants for Specific Precision	
B1Ki	KIND for one-byte INTEGER (whole number) values
B2Ki	KIND for two-byte INTEGER (whole number) values
B4Ki	KIND for four-byte INTEGER (whole number) values
B8Ki	KIND for eight-byte INTEGER (whole number) values
SiKi	KIND for four-byte REAL (floating-point) values
R8Ki	KIND for eight-byte REAL (floating-point) values
QuKi	KIND for 16-byte REAL (floating-point) values
Error Codes	
ErrID_None	Lowest error level: no error
ErrID_Info	Error level 1: an informational message
ErrID_Warn	Error level 2: a warning message
ErrID_Severe	Error level 3: a severe error
ErrID_Fatal	Highest error level: a fatal error
Mathematical Constants	
Pi	π: The ratio of a circle's circumference to its diameter
TwoPi	2π
PiBy2	π/2
TwoByPi	2/π
Factors to Convert Between Different Units	
D2R	Degrees to radians: π/180°
R2D	Radians to degrees: 180°/π
RPM2RPS	Revolutions per minute to radians per second: π/30-s
RPS2RPM	Radians per second to revolutions per minute: 30-s/π
Character Constants	
TAB	ASCII character 9 (the tab character)
NewLine	Character sequence for a new line in WrScr()

Routines

Table 4 contains a list of some of the most common routines in the library. Developers (particularly those in the FAST modularization framework) should use the routines provided in the library whenever possible. Using the library standardizes processes like error checking and also makes modifications easier (e.g., if you link your code to Program A, which cannot write to the screen in a standard way, you can modify the library routines to call the Program A's own functions to display messages).

Table 4. Some of the most commonly used routines in the NWTC Subroutine Library.

Routine	Description
I/O Routines to Open Files	
GetNewUnit	Returns the next unit number not currently connected to a file
OpenFInpFile	Opens a formatted (text) file for reading (input)
OpenFOutFile	Opens a formatted (text) file for writing (output)
OpenBInpFile	Opens a binary file for reading (input)
OpenBOutFile	Opens a binary file for writing (output)
I/O Routines to Read from Text Files	
ReadVar	Reads a variable from the next line of an input file
ReadAry	Reads an array from an input file, array elements separated by whitespace
ReadAryLines	Reads an array from an input file, one array element per line
ReadCom	Reads a comment line from an input file
ReadStr	Reads a string variable (including whitespace) from an input file
I/O Routines to Write to the Screen	
WrScr	Writes a string to the screen
WrOver	Writes a string over the last line written to the screen
WrNR	Write a string to the screen without using a line return
DispNVD	Writes the program name, version, and date to the screen
Routines to Retrieve and Manipulate Strings	
Num2LStr	Converts a numeric value to a left-aligned string
Conv2UC	Converts a string to upper case
CountWords	Counts the number of words in a string
CurDate	Returns the current date as a string
CurTime	Returns the current time as a string
CheckArgs	Returns the input file name specified from a command-line argument
GetPath	Parses the path name from the name of a given file
GetRoot	Parses the root name from the name of a given file
PathIsRelative	Determines if the given file name is an absolute or relative name
Mathematical Functions	
Cross_Product	Calculates the cross product of two vectors
Mean	Calculates the average value of an array
StdDevFn	Calculates the standard deviation of an array
AddOrSub2Pi	Converts an angle to a value within 2π of another angle
MPi2Pi	Converts an angle to a value between $-\pi$ and π
Routine to Compare Real Numbers	
EqualRealNos	Determines if two REAL numbers are within a certain tolerance
Routines for Searching	
LocateBin	Performs a binary search for a value in an array
LocateStp	Performs a stepwise search for a value in an array
Routines for Interpolation	
InterpBin	Performs interpolation using a binary search
InterpStp	Performs interpolation using a step-wise search

Appendix K: Fortran Compilation Options

Table 5. Some compilation options available in Intel and GNU Fortran.

	Intel Fortran (Windows)	Intel Fortran (Linux and Mac)	GNU Fortran
Data options			
Initialize scalar values to zero*	/Qzero	-zero	-finit-local-zero
Place variables in static memory*	/Qsave	-save	-fno-automatic
Allocate variables to the run-time stack	/automatic /auto	-automatic -auto	-fautomatic
Use bytes to specify record length values in unformatted files	/assume:byterecl	-assume byterecl	*(not available)*
Define all default real (and complex) variables as 8 bytes long	/real-size:64	-real-size 64	-fdefault-real-8
Optimization options			
Disable all optimizations (debug mode)	/Od	-O0	-O0
Enable optimizations for speed (release mode)	/O2	-O2	-O2
Enable higher optimizations (may set other options; may not be appropriate for all codes)	/O3	-O3	-O3
Debugging options			
Provide source file traceback information when a severe error occurs at run time	/traceback	-traceback	-fbacktrace
Check array subscripts	/check:bounds	-check bounds	-fcheck=bounds
Fortran Dialect Options			
Produce warning/error for non-standard Fortran 2003 code	/stand:f03	-std03	-std=f2003
Allow free-format code to exceed 132 columns	*(allowed by default)*	*(allowed by default)*	-ffree-line-length-none
Other			
Display compiler version information	/logo	-logo -V	-v --version
Preprocess source files before compilation	/fpp	-fpp	-x f95-cpp-input
Create 32-bit code	*(use appropriate Visual Studio configuration or call appropriate script prior to using compiler from command line)*		-m32
Create 64-bit code			-m64

* not recommended

Appendix L: Instructions for Compiling FAST using IVF for Windows®

The following instructions are provided to give users a general idea on how to compile the FAST code. These instructions have been developed using FAST 7.01.00a-bjj. Please note that names and number of source files may change in different releases; the Compile_FAST.bat script included in the FAST archive will contain the names of all the files required to compile the code.

Before compiling, make sure you have downloaded all of the source files you need to compile. For FAST 7.01.00a-bjj, this includes FAST[*], AeroDyn[†] v13.00.01a-bjj, and NWTC Subroutine Library[‡] v1.04.01.

Compiling FAST Using the Windows Command Line

FAST is currently distributed with a batch file called "Compile_FAST.bat" that will compile the code using IVF. We are aware that a makefile would be a better alternative, but do not currently have the resources to create and support it.

Before using Compile_FAST.bat, you must modify variables in the sections labeled "set compiler internal variables" and "local paths."

Set Compiler Variables

In the "Set Compiler Internal Variables" section, you make sure that the proper paths and environment variables are set for the compiler and linker. The number one reason that people have trouble with the Compile_FAST.bat script is that this step has not been done correctly. The **blue** text shown in Figure 10 (copied from Compile_FAST.bat) must be changed to reflect your compiler.

```
REM ------------------------------------------------------------------------
REM                   set compiler internal variables
REM ------------------------------------------------------------------------
REM    You can run this bat file from the IVF compiler's command prompt (and not
REM    do anything in this section). If you choose not to run from the IVF command
REM    prompt, you must call the compiler's script to set internal variables.
REM    TIP: Right click on the IVF Compiler's Command Prompt shortcut, click
REM    properties, and copy the target (without cmd.exe and/or its switches) here:

CALL "C:\Program Files (x86)\Intel\ComposerXE-2011\bin\ipsxe-comp-vars.bat" ia32 vs2008
```

Figure 10. The "Set Compiler Internal Variables" section of Compile_FAST.bat for FAST v7.01.00a-bjj. Text in blue must be changed by the user before running the script.

One way to find this command is to open the shortcut to the IVF command prompt (also called IVF Build Environment in some versions). You can usually find the shortcut at a location named something like **Start > All Programs > Intel** *CompilerName* **>** *CommandPromptName*. (Different versions of the compiler may have more submenus.) Right click on the shortcut and click "Properties." (See Figure 11 for an example.) A window similar to Figure 12 will open.

[*] http://wind.nrel.gov/designcodes/simulators/fast/
[†] http://wind.nrel.gov/designcodes/simulators/aerodyn/
[‡] http://wind.nrel.gov/designcodes/miscellaneous/nwtc_subs/

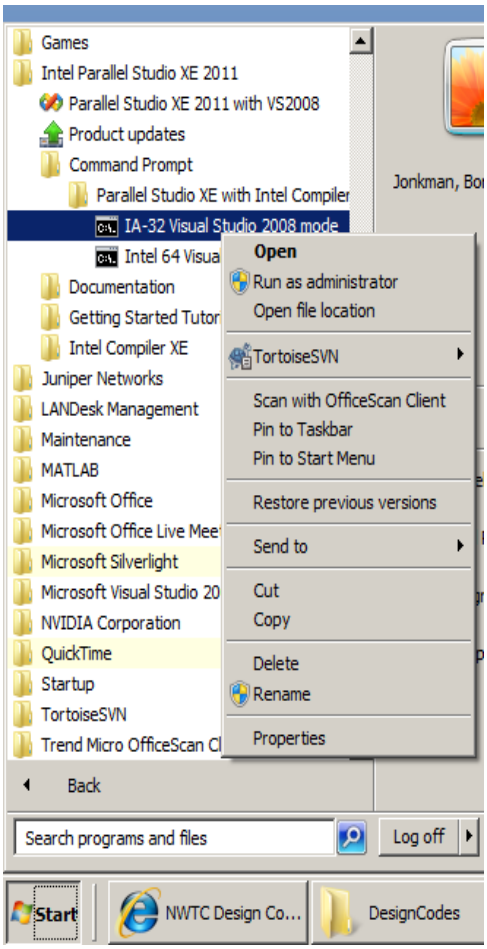


Figure 11. An example of finding the IVF command prompt shortcut

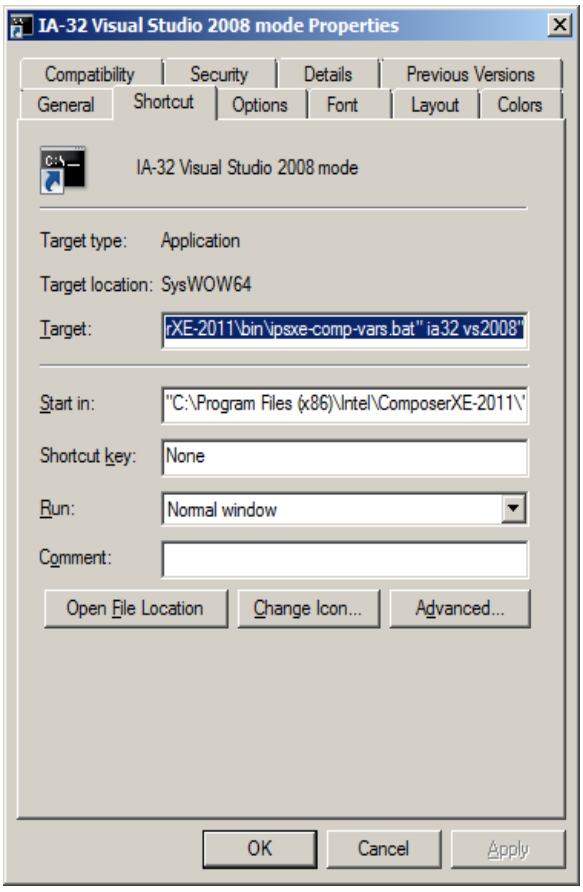


Figure 12. The properties window for an IVF command prompt shortcut

Copy the text from the Shortcut's "Target" field and paste it in the Compile_FAST.bat script:

```
C:\Windows\SysWOW64\cmd.exe /E:ON /V:ON /K ""C:\Program Files
(x86)\Intel\ComposerXE-2011\bin\ipsxe-comp-vars.bat" ia32 vs2008"
```

You will need to remove the call to cmd.exe and its switches, leaving you with the name of a batch file (and possibly some of its arguments):

```
"C:\Program Files (x86)\Intel\ComposerXE-2011\bin\ipsxe-comp-vars.bat" ia32 vs2008
```

If you do not want to call this batch file from Compile_FAST.bat, you may remove the line from the file. However, you must then run Compile_FAST.bat *only* from the compiler's command line window. Please refer to your compiler's documentation about using **ifort** and calling it from the command line.

Set Local Paths

The second section that must be modified in Compile_FAST.bat is labeled "Local Paths." This section needs to be updated with the paths to the source files you are trying to compile. The text that must be updated is shown in **blue** in Figure 13.

```
REM ------------------------------------------------------------------------------
REM ------------------------ LOCAL PATHS -----------------------------------------
REM ------------------------------------------------------------------------------
REM -- USERS WILL NEED TO EDIT THESE PATHS TO POINT TO FOLDERS ON THEIR LOCAL --
REM -- MACHINES.  NOTE: do not use quotation marks around the path names!!!! ---
REM ------------------------------------------------------------------------------
REM NWTC_Lib_Loc is the location of the NWTC subroutine library files
REM AeroDyn_Loc  is the location of the AeroDyn source files
REM Wind_Loc     is the location of the AeroDyn wind inflow source files
REM FAST_LOC     is the location of the FAST source files
REM ------------------------------------------------------------------------------

SET NWTC_Lib_Loc=C:\Users\bjonkman\Data\DesignCodes\NWTC Library\source
SET AeroDyn_Loc=C:\Users\bjonkman\Data\DesignCodes\AeroDyn\Source
SET Wind_Loc=C:\Users\bjonkman\Data\DesignCodes\AeroDyn\Source\InflowWind\Source
SET FAST_Loc=C:\Users\bjonkman\Data\DesignCodes\FAST\Source
```

Figure 13. The "Local Paths" section of Compile_FAST.bat for FAST v7.01.00a-bjj. Text in blue must be changed by the user before running the script.

Run the Script

After you have modified Compile_FAST.bat, you can run it from the command line by typing

```
Compile_FAST.bat
```

in the directory where the batch file is stored. Figure 14 shows the screen output after a successful build using Intel® Composer XE 2011. Notice the title of the command window after Compile_FAST.bat has been run. The script that you call in the "Set Compiler Internal Variables" section to set the paths and environment variables for the IVF compiler also modifies the title of the window. If the title does not say anything about the compiler, please verify that you have modified Compile_FAST.bat correctly.

Creating FAST with the User-Defined Control Options for Interfacing with GH Bladed-style DLLs

If you would like to compile FAST with the user-defined control options that include the interface to GH Bladed-style DLLs, you can use this script to do so. After modifying Compile_FAST.bat as outlined above, you must also copy the FAST source files named UserSubs.f90 and UserVSCont_KP and rename them UserSubs_forBladedDLL.f90 and UserVSCont_KP_forBladedDLL.f90 respectively. In UserSubs_forBladedDLL.f90, you must comment out subroutines UserHSSBr and UserYawCont, and in UserVSCont_KP_forBladedDLL.f90, you must comment out subroutine UserVSCont. To compile, you type

```
Compile_FAST.bat dll
```

at the command line. The executable file that is created will end with `_DLL.exe`.

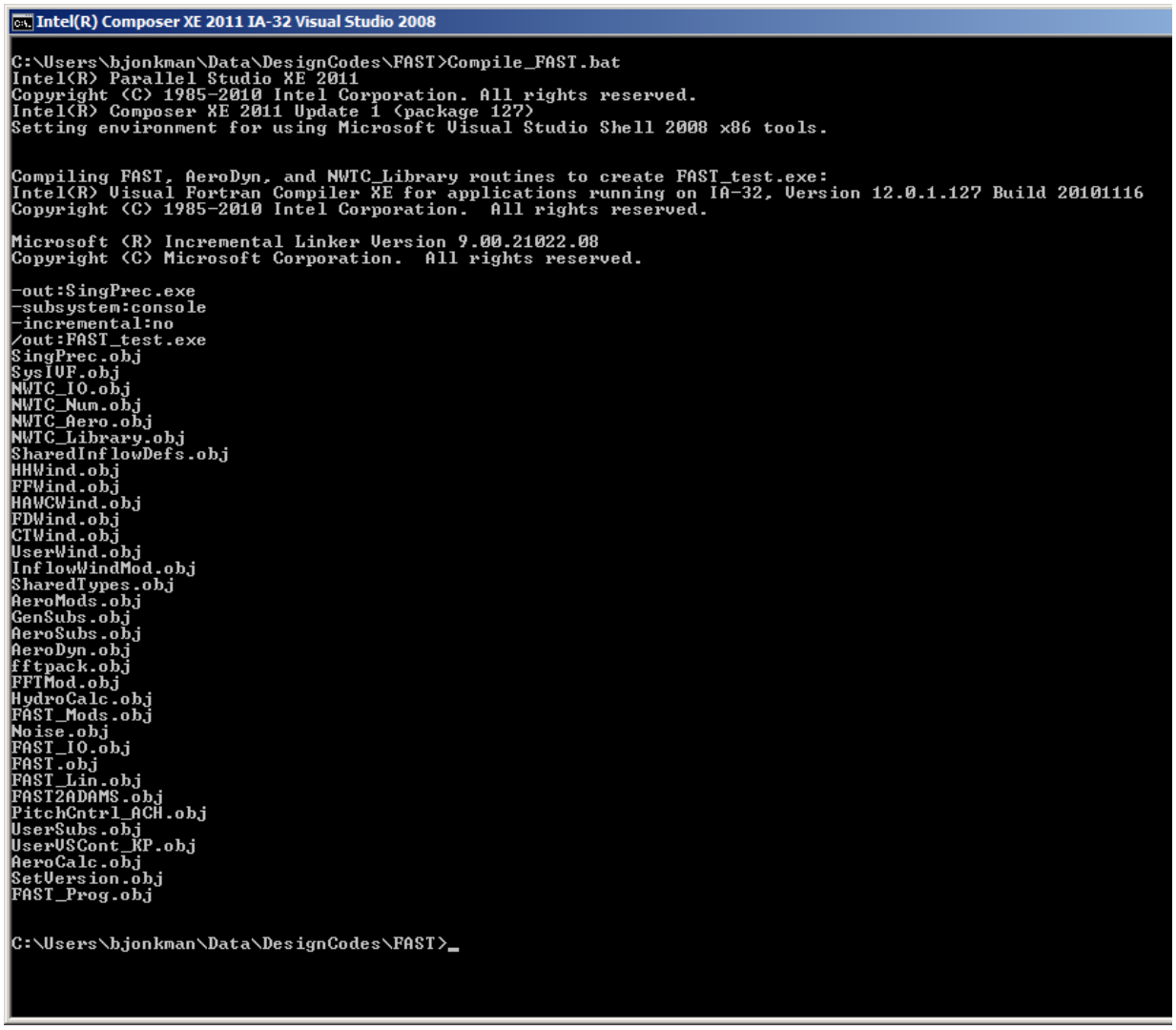


Figure 14. The command prompt window after Compile_FAST.bat has been run.

Compiling FAST Using Microsoft Visual Studio

Microsoft® Visual Studio is an integrated development environment (IDE) that provides useful features for editing source code, compiling, and debugging. The following instructions for compiling FAST have been developed using Microsoft Visual Studio 2008; the steps will be similar for other versions. Note that the size and location of some of the windows in Visual Studio may also vary based on your configuration settings.

Open a New Project

Open Visual Studio and create a new project of type **Intel® Visual Fortran > Console Application**. Choose the **Empty Project** template, and select a name and location for the project before clicking **OK**. See Figure 15.

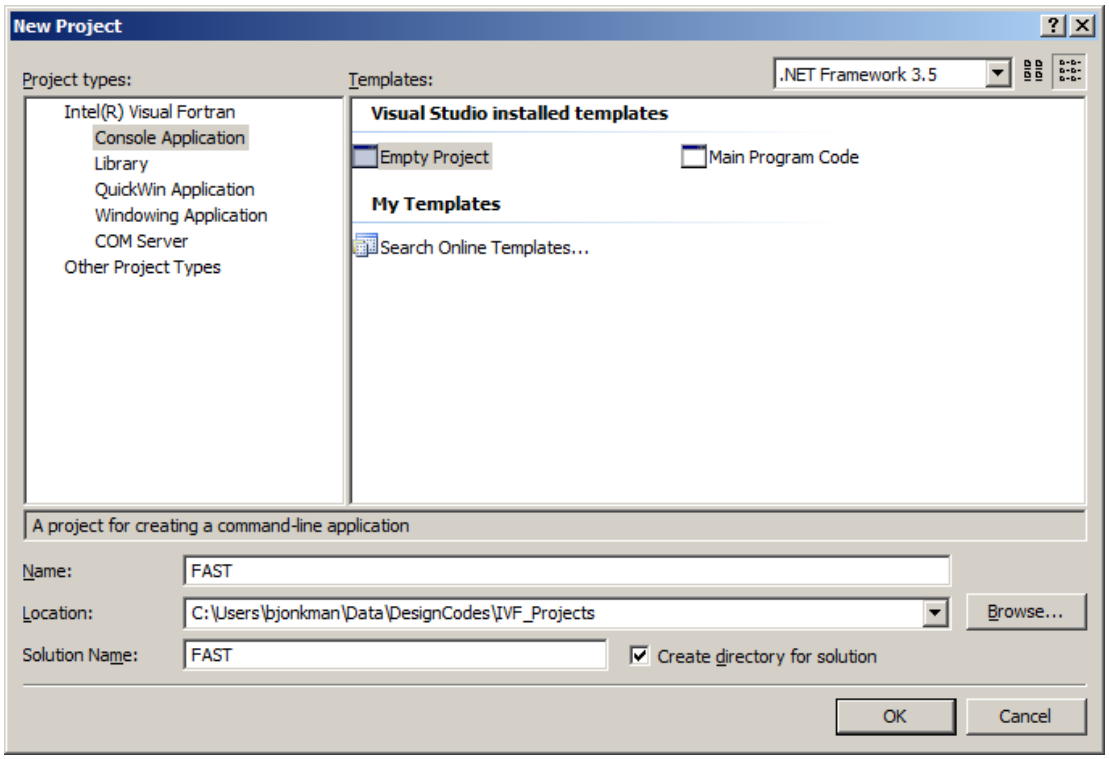


Figure 15. The New Project window in Visual Studio

Add Source Files

Next, you will need to add the source files for FAST and the other codes (components or modules) it uses. To keep the source files organized, you can create folders for each of the components. In the Solution Explorer window, right click on the **Source Files** folder under **<project name>**. Choose **Add > New Folder**, and a new folder named "NewFolder1" will be added under the **Source Files** folder. (See Figure 16.) Rename the folder something descriptive, such as "FAST" for the FAST source files.

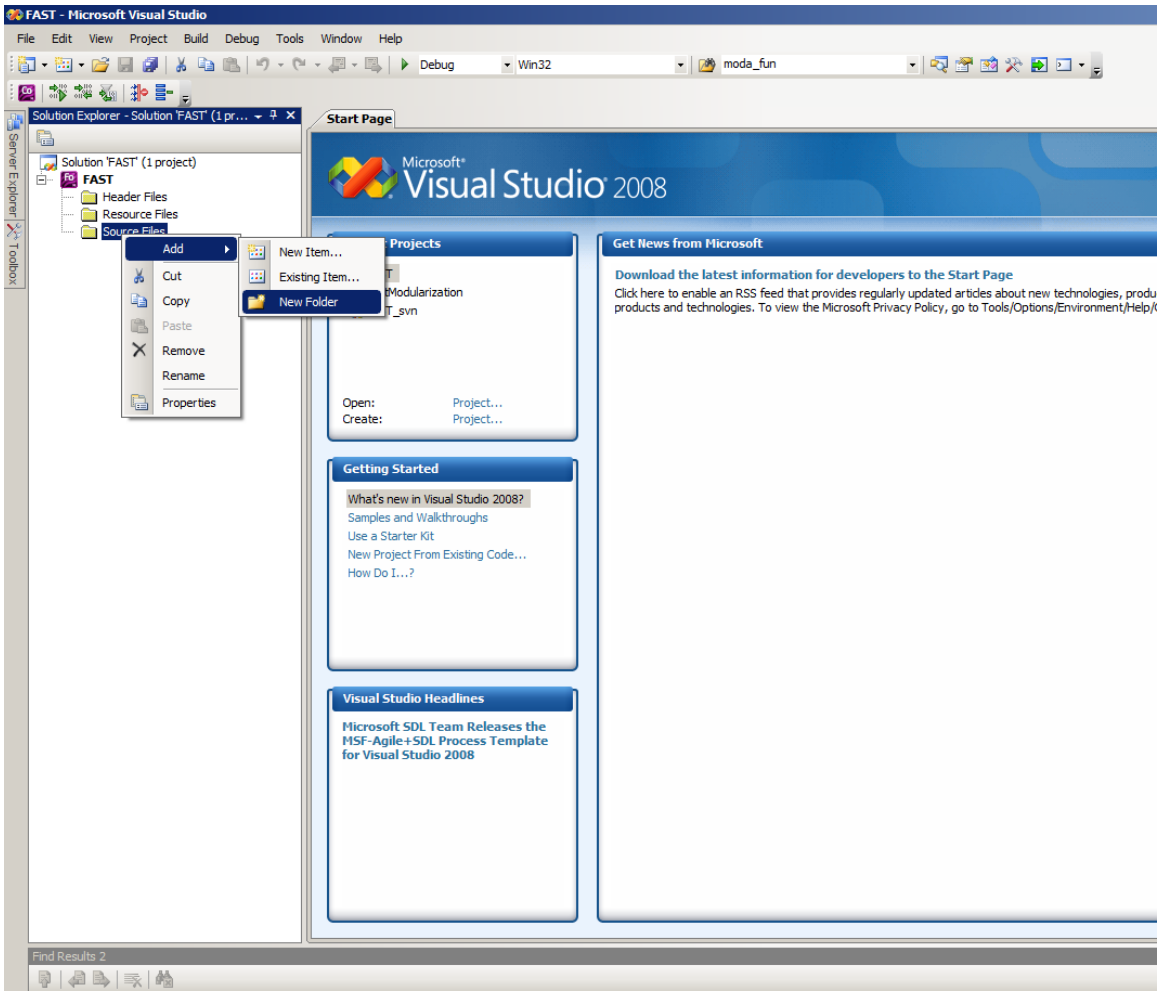


Figure 16. Adding new folders for source files in Microsoft Visual Studio 2008. Note that the Soultion Explorer **window may be located in another part of the screen**

To add the source files for the component, right-click on the folder you have just added and choose **Add > Existing Item…** (Figure 17).

In the **Add Existing Item** window that opens, you must navigate to the folder containing the source files for that component (FAST in this case). Select all of the source files that you need from the directory.

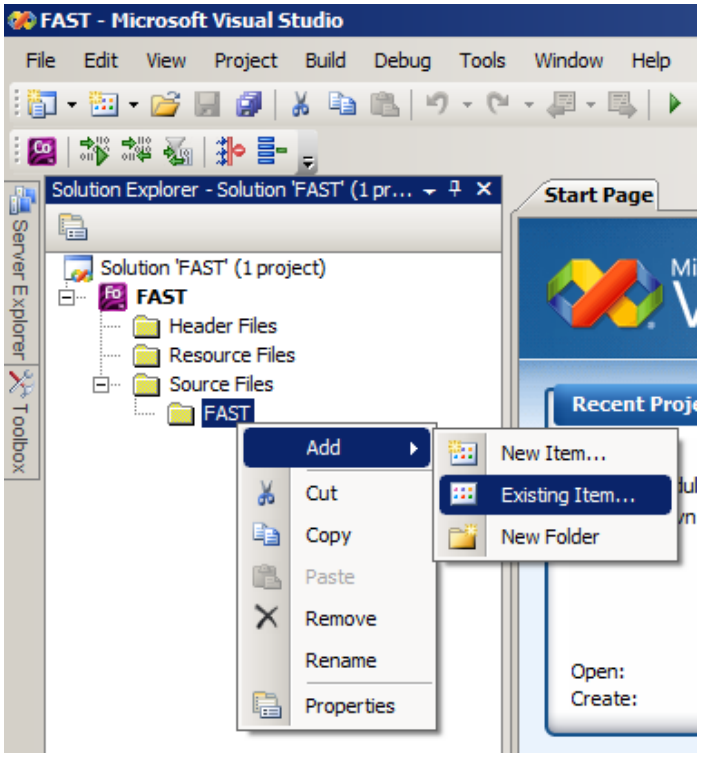


Figure 17. Adding existing source files to a Visual Studio 2008 project.

Create new folders under **Source Files** for each of the other components and add their source files to your project. When you are finished, your **Solution Explorer** window will look something like Figure 18. (Note that if you choose to use the user-defined control options for interfacing with a GH Bladed-style DLL in FAST v7.01.00a-bjj, your list of FAST source files will include BladedDLLInterface.f90, UserSubs_forBladedDLL.f90, and UserVSCont_KP_forBladedDLL.f90. It will *not* include PitchCntrl_ACH.f90, UserSubs.f90, or UserVSCont_KP.f90. The file UserSubs_forBladedDLL.f90 is a copy of UserSubs.f90 with subroutines UserHSSBr and UserYawCont commented out. The file UserVSCont_KP_forBladedDLL.f90 is a copy of UserVSCont_KP with subroutine UserVSCont commented out.)

Set Compiler Options

Next, you must set the compiling options for the project. On the main menu, select **Project > <project name> Properties**. (Note: if you select an individual source file and then click on **Project > Properties**, you will change the properties for *only* that source file. Make sure you are changing properties for the *entire* project here.)

A **<project name> Property Pages** window will open. See Figure 19. In the **Configuration** dropdown box at the top of the window, select "All Configurations" to set the compiler options for all configurations.

Select **Configuration Properties > Fortran > Data** in the box on the left. In the window on the right, select **Use Bytes as RECL = Unit for Unformatted Files** and choose **Yes (/assume:byterecl)**. (This option is required for InflowWind's coherent structures). Because FAST v7.01.00a-bjj still has some unresolved issues relating to local variables and their initialization, you will also need to set **Local Variable Storage** to **All Variables SAVE /Qsave**, and set **Initialize Local Saved Scalars to Zero** to **Yes (/Qzero)**. (Note that once we find and fix these issues, we will no longer compile with **/Qsave** and **/Qzero**.) Click **OK** to save your changes and close the window. The window in Figure 19 shows the project properties with the compiling options changed as necessary.

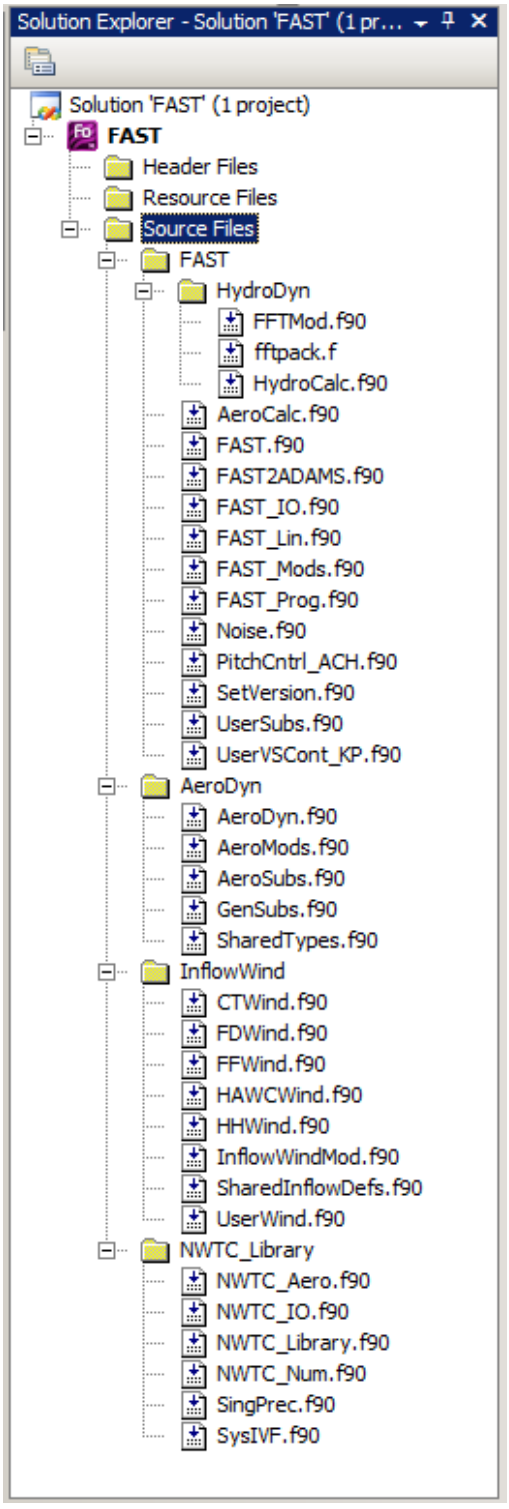


Figure 18. The source files for FAST v7.01.00a-bjj listed in the Visual Studio Solution Explorer **window**

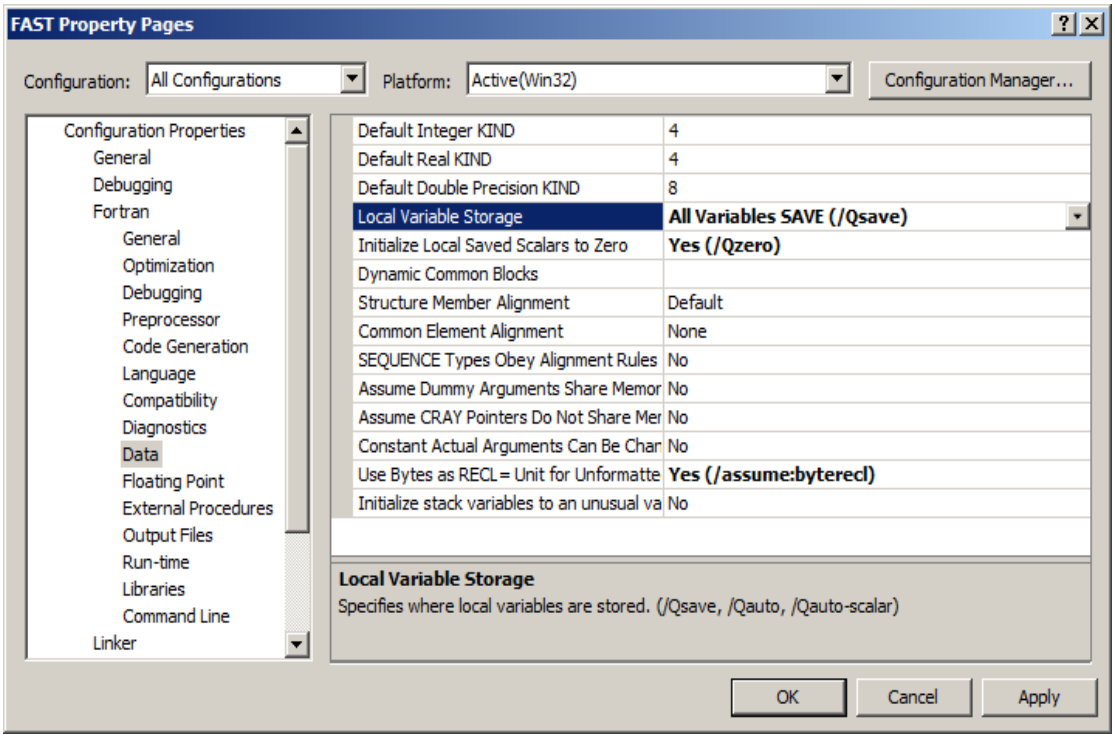


Figure 19. The <project name> Property Pages **window in Visual Studio with the compiling options changed for FAST v7.01.00a-bjj**

Build the Project

To build your project (create an executable file), you can choose from "Debug" or "Release" mode (or create your own settings). Debug mode is set to disable optimizations and generate debugging information. It will compile faster and run slower than Release mode. Release mode is set to optimize the code for speed. It does not generate debugging information. You can choose the desired configuration mode by clicking on the dropdown box on the toolbar or choose **Build > Configuration Manager** from the main menu.

After you have picked the appropriate configuration, choose **Build > Build Solution** from the main menu. The **Output** window will show your progress. (See Figure 20.) Any errors encountered during the build process will also appear there.

This step creates an executable file, named by default <project location>\<configuration name>\<project name>.exe. If you want to change this file's name, you can do so in the **<project name> Property Pages** window. (Select **Project > <project name> Properties** from the main menu.) The name of the executable is defined under **Configuration Properties > Linker > General > Output File**. See Figure 21.

Note that *$(OutDir)* in Figure 21 refers to the **Output Directory** defined under **Configuration Properties > General** in the same window. (Remember that any changes you make apply to only the configuration that is selected at the top of this window.)

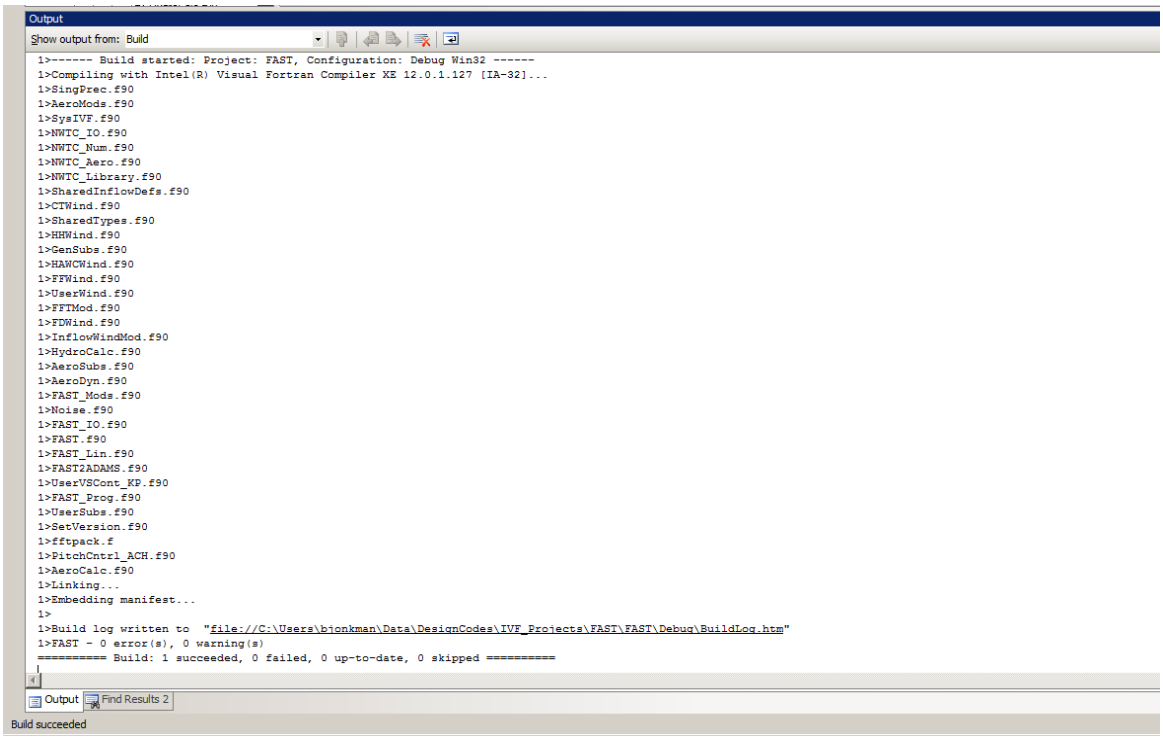


Figure 20. The Output **window in Visual Studio after building FAST using the Debug configuration.**

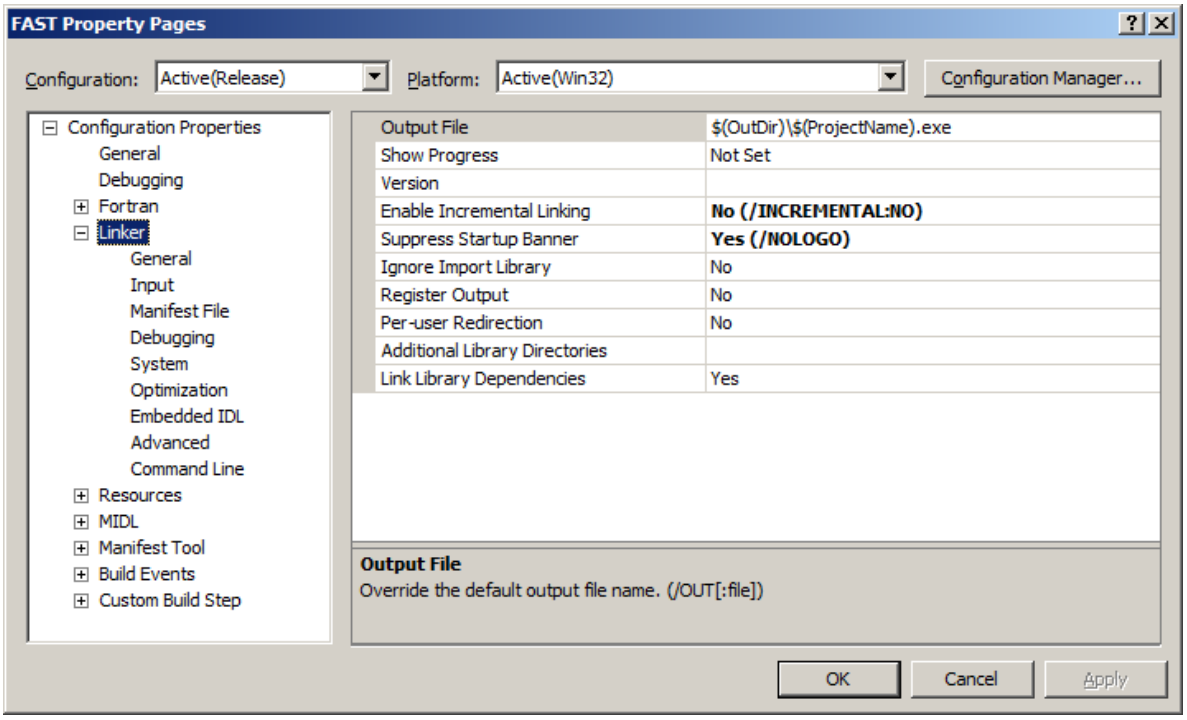


Figure 21. The <project name> Property Pages **window in Visual Studio, showing the location of the Output File.**