NREL
NATIONAL RENEWABLE ENERGY LABORATORY

# Overview of the NWTC Programmer's Handbook

## NREL/DOE Workshop on the New Modularization Framework for the FAST Wind Turbine CAE Tool

Bonnie Jonkman and John Michalakes

October 8, 2012

# Presentation Outline

- **Purpose and Motivation**
- **General Information on NWTC CAE Tools**
  - Copyright and Licensing
  - Version Control
  - NWTC Policy for Working with Subversion
  - Version Naming
  - Software Distribution
- **Steps for Software Development**
  - Planning Code Development
  - Writing Source Code
  - Testing
  - Documenting
- **Implementing the FAST Modular Framework**
  - Module Structure
  - FAST Registry
  - Developer-Provided Subroutines
  - Meshes
  - Units and Coordinate Systems
  - Coupling Modules Together
  - Handling Errors
  - Handling I/O
  - Inter-Language Interfaces

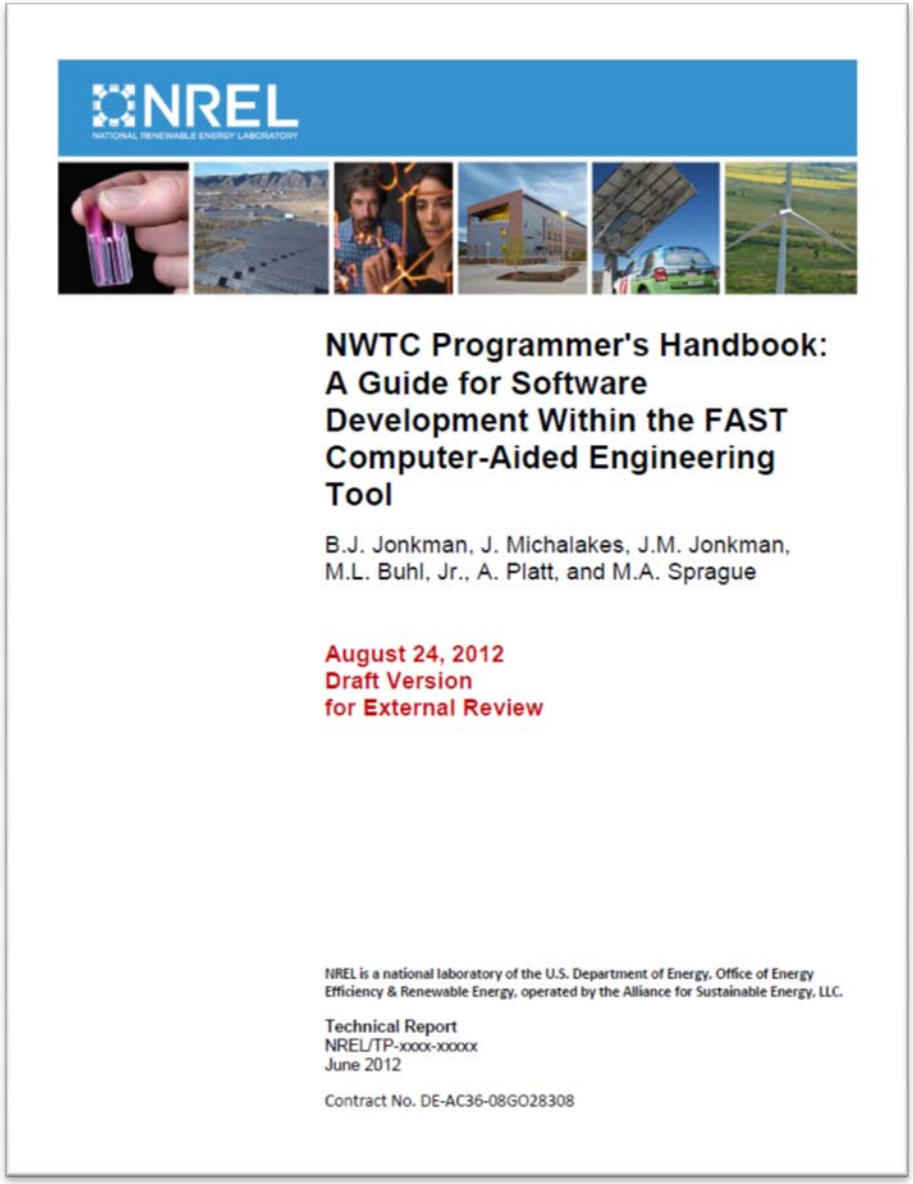
# Purpose and Motivation

## Purpose

- **Provide a single reference with policies, expectations, and guidance for code developers**
  - New developers can benefit from informal guidelines already used at the NWTC (years of experience)
  - Most applies to all NWTC CAE tools
  - There is a difference between developing codes for individual/group research needs and codes that thousands of people will use
- **Provide guidance for implementing the new FAST modularization framework**
  - Distributed with a template module in the new FAST modularization framework

## Motivation

- **The number of CAE-tool developers (and users) has significantly increased**
- **Many new developments are being done simultaneously (e.g., FOAs)**


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

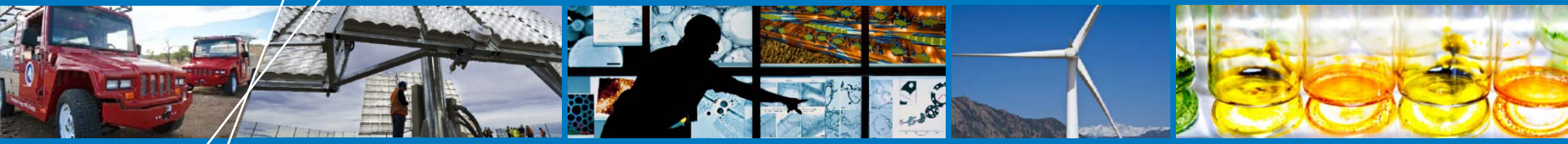
August 24, 2012
Draft Version
for External Review

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
June 2012

Contract No. DE-AC36-08GO28308

NREL
NATIONAL RENEWABLE ENERGY LABORATORY



# General Information on NWTC CAE Tools

# Copyright and Licensing

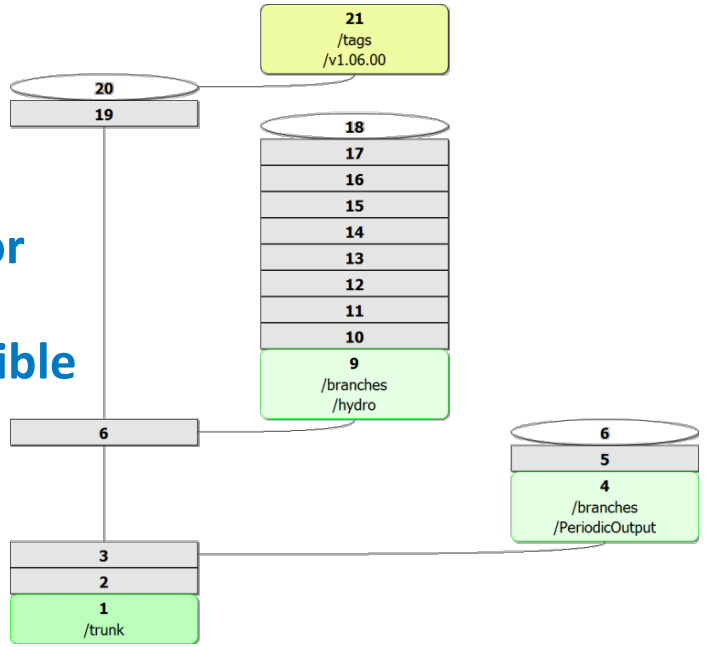
- **Historically, NWTC CAE tools have been distributed under the Data Use Disclaimer Agreement found on our web site**
- **New dual licensing arrangement**
  - Details are still being discussed by our legal team
  - NWTC CAE tools will be released under the GNU General Public License (GPL) v3.0 open-source license
  - Codes may also be offered under other less restrictive open-source licenses on a per-case basis
    - Will allow NREL software to be distributed in commercial codes (or other uses), when (non-NREL) GPL routines are removed
- **All source code files must have copyright and license agreement information listed at the beginning of the file**

# Version Control

- **Subversion (SVN) is an open source automatic software version control and management system**
- **What it does:**
  - Provides developers at NREL and at remote locations access to source code in a centrally managed and backed-up repository
  - Maintains a history of the code, allowing retrieval of earlier versions of the code with change-logs and differences
  - Supports multiple developers working on the code, automatically detecting and reporting potentially conflicting updates
  - Provides notification and an audit trail of code changes so that there's a record of every change and who made it
- **More information**
  - FAST Programmer's Handbook
  - The SVN manual, with tutorials: http://svnbook.red-bean.com
  - Windows Subversion client: http://tortoisesvn.net

# NWTC Software Development Policy

*for working with Subversion*




- The Subversion 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.
- Branches must pass all certification tests prior to being merged back into the trunk.
- The tool's primary owner is responsible for reviewing changes from the branches and merging approved branches back into the trunk.
- Certification tests must be updated to test new features and to test issues that have been found while debugging code.
- When a tool is released, a copy of the trunk is tagged using its NWTC version number.

# Version Naming

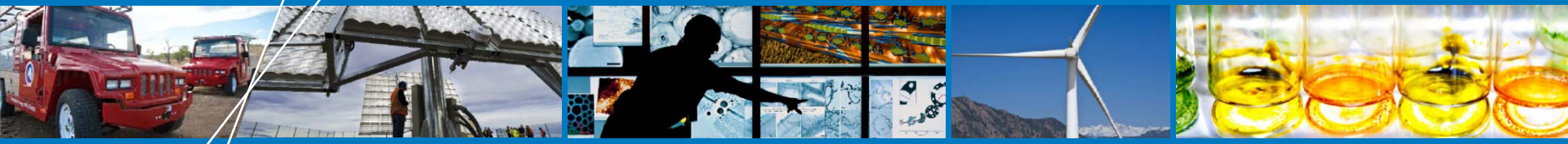
- **Each program and module (component) should have a version number**
- **The form is "v0.00.00a-bjj"**
  - The number left of the first decimal is for major updates.
  - The number between decimals increases for input file changes and new features
  - The number to the right of the last decimal is for minor changes and bug fixes
  - The alpha revision number is present only for alpha versions (not beta)
  - The programmer's initials are present only for alpha versions (not beta)
- **The first thing you change in the source code should be the version number**
- **If possible, display the version number when the program runs and include it in output files**

# Software Distribution

**Files typically distributed in an archive on our web site when software is officially released:**

- All of the source files, including a driver program
- Executable code or library (if applicable) for Windows®
- Name(s) and version(s) of all other codes the software uses and the compiler used to generate the executable code
- A file that indicates the order of compilation for the source files
- A change log that contains the full history of previously released versions
- A user's guide and theory manual
- Sample input files and test cases
  - Output generated from the sample cases
  - A script or program to compare a user's results from the sample test cases with the output included in the software distribution
- Any other files that are useful to run, understand, or maintain the software

NREL
NATIONAL RENEWABLE ENERGY LABORATORY



# Steps for Software Development

# Steps: Planning Code Development

- **Read the Programmer's Handbook**
- **Write your plan for development in a document before you write any source code**
  - Write all equations for FAST modules in the form required for the modularization framework
    - Identify algorithms being used
    - For loose coupling, identify the time-domain integration scheme
  - Identify potential numerical problems and suitable solutions
  - Categorize your data
    - What inputs are needed from other modules? (Are they available?)
    - What outputs will be needed by other modules?
    - Are there states and parameters?
  - Write the input-output transformation equations for FAST modules
  - Create a sample input file (if applicable)
- **Discuss the plan with the primary owner and any other developers it may affect**

***This step may be the most time-consuming part of the development process.***

# Steps: General Guidelines for Source Code

- NWTC CAE tools should be able to link with codes compiled in Fortran 2003 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.
- Modules written for the FAST framework must adhere to the requirements of the template provided.
- Multiple instances of modules must be allowed to exist simultaneously (dynamically allocatable). No “global” data.

# Steps: Writing Source Code

- **Use Fortran, C, or C++**
  - Adhere to the standard for the language being used (Fortran should use f03 [Fortran 2003])
  - If you must use nonstandard code, isolate it in a separate source file
  - FAST glue code is written in Fortran
  - C and C++ developers of FAST modules will need to use the Fortran template to call their C/C++ routines (discussed later)
- **Use the NWTC Subroutine Library when possible**
  - Variable KIND definitions
  - Mathematical constants (π)
  - Writing to the screen
  - Opening files
  - Reading from input files
- **Use the guidelines listed in the Programming Handbook**

# Steps: Testing

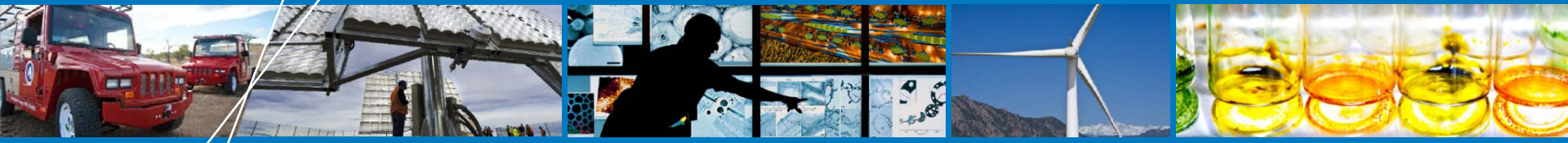
- **Verification**
  - Compare results to hand calculations, results of other software, or other known solutions
- **Validation**
  - Comparing results to test data can be useful to show that a theory is valid
  - Not a substitute for verification
- **Version checking**
  - Make sure that new features do not harm existing capabilities
- **Testing new features**
  - When possible, individual parts of a code should be tested separately before integrating them into a much larger code
  - Create new tests for future version checking

# Steps: Documentation

- **Comments in the source code**
  - Help developers understand what it is intended to do and how it works
  - Include references to additional documentation if necessary
- **Change logs**
  - Provide history of changes and reasons for them
  - Help users see what functionality has changed
- **Developer logs (Commit logs in Subversion)**
  - Help developers understand details of what has changed (and the reasons for them)
- **User's guides**
  - Help people understand how to use the software
- **Theory manuals**
  - Document and explain equations and algorithms implemented in the code
- **Sample input files and test cases**
  - Help users learn how to use the software (along with user's guides)
  - Explain where input values came from (are there simplifications?)

***Software is much easier to understand and maintain when it is well documented.***

NREL
NATIONAL RENEWABLE ENERGY LABORATORY



# Implementing the FAST Modular Framework

# Data Structures for the FAST Framework

- **All data must be separated into these categories:**
  - System input
  - System output
  - System states
    - Continuous
    - Discrete
    - Constraint
    - Other (added for code efficiency, flexibility)
  - System parameters
  - Initialization input
  - Initialization output
  - Variables local to a subroutine (cannot retain their values between subroutine calls)
- **All data types except subroutine-local variables must be defined using the FAST Registry, which will generate the required source code**

| 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 | Optimization/other states |

# Subroutines for the FAST Framework

- Template for the subroutines is provided with the Handbook
- Modules may have other internal (private) subroutines or have their own "sub-modules"
- All of the {*TypeName*} subroutines will be auto-generated with FAST Registry

| Template Requirements | Loose | Tight (Time Marching) | Tight (Linearization) |
|---|---|---|---|
| Initialize/End Subroutines | | | |
| • ModName_Init | ✓ | ✓ | ✓ |
| • ModName_End | ✓ | ✓ | ✓ |
| Time-Stepping Subroutines | | | |
| • ModName_CalcConstrStateResidual | | ✓ | ✓ |
| • ModName_CalcOutput | ✓ | ✓ | ✓ |
| • ModName_UpdateStates | ✓ | | |
| • ModName_CalcContStateDeriv | | ✓ | ✓ |
| • ModName_UpdateDiscState | | ✓ | ✓ |
| Jacobian Subroutines | | | |
| • ModName_JacobianPInput | | ✓ | ✓ |
| • ModName_JacobianPContState | | | ✓ |
| • ModName_JacobianPDiscState | | | ✓ |
| • ModName_JacobianPConstrState | | ✓ | ✓ |
| 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*

# Fortran Module Structure

## MODULE ModuleName_Types

- Contains
  - all data type definitions
  - copy/destroy subroutines
  - pack/unpack subroutines for individual data types ({*TypeName*} subroutines)
- The source code for the module is automatically generated using the FAST Registry

## MODULE ModuleName

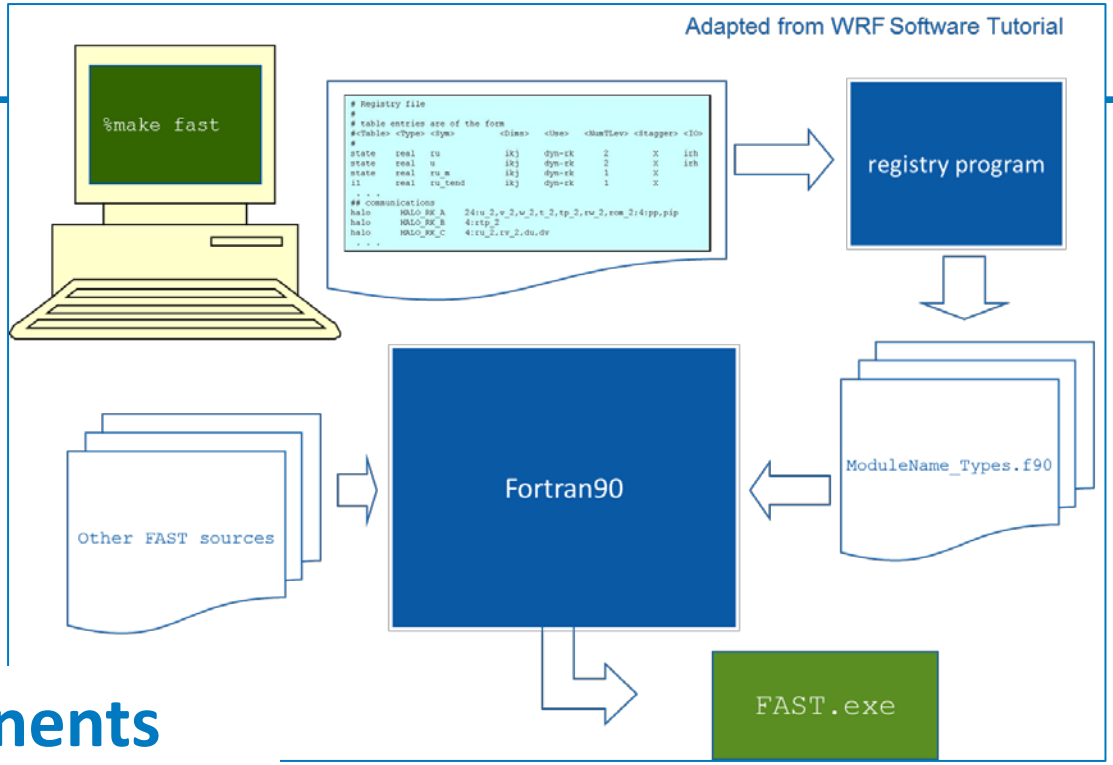
- USEs ModuleName_Types
- Contains
  - initialize/end subroutines
  - time-stepping subroutines
  - Jacobian subroutines
  - pack/unpack subroutines for module (not individual data types)*
- Developers must use the template provided with the handbook to implement this module

* We are looking into the possibility of also generating this with the FAST Registry and including it in the ModuleName_Types MODULE.

# FAST Registry

- **Table-based automatic code generation of large sections of FAST module interface data structures and code**
- **Adapted from NCAR WRF model software framework**
  - "Active" data-dictionary generates ~300K lines of interface, I/O, and parallelization code from concise user-editable table listing data structures and properties
  - Automates time consuming and error-prone programming tasks
  - Changes to Registry propagate to hundreds of places in the code on recompile

# FAST Registry




## FAST Registry components

- Single master text file, or "Registry File"
  - Contains entries, one line per data element
  - Quick reference for data names, types, dimensionality (Data Dictionary)
  - Able to include other Registry files, allowing one Registry file per module; easily extendable
- Registry program
  - Compiled and called to process Registry when FAST framework is compiled
  - Reads Registry and generates a separate source file ModuleName_Types.f90
    - MODULE ModuleName_Types, defining all defined data types required by FAST interface
    - Subroutines to copy, destroy, pack, and unpack types
    - Subroutines to pack and unpack modules
  - Generates wrapper routines for inter-language interfaces to non-Fortran modules

# Registry.txt

```
# Top level FAST Registry file:  Registry.txt

# Incorporate Registry files for individual modules
include Registry-ModuleNameX.txt
include Registry-ModuleNameY.txt
   ...
#  Add more as new modules contributed to FAST
~
```

```
# Part of Registry file: Registry-ModuleNameX.txt
  ...
# ..... Input argument type ..................................................................
# Define inputs that are contained on the mesh here:
typedef  ModuleNameX/ModNmX InputType    MeshType MeshedInput -      - - "An input mesh"
typedef    ^                  ^          ReKi     aScalar     -      - - "Scalar variable"   "units"
typedef    ^                  ^           ^       anArray    {:}     - - "Allocatable array" "units"

# ..... Output argument type .................................................................
typedef    ^                OutputType   MeshType MeshedOutput -      - - "An output mesh"
typedef    ^                 ^           ReKi     bScalar      -      - - "Scalar variable"   "units"
typedef    ^                 ^            ^       bArray      {2}{3} - - "2 by 3 2-D array"  "units"
  ...
```

ditto

Module longname/shortname

Type of field

Name of field

Name of derived data type containing field

Dimensions of field

Description of field

Units of field

Control

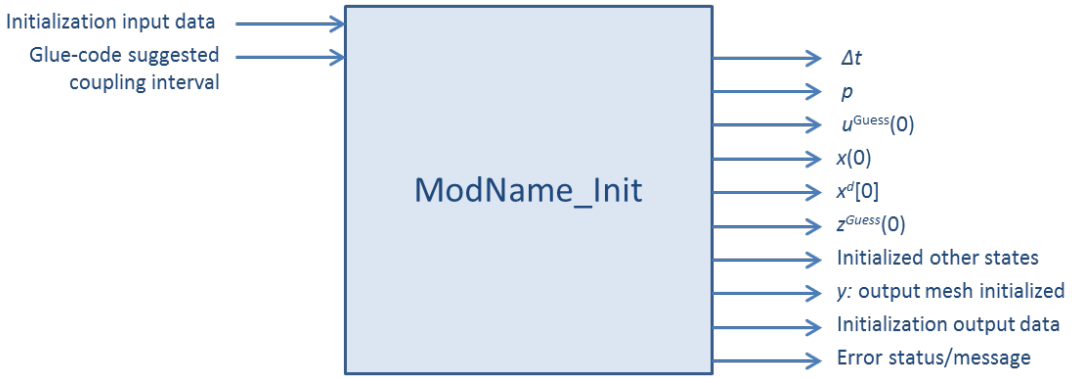
Unused

Keyword (typedef means "define field in a derived data type")

Click here to see generated code

# FAST Framework: Initialization

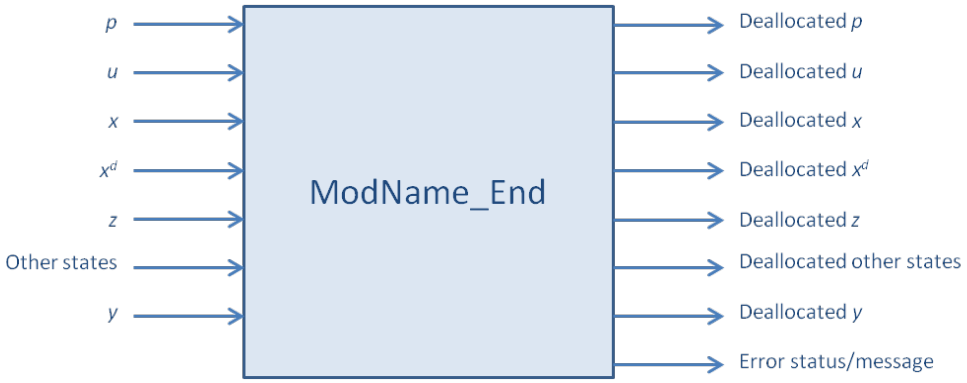
- **Subroutine ModName_Init**
  - Designed to be called one time at the beginning of each simulation for each instance of the module
  - Performs initialization tasks for the module, including
    - Reading input files
    - Defining parameters
    - Setting initial values of states
    - Setting up any meshes
  - Returns the time increment for loose coupling and discrete states
- **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.**



# FAST Framework: End

## Subroutine ModName_End

- Designed to be called one time for each instance of the module at the end of a simulation
- Its main tasks are to release memory and close files.



# FAST Framework: Time-Stepping Subroutines

## Time-stepping routines

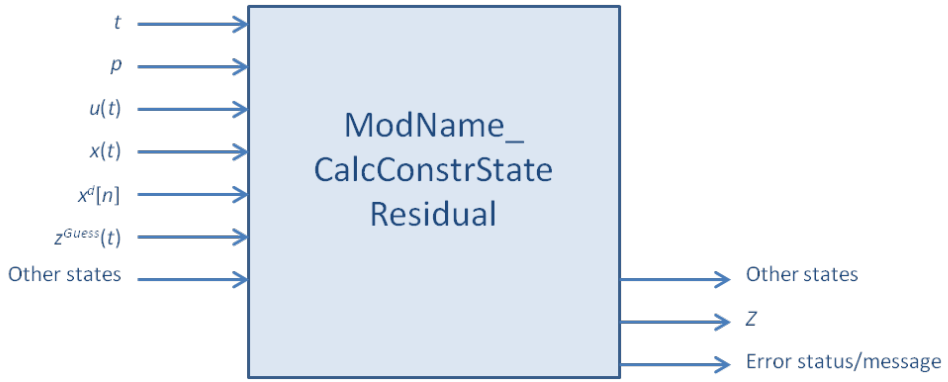
- Purpose is to compute outputs and update the states (continuous, discrete, and constraint)
- Glue code calls different routines based on choice of coupling scheme (tight or loose)
  - Module updates states in loose coupling
  - Glue code updates states in tight coupling
- Current simulation time is input as a double-precision real number (DbKi)

| Template Requirements | Loose | Tight (Time Marching) | Tight (Linearization) |
|---|---|---|---|
| Time-Stepping Subroutines | | | |
| • ModName_CalcConstrStateResidual | | ✓ | ✓ |
| • ModName_CalcOutput | ✓ | ✓ | ✓ |
| • ModName_UpdateStates | ✓ | | |
| • ModName_CalcContStateDeriv | | ✓ | ✓ |
| • ModName_UpdateDiscState | | ✓ | ✓ |

# FAST Framework: Time-Stepping Subroutines

## Subroutine ModName_CalcConstrStateResidual

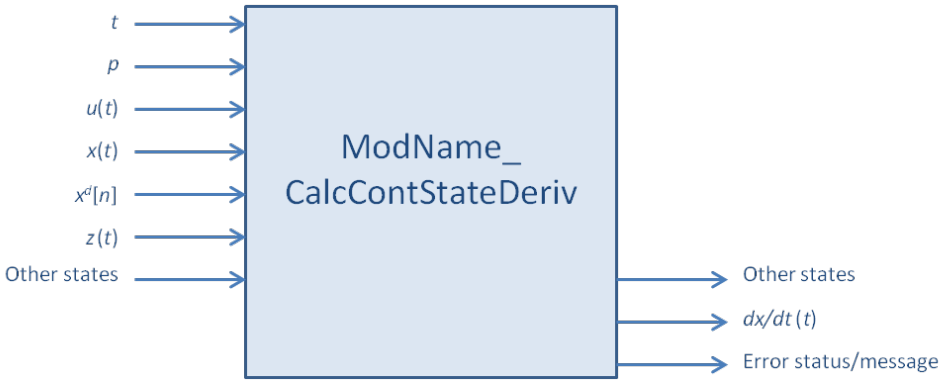
- Solves for the residual of the constraint state equations
- Should return zero when the constraint-state guess is correct
- Called from the glue code in tight coupling schemes



# FAST Framework: Time-Stepping Subroutines

## Subroutine ModName_CalcContStateDeriv

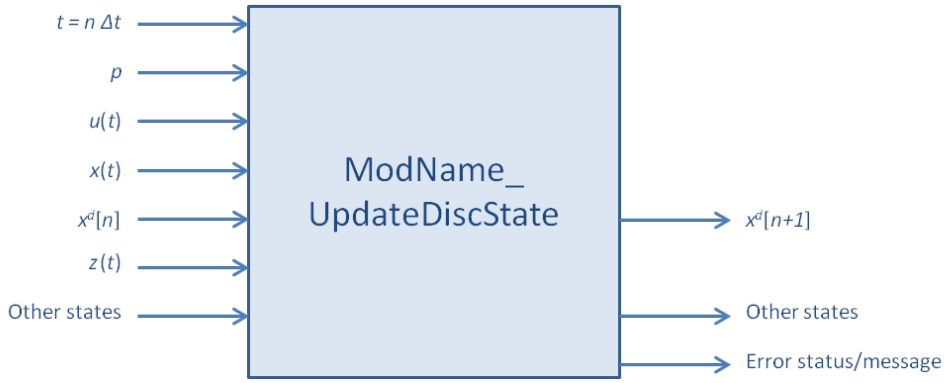
- Calculates the first time derivatives of the continuous states
- Called from the glue code in tight coupling schemes



# FAST Framework: Time-Stepping Subroutines

## Subroutine ModName_UpdateDiscState

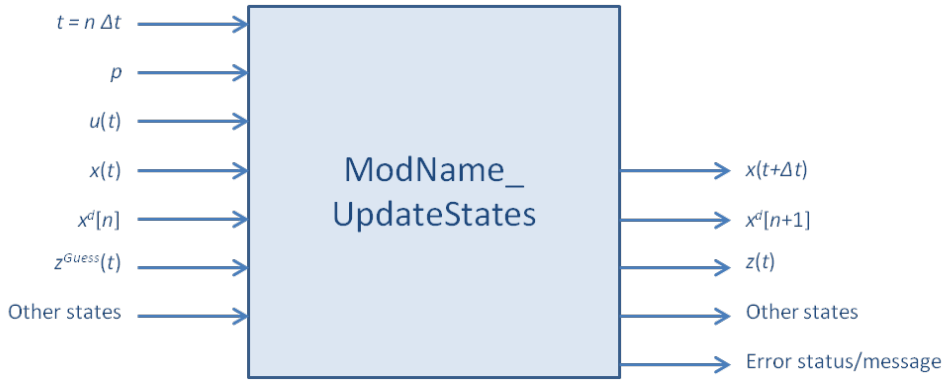
- Updates the discrete states to their values at the next coupling interval
- Called from the glue code in tight coupling schemes
- Called at the coupling interval defined in the module initialization



# FAST Framework: Time-Stepping Subroutines

## Subroutine ModName_UpdateStates

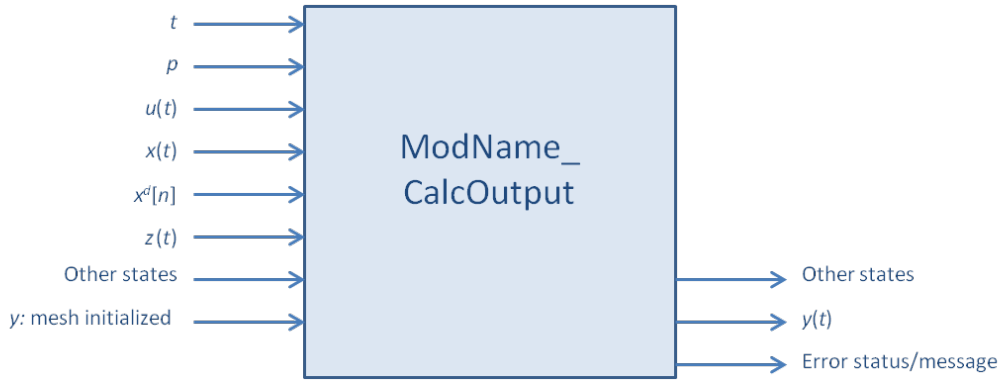
- Solves for the constraint states at the current simulation time
- Updates the continuous and discrete states to their values at the next coupling interval
- Called from the glue code in a loose coupling scheme
- Called at the coupling interval defined in the module initialization
- If the module is able to be tightly coupled, this routine can call the tight coupling routines: ModName_CalcConstrStateResidual, ModName_CalcContStateDeriv, and ModName_UpdateDiscState



# FAST Framework: Time-Stepping Subroutines

## Subroutine ModName_CalcOutput

- Computes the system outputs at the current simulation time
- Called from the glue code in tight *and* loose coupling schemes



# FAST Framework: Jacobian Subroutines

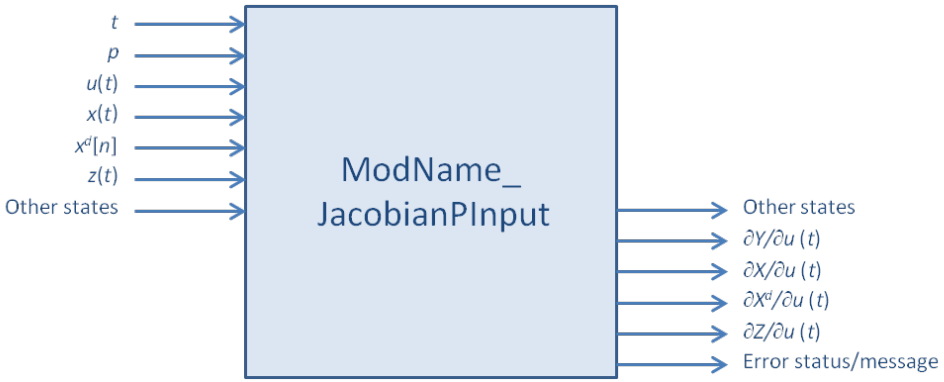
| Template Requirements | Loose | Tight (Time Marching) | Tight (Linearization) |
|---|---|---|---|
| Jacobian Subroutines | | | |
| • ModName_JacobianPInput | | ✓ | ✓ |
| • ModName_JacobianPContState | | | ✓ |
| • ModName_JacobianPDiscState | | | ✓ |
| • ModName_JacobianPConstrState | | ✓ | ✓ |

- **Jacobian Subroutines**
  - Each subroutine computes four partial derivatives (optional arguments)
  - Called by the glue code in tight coupling schemes
    - Four partial derivatives required for tightly coupled time marching schemes: $\partial Z/\partial z$, $\partial Z/\partial u$, $\partial Y/\partial z$, and $\partial Y/\partial u$
    - All 16 partial derivatives are required for linearization (we recommend defining all so the module can be linearized)
- **Implementation of Jacobians**
  - Analytical derivation and implementation gives best numerical convergence performance
  - Numerical implementations are acceptable

# FAST Framework: Jacobian Subroutines

## Subroutine ModName_JacobianPInput

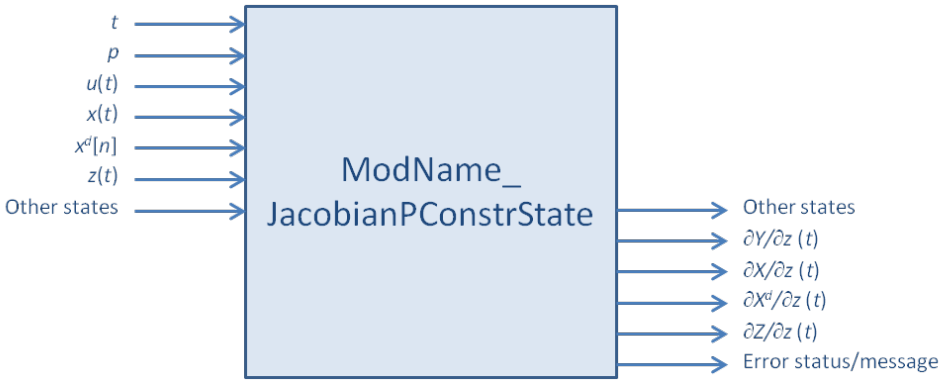
- Calculates four Jacobians: the output, continuous-state, constraint-state, and discrete-state equations with respect to the system inputs
- Called by the glue code in tight coupling schemes
- Jacobians are optional arguments
  - Only $\partial Z/\partial u$ and $\partial Y/\partial u$ are required for tightly coupled time marching simulation
  - All four are required for linearization



# FAST Framework: Jacobian Subroutines

## Subroutine ModName_JacobianPConstrState

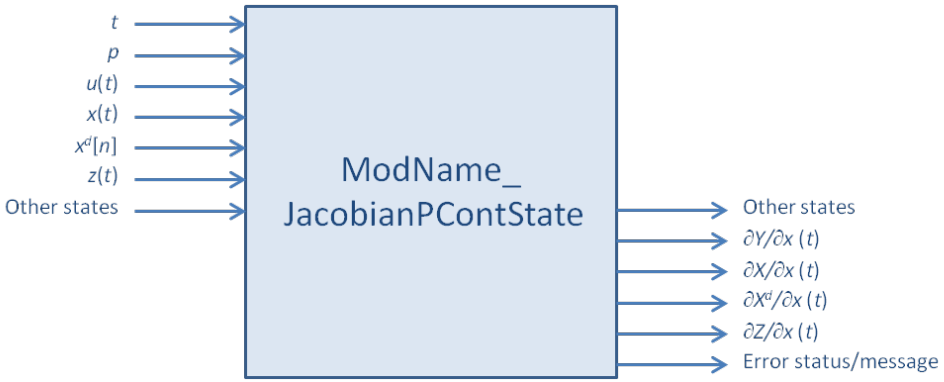
- Calculates four Jacobians: the output, continuous-state, constraint-state, and discrete-state equations with respect to the constraint states
- Called by the glue code in tight coupling schemes
- Jacobians are optional arguments
  - Only $\partial Z/\partial z$ and $\partial Y/\partial z$ are required for tightly coupled time marching simulation
  - All four are required for linearization



# FAST Framework: Jacobian Subroutines

## Subroutine ModName_JacobianPContState

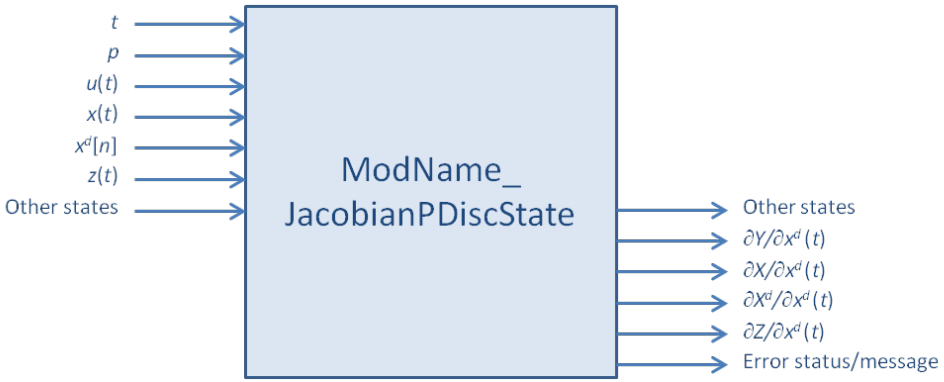
- Calculates four Jacobians: the output, continuous-state, constraint-state, and discrete-state equations with respect to the continuous states
- Used only for linearization



# FAST Framework: Jacobian Subroutines

## Subroutine ModName_JacobianPDiscState

- Calculates four Jacobians: the output, continuous-state, constraint-state, and discrete-state equations with respect to the discrete states
- Used only for linearization



# FAST Framework: Pack/Unpack Subroutines

- **FAST Framework specifies that each module should provide a pack and unpack routine for each derived data type (Table 2, Programmer's Handbook) and for the module type as a whole**
  - Put data in a form easily used for vector/matrix operations (single data type)
  - Put data in a form that can be read/written to restart files
- **Arguments**
  - Type to be packed/unpacked
  - Pointers to 3 buffers: 1-D arrays of real, double, and integer
- **Packing moves data type→buffers, unpacking reverses**
- **Pack routines ALLOCATE the memory for the buffers; it is up to caller to DEALLOCATE buffers (unpack routines do not deallocate)**

| Template Requirements | Loose | Tight (Time Marching) | Tight (Linearization) |
|---|---|---|---|
| Pack/Unpack Subroutines | | | |
| • ModName_Pack | ✓ | ✓ | ✓ |
| • ModName_Pack{*TypeName*•} | ✓ | ✓ | ✓ |
| • ModName_Unpack | ✓ | ✓ | ✓ |
| • ModName_Unpack{*TypeName*•} | ✓ | ✓ | ✓ |

# FAST Framework: Meshes

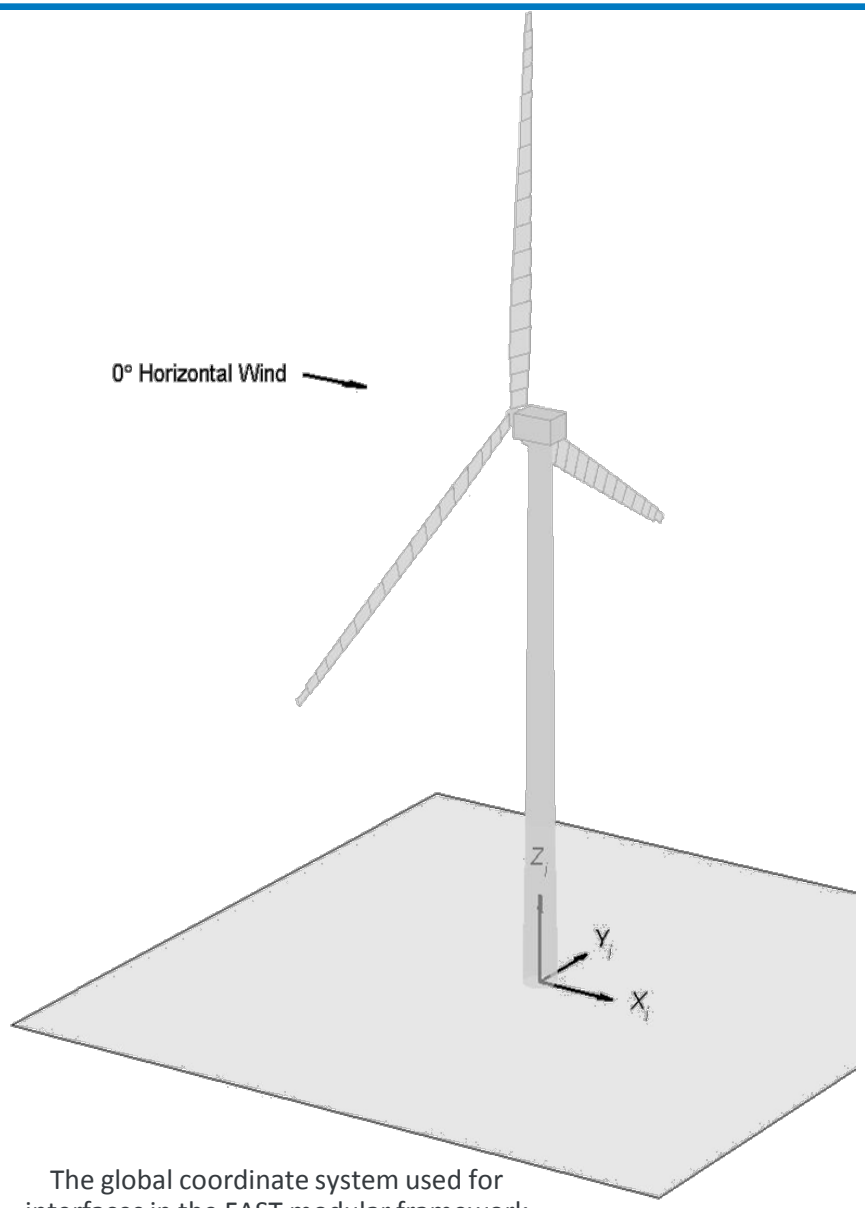
- **FAST framework provides TYPE(MeshType)**
- **Used for defining point, line, surface, volume meshes that need to be passed as input or output to a FAST component**
- **Represented as elements constructed from nodes with associated fields:**
  - Position
  - Displacement, rotational velocity, orientation, translational velocity, force, moment, added mass
  - Scalar fields
- **Defined in developer-provided ModName_Init routine**
- **More on meshes shortly…**

# FAST Framework: Units

- **Use SI base units for passing data through the module interface**
  - kilograms, meters, seconds, and radians
- **Use the following units for MeshType fields of forces, moments, and added mass:**
  - *per unit length* for line elements
  - *per unit area* for surface elements
  - *per unit volume* for volume elements
  - *lumped (concentrated)* for point elements

# FAST Framework: Coordinate Systems

The coordinate positions and loads passed between modules in the FAST framework are assumed to be relative to a global coordinate system.




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

# FAST Framework: Coupling Modules Together

- **NREL will develop the code that couples modules together (glue code)**
  - Module developers *must* follow the framework specifications and Programming Handbook guidelines
  - Time may not always be advancing
  - Unless otherwise specified, we cannot guarantee that the glue code will call subroutines in a particular order
- **Tasks of the glue code**
  - Interconnect individual modules
  - Derive inputs from outputs (module developer provides the equations)
  - Maps spatial interface meshes (may be non-matching) using nearest-neighbor locations for transfer of output data to input data
  - Drive the overall solution forward
  - Integrate the coupled system equations (in tight coupling schemes only)
  - Drive linearization calculations (in tight coupling schemes only)

# FAST Framework: Handling Errors

- **The glue code will handle all errors (possibly write to screen or log file, call routines to end modules, etc.)**
- **Do not allow the program to end in a module**
  - Trap errors and return error code to calling (driver) program.
  - ErrStat argument is an integer parameter from the NWTC Library indicating severity:
    - ErrID_None, ErrID_Info, ErrID_Warn, ErrID_Severe, or ErrID_Fatal
  - ErrMsg is a string of characters describing the error (empty if no error occurred)

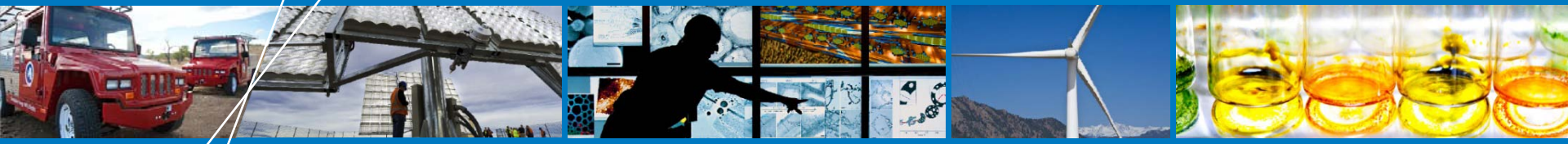
# FAST Framework: Handling I/O

- **Details still being worked on**
- **Current recommendations:**
  - Keep I/O consistent with FAST's current look and feel
    - Modules read own input files
    - Modules have option to write output files
  - Keep I/O in separate subroutines and/or modules that can easily be modified
  - Possibly define and use a derived data type containing all input file information for the module
    - Flexibility: could be populated by either the module or the glue code (added to the ModName_InitInputType)
    - Error checking on the input file data should occur in a separate subroutine in the module
  - For a single master output file, add variable called WriteOutput to ModName_OutputType data type
    - WriteOutput will not appear in Jacobians
    - Glue code may have to interpolate this array for time intervals consistent across all modules

# Inter-Language Interfaces

- **FAST Framework is Fortran 2003 but interfaces to non-Fortran (C, C++) modules through "wrappers"**
  - Developer provides Fortran versions of the FAST module interface routines (Table 1, Programmer's Handbook)
  - Pack/unpack the types passed into/out of buffers, which are 1-D arrays of reals, doubles, integers
  - Pass buffers to developer code subroutines
  - Developer code packs/unpacks these into/out of its own data structures
  - Some overhead from copying
- **Fortran side of wrappers auto-generated (coming soon)**

NREL
NATIONAL RENEWABLE ENERGY LABORATORY



# Thanks for your attention