Component Object Manager Users Guide
Version 0.1.0

IllinoisRocstar LLC

October 10, 2016

License

The software package sources and executables referenced within are developed and supported by Illinois Rocstar LLC, located in Champaign, Illinois.The software and this document are licensed by the University of Illinois/NCSA Open Source License (see opensource.org/licenses/NCSA). The license is included below.

For more information regarding the software, its documentation, or support agreements, please contact Illinois Rocstar at:

- **tech@illinoisrocstar.com**
- **sales@illinoisrocstar.com**

0.1 Introduction

Large-scale numerical simulation of a complex system, such as a solid rocket motor (SRM), requires consideration of multiple, interacting physical components, such as fluid dynamics, solid mechanics, and combustion. Numerical simulations of such systems are commonly called *multiphysics* simulations. Because of their multidisciplinary nature, development of such a multiphysics simulation capability typically involves a broad range of expertise and collaboration among many groups or institutions. It is common to take a partitioned approach, in which the individual physics codes are developed more or less independently of one another, and the software integration effort required to orchestrate them into a coherent system.

The objective of the Component Object Manager (COM) is to ease the integration of such independently developed applications into a coherent, integrated software system, particularly in a distributed parallel setting. It is designed to maximize concurrency in development of different applications and components, minimize user effort in software integration, and provide interoperability between different programming languages (in particular, C, C++, and Fortran 90).

The motivating application for the this integration infrastructure is Illinois Rocstar's flagship multiphysics simulation application, *Rocstar*. Originally developed by the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois DOE ASCI Center (`http://www.csar.uiuc.edu`), for simulating SRM, *Rocstar* is now a general multiphysics application and is applicable to fluid-structure interaction (FSI) across a moving, reacting interface.

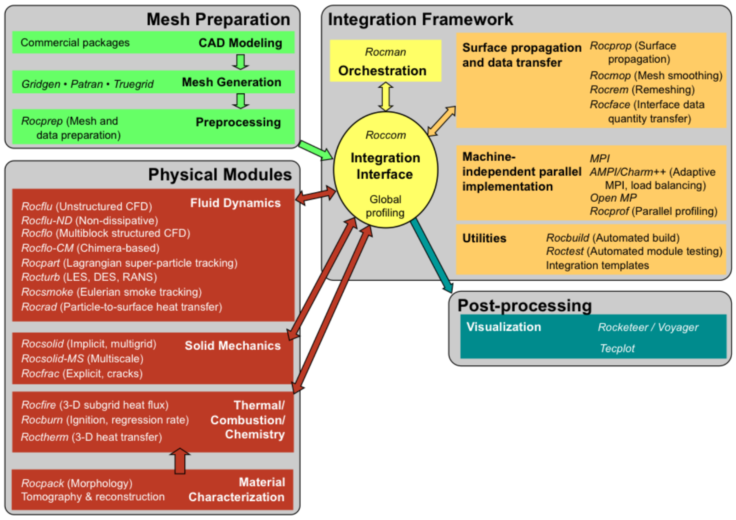
Mesh Preparation
Commercial packages
CAD Modeling
Gridgen • Patran • Truegrid
Mesh Generation
Rocprep (Mesh and data preparation)
Preprocessing
Physical Modules
Rocflu (Unstructured CFD)
Rocflu-ND (Non-dissipative)
Rocflo (Multiblock structured CFD)
Rocflo-CM (Chimera-based)
Rocpart (Lagrangian super-particle tracking)
Rocturb (LES, DES, RANS)
Rocsmoke (Eulerian smoke tracking)
Rocrad (Particle-to-surface heat transfer)
Fluid Dynamics
Rocsolid (Implicit, multigrid)
Rocsolid-MS (Multiscale)
Rocfrac (Explicit, cracks)
Solid Mechanics
Rocfire (3-D subgrid heat flux)
Rocburn (Ignition, regression rate)
Roctherm (3-D heat transfer)
Thermal/ Combustion/ Chemistry
Rocpack (Morphology)
Tomography & reconstruction
Material Characterization
Integration Framework
Rocman
Orchestration
Roccom
Integration Interface
Global profiling
Surface propagation and data transfer
Rocprop (Surface propagation)
Rocmop (Mesh smoothing)
Rocrem (Remeshing)
Rocface (Interface data quantity transfer)
Machine-independent parallel implementation
MPI
AMPI/Charm++ (Adaptive MPI, load balancing)
Open MP
Rocprof (Parallel profiling)
Utilities
Rocbuild (Automated build)
Roctest (Automated module testing)
Integration templates
Post-processing
Visualization
Rocketeer / Voyager
Tecplot

Figure 1: *Rocstar* architecture. Many software components (i.e. modules) interact through the integration infrastructure.

Figure 1 shows the overall structure of the current generation of *Rocstar*. In the *Rocstar* architecture, user applications are built as *modules* which are integrated into the composite software system through the *Rocstar* integration interface, *Roccom*. *COM* is the integration interface of *Rocstar* extracted and generalized for use in arbitrary parallel software integration efforts.

COM categorizes modules into two types: *application modules* (including *computation (physics) modules* and *orchestration modules*) and *service modules* In Figure 1, the boxes on the left show the physics modules. On the top is the *orchestration* module, *Rocman*, which manages the coupling algorithms. On the right are the computer science modules that provide services to the physics and orchestration modules through *COM*. Typically, the physics modules are written in Fortran 90 and the service modules are written in C++. The parallel implementation uses the standard Message Passing Interface (MPI) for all modules.

Although it was motivated specifically by the needs of the rocket simulation application described above, the integration infrastructure we have developed is quite general, and should be equally applicable to many other multiphysics simulations involving multiple, interacting software modules representing various physical components, especially those based on spatial decomposition into geometric domains with associated meshes. *COM* provides systematic methods for modules in a complex simulation to keep track of their data and to access data defined by other modules. Besides declaring variables and allocating buffers, each computation module registers its datasets with *COM*. These datasets can later be retrieved from *COM* by the same module or other modules, using parameters such as data block number, attribute name, etc. Functions can be registered and invoked in a similar way through *COM*. This scheme allows great independence in design and development of individual modules, hides the coding details and potential IP of different modules, and can enable plug-and-play of different modules.

COM is composed of three parts: a simple API (Application Programming Interface) for application modules, a C++ interface for developing service modules, and a runtime system. The API provides subroutines for registering the public data and functions of a module, querying a publicized data of a module, and invoking registered functions. In general, the API is the only part that application code developers need to learn in order to use *COM*. After an application code registers its data with *COM*, it can easily take advantage of the service utilities built on top of *COM*'s developers interface (such as parallel I/O). *COM* also provides support for eliminating global variables from application codes, which is highly desirable in threaded environments. In this documentation, we address only the general concepts of *COM* and its API. For a more in-depth discussion on the developers interface or runtime system, please see the (upcoming) *COM* Developers Guide.

0.2 Overview

COM (standing for Component Object Manager) is a component-based, object-oriented, data-centric software integration infrastructure, which provides a systematic, object-oriented, data-centric approach for intermodule interaction. Using infrastructure constructs, a computation module implements a *Component-side Client* (CSC) which creates distributed objects called *ComponentInterfaces* (CI) and registers its datasets into ComponentInterface instances called *Windows*. With the authorization of their owner modules or the orchestration module, these datasets can later be retrieved from *COM* by other modules using handles provided by *COM*. Functions can be registered and invoked similarly through *COM*. This scheme allows great independence in design and development of individual modules, hides the coding details of different research subgroups, and provides additional features such as automatic tracing and profiling.

0.2.1 Object-Oriented Interfaces

to simplify inter-module interfaces, *COM* utilizes an object-oriented methodology for abstracting and managing the data and functions of a module. This abstraction is mesh- and physics-aware and supports encapsulation, polymorphism, and inheritance.

ComponentInterface Windows and Panes *COM* organizes data and functions into distributed objects called *ComponentInterface* Windows. A CI Window (or simply window) encapsulates a number of *DataItems* (such as the mesh and some associated field variables) and public *functions* of a module, any of which can be empty. DataItems may be gathered together into groups called *DataGroup*s. A window can be partitioned into multiple frames called *Panes*, each of which instantiate a DataGroup. Panes and their DataGroups are useful for exploiting parallelism or for distinguishing different material or boundary-condition types. In a parallel setting, a pane belongs to a single process, while a process may own any number of panes. All panes of a given window must have the same DataGroup, although the total sizes of the Pane's DataGroup DataItems may vary. A module constructs windows inside its CSC at runtime by creating DataItems and registering the addresses of the DataItems and functions. Typically, the DataItems registered with *COM* are *persistent* (instead of temporary) datasets, in the sense that they live throughout the simulation (except that CI windows may need to be reinitialized at some events, such as remeshing). Different modules can communicate with each other only through their CI windows, as illustrated in Figure 2.

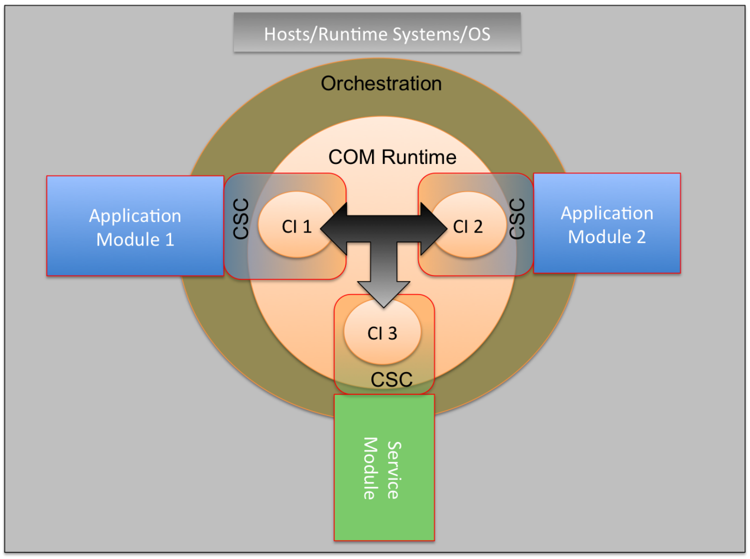

Figure 2: *COM* architecture with multiple software modules. Each module is loaded at runtime and shares the same process with the orchestrator. All module-module interactions are conducted through the CI and mediated by *COM*.

A code module references CI windows, DataItems, or functions using their names, which are of character-string type. Window names must be unique across all modules, and an DataItem or function name must be unique within a window. A code module can obtain an integer *handle* of (i.e., a reference to) an DataItem/function from *COM* with the combination of the window and DataItem/function names. The handle of an DataItem can be either *mutable* or *immutable*, where an immutable handle allows only read operations to its referenced DataItem, similar to a const reference in C++. Each pane has a user-defined positive integer ID, which must be unique within the window across all processors but need not be consecutive.

DataItems DataItems of a module can include mesh data, field variables, and other data associated with the CI window or pane. The former two types of DataItems are associated with nodes or elements. A nodal or elemental DataItem of a pane is conceptually a two-dimensional dataset: one dimension corresponds to the nodes/elements, and the other dimension corresponds to the data within a node/element. The dataset can be stored in a row- or column-major two-dimensional array, or be stored in separate arrays for each component of the dataset. *COM* allows users to specify a stride (the distance in the base data type, such as int or double precision) between the same component of two consecutive items (such as nodes/elements).

Mesh Data Mesh data include nodal coordinates, pane connectivity, and element connectivity (or simply connectivity), whose DataItem names and data types are predefined by *COM*. The nodal coordinates are double-precision floating-point numbers, with three components per node. The pane connectivity specifies the communication patterns between nodes shared by two or more panes, and is a pane DataItem packed in a 1-D array. The registration of pane connectivity is desirable for many purposes, but it is optional and can be computed automatically from coordinates using Rocmap.

COM supports both surface and volume meshes, which can be either multi-block structured or unstructured with mixed elements. For multi-block meshes, each block corresponds to a pane in a window. For unstructured meshes, each pane has one or more connectivity tables, where each connectivity table contains consecutively numbered elements (i.e., their corresponding field variables are stored consecutively) of the same type. Each connectivity table must be stored in an array with contiguous or staggered layout. To facilitate parallel simulations, *COM* also allows a user to specify the number of layers of ghost nodes and cells for structured meshes, and the numbers of ghost nodes and cells for unstructured meshes.

Field Variables Field variables are nodal or elemental DataItems that have no designated names or data types. A user must first define such an DataItem in the window and then register the addresses of the DataItem for each pane. For a specific pane, if a field variable is stored in one single array, then the array is registered with a single call; if it is stored in multiple arrays, then the user must register these arrays separately.

Windowed and Panel DataItems A data member can also be associated with either the CI window or a pane. Examples of windowed DataItems include data structures that encapsulate the internal states of a module, its CI, or some control parameters. An example of a pane DataItem is an integer flag for the boundary condition type of a surface patch. Similar to field variables, these DataItems do not have designated names or data types, and must be created within a CI window and then registered, or allocated.

Aggregate DataItems In *COM*, although DataItems are registered as individual arrays, DataItems can be referenced as an aggregate. For example, the name "mesh" refers to the collection of nodal coordinates and element connectivity; the name "all" refers to all the data DataItems in a window. For staggered DataItems, one can use "i-DataItem" ($i \geq 1$) to refer to the ith component of the DataItem or use "DataItem" to refer to all components collectively.

Aggregate DataItems enable high-level inter-module interfaces. For example, one can pass the "all" DataItem of a window to a parallel I/O routine to write all of the contents of a window into an output file with a single call. As another example, it is sometimes more convenient for users to have *COM* allocate memory for data DataItems and have application codes retrieve memory addresses from *COM*. *COM* provides a call for memory allocation, which takes a window DataItem name pair as input. A user can pass in "all" for the DataItem name, which will have *COM* allocate memory for all the unregistered DataItems.

0.2.2 Functions

A CI can contain not only data members but also function members. A module can register a function into its CI window, to allow other modules to invoke the function through *COM*. Registration of functions enables a limited degree of runtime polymorphism. It also overcomes the technical difficulty of linking object files compiled from different languages, where the mangled function names can be platform and compiler dependent.

Member Functions Except for very simple functions, a typical function needs to operate with certain internal states. In object-oriented programs, such states are encapsulated in an "object", which is passed to a function as an argument instead of being scattered into global variables as in traditional programs. In some modern programming language, this object is passed implicitly by the compiler to allow cleaner interfaces.

In mixed-language programs, even if a function and its context object are written in the same programming language, it is difficult to invoke such functions across languages, because C++ objects and F90 structures are incompatible. To address this problem, we introduce the concept of member functions of DataItems into *COM*. Specifically, during registration a function can be specified as the member function of a particular data DataItem within one of its CI windows. *COM* keeps track of the specified DataItem and passes it implicitly to the function during invocation, in a way similar to C++ member functions. Because the caller no longer needs to know the context object of the callee, this concept overcomes the incompatibility without sacrificing object-orientedness.

Optional Arguments *COM* supports the semantics of optional arguments similar to that of C++ to allow cleaner codes. Specifically, during function registration a user can specify the last few arguments as optional. *COM* passes null pointers for those optional arguments whose corresponding actual parameters are missing during invocation.

0.2.3 Inheritance

In multiphysics simulations, inheritance of CI data on the interface surface between domains is useful in many situations. First, the orchestrator sometimes needs to create data buffers associated with a computation module for the manipulation of jump conditions. Inheritance of windows allows the orchestration module to create a new window for extension or alteration, without altering an existing application's CI. Second, a module may need to operate on a subset of the mesh of another module. In rocket simulation, for example, the combustion module needs to operate on the burning surface between the fluid and solid. Furthermore, the orchestrator sometimes needs to split a user-defined CI into separate windows based on boundary-condition types, so that these subwindows can be treated differently (e.g., written into separate files for visualization). Figure 3 depicts a scenario of inheritance among three windows.

To support these needs, *COM* allows inheriting the mesh from a parent window to a child window in either of two modes. First, the mesh can be inherited as a whole. Second, only a subset of panes that satisfy a certain criterion are inherited. After inheriting mesh data, a child window can inherit data members from its parent window, or other windows that have the same mesh (this allows for *multiple inheritance*). The child window obtains the data only in the panes it owns and ignores other panes. During inheritance, if an DataItem already exists in a child window, *COM* overwrites the existing DataItem with the new DataItem.

COM supports two types of inheritance for data members: cloning (with duplication) and using (without duplication). The former allocates new memory space and makes a copy of the data DataItem in the new window, and is safer in terms of data integrity. The latter makes a copy of the references of the data member, which avoids the copying overhead associated with cloning and guarantees data coherence between the parent and child, and is particularly useful for implementing orchestration modules.

0.2.4 Data Integrity

In complex systems, data integrity has profound significance for software quality. Two potential issues can endanger data integrity: dangling references and side effects. *COM* addresses these issues through the mechanisms of persistency and immutable references, respectively.

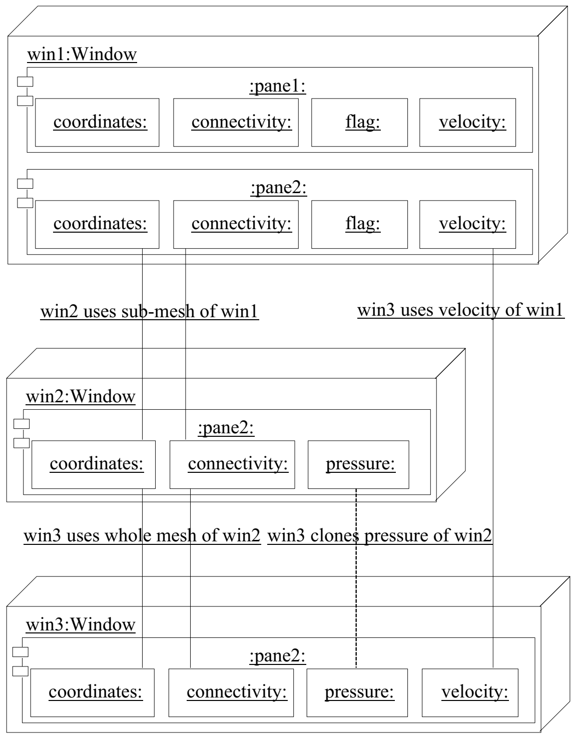

Figure 3: Scenario of inheritance of mesh and field DataItems among three CI windows.

Persistency *COM* maintains references to the datasets registered with its CI. To avoid dangling references associated with data registration, *COM* imposes the following persistency requirement: the datasets registered with a CI window must outlive the life of the window. Under this model, any persistent object can refer to other persistent objects without the risk of dangling references. In a heterogeneous programming environment without garbage collection, persistency cannot be enforced easily by the runtime systems instead, it is considered as a design pattern that application code developers must follow.

Immutable References Another potential issue for data integrity is side effects due to inadvertent changes to datasets. For the internal states of the modules, *COM* facilitates the traditional integrity model through member functions described earlier. In *COM*, a service module can obtain accesses to another module's data DataItems only through its function arguments, and *COM* enforces at runtime that an immutable handle cannot be passed to mutable arguments.

0.3 Architecture of *COM*

The core of *COM* is composed of three parts: an Application Programming Interface (API), a C++ class interface for development of service modules, and a runtime system for the bookkeeping associated with data objects and invocation of functions.

0.3.1 *COM* API

The *COM* API supplies a set of primitive function interfaces to physics and service modules and orchestrators for system setup, CI management, information retrieval, and function invocation. The subset of the API for CI management serves essentially the same purpose as the Interface Definition Language (IDL) of other frameworks (such as CCA), except that *COM* parses the definitions of the CI at runtime. *COM* provides different bindings for C++ and F90, with similar semantics. See Section 0.5 for details.

0.3.2 C++ Class Interfaces

COM provides a unified view of the organization of distributed data objects for service modules through the abstractions of CI windows and panes. Internally, *COM* organizes windows, panes, DataItems, functions, and connectivities into C++ objects, whose associations are illustrated in Figure 4,on a UML class diagram.

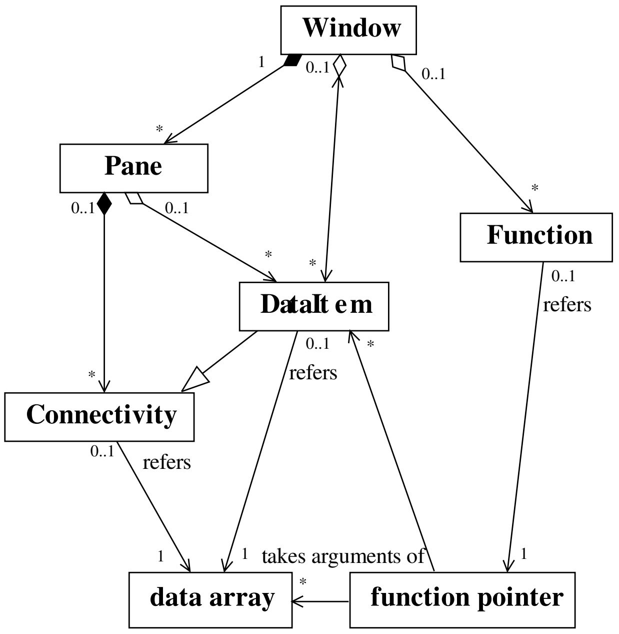

Figure 4: UML associations of *COM*'s classes.

A ComponentInterface window maintains a list of its local panes DataItems, and functions; a Pane object contains a DataGroup which is a list of DataItems and connectivities; a DataItem object contains a reference to its owner window. By taking references to DataItems as arguments, a function can follow the links to access the data DataItems in all local panes. The C++ interfaces conform to the principle of deeply immutable references, ensuring that a client can navigate through only immutable references if the root reference was immutable. Through this abstraction, the developers can implement service utilities independently of application codes, and ensure applicability in a heterogeneous environment with mixed meshes, transparently to physics modules.

0.3.3 *COM* Runtime System

The runtime serves as the middleware between modules. It keeps track of the user-registered data and functions. During function invocation, it translates the function and DataItem handles into their corresponding references with an efficient table lookup, enforces access protection of the DataItems, and checks whether the number of arguments of the caller matches the declaration of the callee. Furthermore, the runtime system also serves as the middleware for transparent language interoperability. For example, if the caller is in F90 whereas the callee is in C++, the runtime system will null-terminate the character strings in the arguments before passing to the callee.

Through the calling mechanism, *COM* also provides tracing and profiling capabilities for inter-module calls to aid in debugging and performance tuning. It also exploits hardware counters through PAPI to obtain performance data such as the number of floating-point instructions executed by modules. A user can enable such features using command-line options without additional coding.

0.4 Module Requirements

A *COM* application has a driver or an orchestrator, which is responsible for system setup and invoking the registered functions in turn. Each *COM*-compliant module must provide a load-module routine, which creates a CI window to encapsulate its interface functions and context objects, and an unload-module, which destroys the window, where the window name is typically the same as that of the module. By calling the load-module routines, the driver dynamically loads a set of modules into the runtime system. Through *COM*'s calling mechanism, the orchestrator then invokes the functions of the physics and service modules, which in turn can also invoke functions provided by other modules.

0.5 COM API

COM provides different bindings for C, C++, and Fortran 90, with similar semantics, except that C/C++ is case-sensitive whereas Fortran is case-insensitive, and C/C++ passes arguments by value whereas Fortran passes by reference. Another subtle difference is that Fortran character strings, which are not null terminated by default, must be interpreted differently from C/C++ character strings. These differences are apparent in the prototype definitions of the subroutines, but are mostly transparent to users. COM's interface prototypes are defined in "com.h" (for C/C++) and "comf90.h" (for Fortran 90), which must be included by the codes in corresponding languages, respectively.

COM's interface subroutines follow the following conventions: They all start with the prefix COM_, followed by lower-case letters (for C/C++). Most subroutines return no values unless otherwise specified. If a non-fatal error occurred inside a COM subroutine, an error flag will be set. For the C and Fortran interfaces, the error code can be obtained by calling **COM_get_error_code**. For the C++ interface, the error code will be thrown as an exception.

Although COM's API has about 40 functions, a simple computation module needs to use only about 10 of them, mostly in Section 0.5.2. The other functions are more advanced and provided mostly for the orchestrators and for more complex physics modules.

0.5.1 Initialization and Finalization

Startup and Shutdown of COM Implementations of COM's runtime system require some setup operations before any other COM operations can be performed. To provide for this, COM includes an initialization subroutine **COM_init**.

C:	**COM_init**(int *argc, char ***argv)
Fortran:	**SUBROUTINE COM_INIT**

This subroutine must be called exactly once from every process before any other COM subroutine (apart from **COM_initialized**) is called. It is typically called from the driver routine of the application. The C version accepts the arguments argc and argv, which are the arguments of the main routine of C. **COM_init** parses the following options:

- "-com-v *n*": Set the verbose level of all processes to *n* (see **COM_set_verbose**);
- "-com-v*p* *n*": Set the verbose level of process *p* to *n* (see **COM_set_verbose**);
- "-com-mpi": Call MPI_Init within COM_init.
- "-com-home <directory>":Search for shared libraries under <directory>/lib. Alternatively, one can pass the directory by setting either COM_HOME or ROCSTAR_HOME enrionment variables.

The Fortran version **COM_INIT** takes no arguments. A corresponding subroutine **COM_finalize** is also provided for COM to clean up its state after the execution of the program. It also needs to be called on every process. Once this subroutine is called, no COM subroutine may be called.

C: **COM_finalize**(void)

Fortran: **SUBROUTINE COM_FINALIZE**

The following is a piece of code in C that illustrates its usage.

```
int main(int argc, char **argv) {
  COM_init(&argc, &argv);
  /* main program */
  COM_finalize();
}
```

COM provides a subroutine **COM_initialized** for checking whether **COM_init** has been called.

C: int **COM_initialized**(void)

Fortran: **FUNCTION COM_INITIALIZED()**
INTEGER :: **COM_INITIALIZED**

The argument flag is set to nonzero if **COM_init** has been called and zero otherwise.

Furthermore, COM runtime environment can also be shut down abnormally by calling **COM_abort**.

C: void **COM_abort**(int ierr)

Fortran: **SUBROUTINE COM_ABORT**(IERR)
INTEGER, INTENT(IN) :: IERR

This function terminates a COM program and returns the error code ierr to the invoking environment. If MPI was initialized, then **COM_abort** calls MPI_Abort internally on MPI_COMM_WORLD. Otherwise, it calls the exit function of the standard C library. For Fortran codes, COM also provides a related subroutine **COM_CALL_EXIT**, which we describe in Section 0.6.6.

Loading and Unloading of Modules In the COM infrastructure, user applications can be built into a dynamic library named "libfoo.so" (where "foo" is arbitrary). The dynamics libraries are called *modules*, and COM provides the following interface to load and unload the dynamic library for a module.

C:	**COM_load_module**(const char *modName, const char *winName)
Fortran:	**SUBROUTINE COM_LOAD_MODULE**(MODNAME, WINNAME) CHARACTER(*), INTENT(IN) :: MODNAME, WINNAME
C:	**COM_unload_module**(const char *modName, const char *winName=NULL)
Fortran:	**SUBROUTINE COM_UNLOAD_MODULE**(MODNAME,WINNAME) CHARACTER(*), INTENT(IN) :: MODNAME,WINNAME OPTIONAL :: WINNAME

The argument modName is the main part of the library name (e.g., "foo" for libfoo.so). The "foo" module needs to supply two subroutines, **foo_load_module** and **foo_unload_module**, which takes winName as its argument. When **COM_load_module/** or **COM_unload_module** is called, COM locates the symbol **foo_load_module/foo_unload_module** in libfoo.so, respectively, and invokes these user-provided routines by passing winName to load/unload the module. For **COM_unload_module**, the winName argument can be omitted if the module is loaded only once.

For ease of debugging, sometimes it is desirable to build COM modules and applications statically. In this case, an application need to define the prototypes of **foo_load_module** and **foo_unload_module** and call them directly instead of through **COM_load_module** or
COM_unload_module. COM provides the following macros to C/C++ codes (defined when "com.h" is included):

COM_EXTERN_MODULE(modName_noquotes)

COM_LOAD_MODULE_STATIC_DYNAMIC(modName_noquotes, winName)

COM_UNLOAD_MODULE_STATIC_DYNAMIC(modName_noquotes, winName)

Depending on whether the macro **STATIC_LINK** is defined (e.g., by passing -DSTATIC_LINK to the compiler) or not, these macros expands to different statements. When **STATIC_LINK** is defined, then **COM_EXTERN_MODULE**(foo) defines **foo_load_module** and **foo_unload_module**. Otherwise, it expands to noop. For C++ codes, since **COM_EXTERN_MODULE** uses the extern "C" modifier, it cannot be used inside a function. **COM_LOAD_MODULE_STATIC_DYNAMIC**(foo, "FOO") expands to **foo_load_module**("FOO") if the **STATIC_LINK** is defined, and to **COM_load_module**("foo","FOO"), otherwise; similarly for **COM_UNLOAD_MODULE_STATIC_DYNAMIC**.

0.5.2 Data and Function Registration

COM organizes data and functions into CI windows. A window encapsulates a number of data members (such as the mesh and some associated data DataItems) and public functions of a module. A module can create any number of CI windows. Technically, an application has only one CI with multiple windows, but this distinction is currently irrelavent. All panes of a window must have the same DataGroup, although the size of each DataItem may vary.

0.5.3 Creation of CI Window

A call to **COM_new_window** creates an empty window with a given name.

C++: **COM_new_window**(const std::string wName, MPI_Comm comm=MPI_COMM_SELF)

C: **COM_new_window**(const char *wName, MPI_Comm comm=MPI_COMM_SELF)

Fortran: **SUBROUTINE COM_NEW_WINDOW**(WNAME, COMM)
CHARACTER(*), INTENT(IN) :: WNAME
OPTIONAL, INTEGER, INTENT(IN) :: COMM

The wName argument is a character string and must be unique across all modules, and comm is the MPI communicator of the owner processes of the window. Because a window is a collective concept, this subroutine should be called on all processes within the MPI communicator. The communicator can be retrieved by calling **COM_get_communicator** (see subsection 0.8).

After a window is created, a user can create data DataItem members and register addresses of data and functions to it as described later, followed by calling **COM_window_init_done** to mark the end of the registration of a window.

C++: **COM_window_init_done**(const std::string name, int clct=1)

C: **COM_window_init_done**(const char *name, int clct=1)

Fortran: **SUBROUTINE COM_WINDOW_INIT_DONE**(NAME, CLCT)
CHARACTER(*), INTENT(IN) :: NAME
INTEGER, INTENT(IN), OPTIONAL :: CLCT

It also takes the window name as its first argument. The second argument clct specifies whether the function is being called collectively on all the owner processes of the window, so that the mapping from panes to processes can be determined. If any pane was created or deleted, then **COM_window_init_done** must be called collectively before the window is used, by passing a non-zero value (the default) to clct. After calling **COM_window_init_done**, the sizes and arrays of the DataItems can be changed without calling **COM_window_init_done** again. DataItems and/or panes can be added or deleted, given that **COM_window_init_done** is called after the changes. If the DataItems were changed but no panes were added or deleted from any process, then **COM_window_init_done** can be called with clct equal to zero, to avoid recomputing the pane-to-process mapping.

DataItems

Declaration of New DataItems Besides mesh data, a window can have other data members, which can be associated with the window, a pane, nodes, or elements of the pane. Different from keywords, these data DataItems do not have designated names or data types. Therefore, a user must first define an DataItem by calling **COM_new_dataitem** before registering the addresses of the DataItem. Again, this subroutine must be called collectively on all owner processes of a window.

C++: **COM_new_dataitem**(const std::string aName, char loc, COM_Type type,
int ncomp, const char *unit)

C:	**COM_new_dataitem**(const char *aName, char loc, COM_Type type, int ncomp, const char *unit)
Fortran:	**SUBROUTINE COM_NEW_DATAITEM**(ANAME, LOC, TYPE, NCOMP, UNIT) CHARACTER(*), INTENT(IN) :: ANAME, FNAME, UNIT CHARACTER*1, INTENT(IN) :: LOC INTEGER, INTENT(IN) :: TYPE, NCOMP

In the argument list, aName is the DataItem name in the format of "window DataItem", similar to mesh data names; loc can be either '**w**', '**p**', '**n**', or '**e**', corresponding to *windowed*, *panel*, *nodal*, or *elemental* data; type specifies the base datatype of the DataItem, which can be one of the following constant C/C++ Data types:

- COM_CHAR
- COM_UNSIGNED_CHAR
- COM_BYTE
- COM_SHORT
- COM_UNSIGNED_SHORT
- COM_INT
- COM_UNSIGNED
- COM_LONG
- COM_UNSIGNED_LONG
- COM_FLOAT
- COM_DOUBLE
- COM_LONG_DOUBLE
- COM_BOOL

or Fortran Data types:

- COM_CHARACTER
- COM_LOGICAL
- COM_INTEGER
- COM_REAL
- COM_DOUBLE_PRECISION
- COM_COMPLEX
- COM_DOUBLE_COMPLEX

or other:

- COM_MPI_COMMC
- COM_MPI_COMM(=COM_MPI_COMMC)
- COM_MPI_COMMF
- COM_MAX_TYPEID(=COM_MPI_COMMF)
- COM_STRING
- COM_RAWDATA
- COM_METADATA
- COM_VOID
- COM_F90POINTER
- COM_OBJECT
- COM_MIN_TYPEID

ncomp is the number of components of the DataItem (for example, the number of entries associated with each node/element for nodal/elemental data); and unit is the unit of the DataItem, which can be the empty string "" if the DataItem is unitless.

One can call COM_new_dataitem on an existing DataItem to re-define an DataItem (including the keyword "nc"). However, one cannot increase the number of components of the predefined DataItems (see next subsection). After calling COM_new_dataitem, the previously registered data are reset, and its handles and inherited DataItems become invalid. It is the user's responsibility to ensure the consistency for the DataItem.

Predefined Mesh Data Mesh data, including nodal coordinates, pane connectivity, and element connectivity (or simply connectivity), have predefined DataItem names and data types. Nodal coordinates ("nc") are predefined as double-precision nodal DataItems with three components (corresponding to x, y, and z, respectively) per node and a default unit "m". However, nodal coordinates may be redefined by calling COM_new_dataitem to have less than 3 components, a different base data type, or a different unit. Pane connectivity ("pconn") is predefined as a 1-D integer pane DataItem with no unit.

"nc" for nodal coordinates
"pconn" for pane connectivity

For each pane, the pane-connectivity array can have multiple blocks:

1. shared nodes;
2. real nodes to send;
3. ghost nodes to receive;

4. real cells to send;
5. ghost cells to receive.

The first block goes into the real part of pconn and blocks 2–5 go into the ghost part. Blocks 2 and 3 must be present together, so are blocks 4 and 5. The ghost part of pconn is optional, and within the ghost part of pconn, blocks 4 and 5 are optional. Each block has the following content:

```
<number of communicating-pane blocks to follow>
<communicating pane id 1>
<#local nodes to follow>
<list of local nodes>...
... ! repeat for other remote panes
```

The lists of nodes for a pair of communicating panes are stored in the same order in their corresponding tables. Furthermore, the panes are stored in increasing order of pane IDs. If a node is shared by more than two panes, then every pair of shared nodes is stored in pconn. Note that it is possible for a single pane to have duplicated nodes, for example, in the case of branch-cut for structured meshes. In this case, in the block for shared nodes, the list of local nodes is composed of a series of node pairs, where the first node in the pair always has a smaller node ID than the second, and the number of local nodes is equal to twice the number of pairs. Note that in the case of partial inheritance, where a subwindow may inherit a subset of panes from a parent window, pconn may be inherited by the subwindow for its inter-pane communication, and as a consequence pconn may refer to some remote panes that no longer exist. In this case, it is important to note that the first number in each block is no longer the actual number of communicating panes, and a traversal of pconn should skip the nonexisting remote panes.

The names of element connectivities have the format of "**:***elementtype***:***aname*", where the ":*aname"* part is optional and is useful when there are multiple connectivity tables for one type of elements. Note that element connectivities are not regular DataItems, in that different panes may contain different types of elements, and an element connectivity must not be created by calling COM_new_dataitem but by setting its size and registering its address.

":st1:*aname*", **":st2**:*aname*", **":st3**:*aname*" for structured mesh of 1, 2 and 3 dimensions.
"**:b2**:*aname*" and **":b3**:*aname*" for 2- and 3-node bar elements.
":t3:*aname*", **":t6**:*aname*", **":q4**:*aname*", **":q8**:*aname*", and **":q9**:*aname*" for connectivity tables of 3- and 6-node triangles, and 4-, 8-, and 9-node quadrilaterals, respectively.
":T4:*aname*", **":T10**:*aname*", **":B8**:*aname*" ("**:H8**:*aname*"), and **":B20**:*aname*" for connectivity tables of 4- and 10-node tetrahedra, and 8- and 20-node bricks, respectively
":P5:*aname*", **":P14**:*aname*", **":P6**:*aname*" (**":W6**:*aname*"), **":P15**:*aname*" (**":W15**:*aname*"), and "**:P18**:*aname*" (**":W18**:*aname*") for connectivity tables of 5- and 14-node pyramids and 6-, 15-, and 18-node prisms (aka pentahedra or wedges), respectively.

For elements of unstructured meshes, COM uses the same numbering convention as the CFD General Notation System (CGNS), of which a detailed description can be found in Section 3.3 of CGNS Standard Interface Data Structures (`http://www.grc.nasa.gov/WWW/cgns/sids/sids.pdf`). If a pane has multiple connectivity tables, these tables must be registered in increasing order of the element numbering (i.e., the elements with smaller indices in field-variable arrays must be registered earlier), and ghost elements must be registered last. Note that for structured meshes, a pane can register only one connectivity using **COM_set_array_const** described in the next subsection, by passing in the numbers of nodes of all directions in a single array listed in Fortran convention (See example code in subsection 0.5.5). We will allow for users to add new element types in future releases.

Registration of Sizes One sets the sizes of an DataItem using the following routine.

C++:	**COM_set_size**(const std::string aName, int pane_id, int size, int ng=0)
C:	**COM_set_size**(const char *aName, int pane_id, int size, int ng=0)
Fortran:	**SUBROUTINE COM_set_size**(ANAME, PANE_ID, SIZE, NG) CHARACTER(*), INTENT(IN) :: ANAME INTEGER, INTENT(IN) :: PANE_ID, SIZE, NG OPTIONAL :: NG

In the arguments, aName is the data name in the format of "window.aname" or "window.:elementtype:aname" for connectivity tables paneID is a user-defined *positive* integer identifier of the pane, which must be unique within the window across all processors but need not be consecutive. Window DataItems should be registered with pane-ID 0. The argument size is either the total number of nodes (including ghost nodes) in the pane (for nodal coordinates), or the number of elements (including ghost elements, for a connectivity table), or the length of the dataset for panel or windowed DataItems. ng (optional in F90 and C++; default is 0) is either the number of ghost nodes in the pane for nodal data or the number of ghost elements for a connectivity table.

The default size of a window DataItem is 1, but is undefined for other types of DataItems. Note that setting the number of nodes for one nodal DataItem affects all other nodal DataItems, and it is more efficient to set size for "nc". Typically, one should set the number of elements for each element connectivity.

Registration of Preallocated Array After creating an DataItem, a user can register the address or addresses of the DataItem using the following subroutine.

C++:	**COM_set_array**(const std::string aName, int paneID, void *addr, int stride=0, int cap=0)
C:	**COM_set_array**(const char *aName, int paneID, void *addr, int stride=0, int cap=0)
Fortran:	**SUBROUTINE COM_SET_ARRAY**(ANAME, PANEID, ADDR, STRIDE, CAP) CHARACTER(*), INTENT(IN) :: ANAME INTEGER, INTENT(IN) :: PANEID, STRIDE, CAP <TYPE> :: ADDR OPTIONAL :: STRIDE, CAP

As for **COM_set_size**, the aName is either an DataItem name or the name to a connectivity table, and the paneID is a positive integer ID for the owner pane or 0 for window DataItems. The addr argument specifies the address of the array for the DataItem. If the components of each item are stored contiguously in an array of Array(stride,cap) in Fortran convention with stride>=ncomp and cap>=size, one can register the array by with a single call. If it is stored in an array of Array(cap, ncomp), then the stride should be set to 1. The stride argument can be omitted if stride is equal to ncomp, and it is invalid if stride is greater than 1 but smaller than ncomp. **In Fortran 90, it is very important not to register a scalar variable defined locally in a subroutine or function (i.e., a stack variable), unless it has the TARGET or POINTER property.**

The cap argument can be omitted if cap==size. Otherwise, the user must register an array for each individual component of the DataItem using DataItem name in the format "window.*i*-DataItem", where *i* is an integer between 1 and the number of components of the DataItem (Not applicable for connectivity tables). One can change the sizes and the arrays by calling **COM_set_size** and **COM_set_array**.

To protect data integrity, COM allows registration of a read-only data by calling **COM_set_array_const**, which takes the same arguments as **COM_set_array**.

C++: **COM_set_array_const**(...)

C: **COM_set_array_const**(...)

Fortran: **SUBROUTINE COM_SET_ARRAY_CONST**(...)

For Fortran 90, the types supported are scalars and pointers to 1-, 2-, and 3-dimensional integer, single-precision, and double-precision arrays. For other types of variables (such as a function pointer), one can register using one of the following two functions.

Fortran: **SUBROUTINE COM_SET_EXTERNAL**(ANAME, PANEID, VAR)
```
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID
EXTERNAL VAR
```

Fortran: **SUBROUTINE COM_SET_EXTERNAL_CONST**(ANAME, PANEID, VAR)

One can obtain a pointer set by **COM_set_array_const** or **COM_SET_EXTERNAL** only through **COM_get_array_const**.

Registration of Bounds A user can register the lower and upper bounds of a specific DataItem. One can register two sets of bounds: one set of hard bounds, which specifies the universal limits that the dataset must satisfy at all times and whose violation would result in runtime errors; the second set corresponds to soft bounds, whose violations would result in printing of warning messages at runtime.

C++: **COM_set_bounds**(const std::string aName, int pane_id,
const void *lbnd, const void *ubnd, int is_soft=0)

C: **COM_set_bounds**(const char *aName, int pane_id,
const void *lbnd, const void *ubnd, int is_soft=0)

Fortran: **SUBROUTINE COM_set_bounds**(ANAME, PANE_ID, LBND, UBND, IS_SOFT)
```
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANE_ID
<TYPE>, INTENT(IN) :: LBND, UBND
INTEGER, INTENT(IN), OPTIONAL :: IS_SOFT
```

When pane_id is 0, then the given bounds will be applied to all panes; if it is greater than 0, then they will be applied to the specific pane with the given pane ID. If the function is called for a vector DataItem, then the bounds apply to the magnitude of the vectors. One can also set the bounds for individual components by calling the function on the corresponding component DataItems (using DataItem names "window.*i*-DataItem"). The fifth argument, is_soft, which is optional with default value 0, specifies whether the bounds are hard (is_soft==0) or soft (is_soft$\neq$0).

0.5.4 Functions

A window can contain not only data members but also function members. A function is registered into a window by calling **COM_set_function** on all owner processes of the window.

C: **COM_set_function**(const char *fName, void (*faddr)(), const char* intents, COM_Type types[])

Fortran: **SUBROUTINE COM_SET_FUNCTION**(FNAME, FADDR, INTENTS, TYPES)
CHARACTER(*), INTENT(IN) :: FNAME, INTENTS
EXTERNAL FADDR
INTEGER, INTENT(IN) :: TYPES

Similar to DataItem names, fName has the format of "window.function". The argument faddr takes the actual function pointer. For the C interface, to register a function that takes at least one argument (note that all arguments must be pointers), a user code must cast the pointer to the void (*)() type, which is predefined as COM_Func_ptr. The argument intents is a character string of length equal to the number of arguments taken by the registered function, and its *i*th character indicates whether the *i*th argument is for input, output, or both if intents_i is '**i**'/'**I**', '**o**'/'**O**', or '**b**'/'**B**', respectively (see the Optional Arguments paragraph of this section for more discussion). The argument types is an integer array of length also equal to the number of arguments, and its *i*th entry indicates the data type of the *i*th argument. All arguments of a registered function must be passed by reference. If a function is expecting an integer pointer/reference for its *i*th argument, for example, the *i*th entry should be either COM_INT (for C/C++) or COM_INTEGER (for Fortran).

See the section on DataItems above for a list of supported data types.

Member Function Many functions perform operations in a specific context. In object-oriented programs, such contexts are typically encapsulated in objects instead of being scattered into global variables as in traditional programs. Such an object is passed into a function as an argument, and frequently is passed implicitly by the compiler to allow cleaner interfaces in modern programming languages.

To encourage object-oriented programming and cleaner interfaces of application codes, COM supports the concept of member functions of DataItems. A user registers a member function using the interface **COM_set_member_function**, which takes an DataItem name as an additional argument.

C: **COM_set_member_function**(const char *fName, void (*faddr)(), const char* aName, const char* intents, COM_Type types[])

Fortran: **SUBROUTINE COM_SET_MEMBER_FUNCTION**(FNAME, FADDR, ANAME, INTENTS, TYPES)
CHARACTER(*), INTENT(IN) :: FNAME, ANAME, INTENTS
EXTERNAL FADDR
INTEGER, INTENT(IN) :: TYPES(:)

The given DataItem should encapsulate the context of the registered function. The first entries in intents and types should specify the intention and data type of this DataItem, respectively. When an application code invokes a registered member function through COM, it will not list this DataItem in the arguments, but COM will pass it implicitly as the first argument to the function.

In addition, COM also provides a function for registering C++ member functions of a class, which must be a derived class of COM_Object. An object of a derived class of COM_Object, especially those with virtual functions, must be registered and retrieved using **COM_set_object**() and **COM_get_object**(), which takes the same arguments as **COM_set_array**() and **COM_get_array**(). The member functions are registered using the following interface,

C++: **COM_set_member_function**(const char *fName, void (COM_Object::*faddr)(), const char* aName, const char* intents, COM_Type types[])

and must be casted to the void (COM_Object::*faddr)() type, which is predefined as COM_Member_func_ptr. For example, a member function func of a class Rocfoo can be casted as

reinterpret_cast<COM_Member_func_ptr>(&Rocfoo:func).

Optional Arguments If a registered function is written in C or C++, the last few arguments can be specified as *optional*. COM will pass in null pointers for them if the caller omit these arguments. To specify an argument to be optional, a user should use uppercase letters 'I', 'O', or 'B' instead of 'i', 'o', or 'b' in its corresponding entry in intents.

Data Types As we noted earlier, all arguments of a registered data must be passed by reference. A primitive data type (such as COM_INT) used in the argument types would indicate that the function is expecting a pointer or reference to that type. There are three special cases, however. First, if a function is expecting a character string (vs. a single character), which must be null terminated for C/C++ functions or whose length must be passed in implicitly for Fortran functions, then the corresponding data type of the argument must be set to **COM_STRING**. This data type tells COM to adapt the string if necessary (such as null-terminating the charactering string) to bridge C/C++ and Fortran transparently from users. Second, if a function is a service utility written in C++ and is expecting a C++ object that contains the description of an DataItem, the corresponding data type of the argument must be set to **COM_METADATA**. If the function is expecting the physical address of a window DataItem, then the corresponding datatype should be **COM_RAWDATA**. In general, two types of arguments should use **COM_RAWDATA**: the implicit argument for a member function, and an argument that is a function pointer.

Limitations and Special Notes For language interoperability, a registered function must return no value, and all its arguments must be passed by reference (i.e., must be pointers/references for C/C++ functions). Due to technical reasons, COM has to impose a limit on the maximum number of the arguments that a registered function can take, and the limit is currently set to 7, including the implicit arguments passed by COM, i.e., the first argument of member functions and character lengths for Fortran functions. This preset limit is large enough for most applications, but can be enlarged by changing COM's implementation if desired. Similar to DataItems, a function can be registered multiple times, but only the address of the last registration will be used.

0.5.5 Example Code

The following is a piece of Fortran code segment that demonstrates the registration of data and functions.

```
INTEGER          ::  nn1, ni2, nj2    ! sizes of nodes
INTEGER          ::  ne1              ! sizes of elements
INTEGER          ::  types(2), dims(2)
INTEGER, POINTER          ::  conn1(3,ne1)
DOUBLE PRECISION, POINTER  ::  coors1(3,nn1), coor2(3,ni2, nj2)
DOUBLE PRECISION, POINTER  ::  disp1(3,nn1), disp2(3,ni2, nj2)
DOUBLE PRECISION, POINTER  ::  velo1(ne1,3), velo2(ni2-1, nj2-1,3)
EXTERNAL   fluid_update

CALL COM_NEW_WINDOW("fluid", MPI_COMM_WORLD)

! Create a node-centered double-precision dataset
CALL COM_NEW_DATAITEM( "fluid.disp", "n", COM_DOUBLE, 3, "m")

! Create a element-centered double-precision dataset
CALL COM_NEW_DATAITEM( "fluid.velo", "e", COM_DOUBLE, 3, "m/s")

! Create a pane with ID 11 of a triangular surface mesh
CALL COM_SET_SIZE("fluid.nc", 11, nn1)
CALL COM_SET_ARRAY("fluid.nc", 11, coors1, 3)
CALL COM_SET_SIZE("fluid.:t3:", 11, ne1)
CALL COM_SET_ARRAY("fluid.:t3:", 11, conn1, 3)

! Create a pane with ID 21 of a structured surface mesh
dims(1)=ni2; dims(2)=nj2;
CALL COM_SET_ARRAY_CONST("fluid.:st2:actual", 21, dims)
CALL COM_SET_ARRAY("fluid.nc", 21, coors2, 3)

! Register addresses of DataItems for both panes
CALL COM_SET_ARRAY("fluid.disp", 11, disp1)
CALL COM_SET_ARRAY("fluid.velo", 11, velo1, 1) ! Staggered layout
CALL COM_SET_ARRAY("fluid.disp", 21, disp2)
CALL COM_SET_ARRAY("fluid.velo", 21, velo2, 1) ! Staggered layout

! Register a function that takes two input arguments
type(1)=COM_DOUBLE; type(2)=COM_DOUBLE
CALL COM_SET_FUNCTION("fluid.update", fluid_update, "ii", types)

CALL COM_WINDOW_INIT_DONE("fluid")
......
CALL COM_DELETE_WINDOW("fluid")
```

0.6 Procedure Calls

0.6.1 DataItem and Function Handles

A handle is an integer from which COM can obtain the actual data about DataItems and functions. A user can obtain a mutable handle to an DataItem using **COM_get_dataitem_handle**, or an immutable handle using **COM_get_dataitem_handle_const**, which take the same arguments.

C++:	int **COM_get_dataitem_handle**(const std::string aName)
C:	int **COM_get_dataitem_handle**(const char *aName)
Fortran:	**FUNCTION COM_GET_DATAITEM_HANDLE**(ANAME) CHARACTER(*), INTENT(IN) :: ANAME INTEGER :: **COM_get_dataitem_handle**

The function can be called on user-defined DataItems, or a pre-defined DataItem "**nc**", "**conn**", "**pconn**", "**mesh**", "**pmesh**", "**data**", and "**all**", which refer to nodal coordinates, element connectivity, pane connectivity, mesh data (including coordinates and element connectivity), parallel mesh data (including mesh and pane connectivity), all field DataItems (excluding parallel mesh), and all DataItems, respectively. Note that it is illegal to call **COM_get_dataitem_handle** on connectivity tables, whose scopes are within panes instead of within windows.

To obtain a handle to a function, one should use **COM_get_function_handle** instead, whose prototype is essentially the same as **COM_get_dataitem_handle**.

If the function or DataItem exists, then a positive integer ID will be returned; otherwise, 0 will be returned. So these functions can be used to detect the existence of a function or DataItems. Similarly, one can detect the existence of a window by calling **COM_get_window_handle**.

0.6.2 Invocation

To invoke a function registered with COM in C or Fortran, a user need to use the following function.

C:	**COM_call_function**(int fHandle, int argc, void *arg1, ...)
Fortran:	**SUBROUTINE COM_CALL_FUNCTION**(FHANDLE, ARGC, ARG1, ...) INTEGER, INTENT(IN) :: FHANDLE, ARGC <TYPE> :: ARG1, ...

The first argument is a function handle, and the second DataItem is the number of arguments to be passed, followed by the pointers (or references) to the data values or DataItem handles.

For C++, we take advantage of the function overloading feature of the language to provide a cleaner interface **COM_call_function**.

C++:	**COM_call_function**(int fHandle, void *arg1, ...)

It does not require passing the number of arguments.

0.6.3 Call Tracing

To help debugging application codes, COM allows users to trace the procedure calls by setting a nonzero verbose level.

C:	**COM_set_verbose**(int v)
Fortran:	**SUBROUTINE COM_SET_VERBOSE**(V) INTEGER, INTENT(IN) :: V

If v is a positive number, then COM will print out traces of the calls up to depth $(v+1)/2$. If v is an odd number, COM will print only the names of the functions if v even, COM will also print the data types and values of the arguments passed to the functions.

0.6.4 High-Level Profiling

COM contains a simple profiling tool for timing the execution times of the functions invoked through COM.

C:	**COM_set_profiling**(int enable)
Fortran:	**SUBROUTINE COM_SET_PROFILING**(ENABLE) INTEGER, INTENT(IN) :: ENABLE

If enable is zero, it disables profiling; otherwise, it enables profiling and resets all the counters of the profiler.

In a parallel run, the timing results are typically more accurate if MPI_Barrier is called before and after a function call, but putting too many barriers may also affect performance. COM allows a user to control where barriers should be placed by the following call.

C:	**COM_set_profiling_barrier**(int fHandle, MPI_Comm comm)
Fortran:	**SUBROUTINE COM_SET_PROFILING_BARRIER**(FHANDLE, COMM) INTEGER, INTENT(IN) :: FHANDLE, COMM

This routine will enable COM to call MPI_Barrier on the given communicator before and after the given function for the processes of the given communicator.

The profiling results can be printed by calling

C:	**COM_print_profile**(const char *fname, const char *header)
Fortran:	**SUBROUTINE COM_PRINT_PROFILE**(FNAME, HEADER) CHARACTER(*), INTENT(IN) :: FNAME, HEADER

This routine will append the header and the timing results to the file with name fname. If fname is NULL or the empty string, the standard output will be used instead. A typical timing result looks as follows.

```
                   Function      #calls    Time(tree)    Time(self)
-----------------------------------------------------------------------
     Rocflu.update_solution         100       43.1856       42.8221
    Rocfrac.update_solution         100       30.4038       30.3861
 RFC.least_squares_transfer         400      0.957599      0.957599
......
-----------------------------------------------------------------------
      Total(top level calls)                                  74.806
```

In the output, the Time(tree) indicates the sum of the elapsed wall-clock time during the execution of a function since the last call to **COM_init_profiling**, and Time(self) subtracts the elapsed time of the calls made with the function.

0.6.5 Calling System Calls in Fortran

To allow Fortran to execute a shell command using the system call interface of C, COM provides the following **COM_CALL_SYSTEM** function.

Fortran: **FUNCTION COM_CALL_SYSTEM**(COMMAND)
CHARACTER(*), INTENT(IN) :: COMMAND

It will execute the command and return the return status of the command after the command has been completed; if the command fails to execute due to fork failure, then -1 will be returned.

0.6.6 Calling AtExit and Exit Functions In Fortran

A Fortran code can also call the atexit and exit functions of the C standard through COM.

Fortran: **FUNCTION COM_CALL_ATEXIT**(FUNC)
EXTERNAL FUNC

Fortran: **FUNCTION COM_CALL_EXIT**(IERR)
INTEGER, INTENT(IN) :: IERR

COM_CALL_ATEXIT registers a subroutine to be executed when the program terminates normally.
COM_CALL_EXIT causes the program to end and supplies a status code to the calling environment.

0.7 Advanced Window Management

0.7.1 Memory Management

Sometimes, it is more convenient to let COM allocate arrays instead of registering user-allocated arrays. This approach avoids having to duplicate the data structures of windows and panes in application codes for multi-block meshes, and is particularly beneficial for implementing complex orchestration modules. COM provides the following subroutines for memory allocation.

C++: **COM_allocate_array**(const std::string aName, int paneID=0, void **addr=NULL, int strd=0, int cap=0)

C: **COM_allocate_array**(const char *aName, int paneID=0, void **addr=NULL, int strd=0, int cap=0)

Fortran: **SUBROUTINE COM_ALLOCATE_ARRAY**(ANAME, PANEID, ADDR, STRIDE, CAP)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRIDE, CAP
<TYPE>, POINTER :: ADDR
OPTIONAL :: PANEID, ADDR, STRIDE, CAP

C++: **COM_resize_array**(const std::string aName, int paneID=0, void **addr=NULL, int stride=-1, int cap=0)

C: **COM_resize_array**(const char *aName, int paneID=0, void **addr=NULL, int stride=-1, int cap=0)

Fortran: **SUBROUTINE COM_RESIZE_ARRAY**(ANAME, PANEID, ADDR, STRIDE, CAP)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRIDE, CAP
<TYPE>, POINTER :: ADDR
OPTIONAL :: PANEID, ADDR, CAP, STRIDE

These functions take arguments similar to **COM_set_array**, except that addr is returned passed out instead of passed into the procedure. They allocate memory for a specific DataItem in a given pane if paneID is nonzero or all panes if paneID is zero (the default value). The differences between **allocate** and **resize** are that the latter allocates memory only if the array was not yet initialized, or was previously allocated by COM but the current capacity is increased or the stride is no longer the same. During resize, values of the old array will be copied automatically to the new array. If strd is -1, which is the default for **COM_resize_array**, the current value registered with COM (or the number of components if not yet registered) will be used; if strd is 0, then the number of components of the DataItem will be used. If cap is 0, then the larger of the current capacity and the number of items will be used. Note that it is an error to resize an inherited or user-allocated (i..e, not allocated by COM) DataItem. For the Fortran interface, only scalar, 1-D and 2-D pointers are allowed. If a scalar pointer is used, the data itself must be a scalar and the argument STRD and CAP must not be present. If a 1-D pointer is given, then the size of the array will be STRD*CAP. If a 2-D pointer is given, then the sizes of the array will be (STRD,CAP) if STRD is no smaller than the number of components of the DataItem (NCOMP), or be (CAP,NCOMP) if STRD is 1.

A user can use the keyword **all** in the form of "window.**all**" for aName to have COM allocate memory for all DataItems (including the mesh) in a window. The capacity must be no smaller than the size specified by **COM_set_size**; if a value smaller than the actual size is passed to **COM_resize_array**, then the actual size will be used instead.

Furthermore, using **COM_append_array**, COM provides a function to append a series of values to the end of an array associated with a pane or window DataItem that has no ghost items.

C++: **COM_append_array**(const std::string aName, int paneID, const void *addr, int strd, int size)

C: **COM_append_array**(const char *aName, int paneID, const void *addr, int strd, int size)

Fortran: **SUBROUTINE COM_APPEND_ARRAY**(ANAME, PANEID, ADDR, STRD, SIZE)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRD, SIZE
<TYPE>, INTENT(IN) :: ADDR

This function is equivalent to calling **COM_resize_array** to increase the capacity of the array if necessary using the stride currently registered with COM, calling **COM_set_size** to increase the number of items by size, and then copying data from user buffer addr with a stride strd. This function is particularly useful for packing a series of values into a big array in COM. Note that after calling **COM_append_array**, the array in COM may have been reallocated if its capacity was increased, in which case the address previously obtained from COM becomes invalid and the user must reobtain the address by calling **COM_get_array**.

Allocated memory should be deallocated by calling **COM_deallocate_array**.

C++: **COM_deallocate_array**(const std::string aName, int paneID=0)

C: **COM_deallocate_array**(const char *aName, int paneID=0)

Fortran: **SUBROUTINE COM_DEALLOCATE_ARRAY**(ANAME, PANEID)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID
OPTIONAL :: PANEID

If the deallocation routine is not called, the memory will be freed automatically when the window is destroyed.

0.7.2 Pointer DataItems

COM provides two special data types, **COM_VOID** and **COM_F90POINTER**. The former means a void pointer in C or C++, and the latter a Fortran 90 pointer. A F90 pointer is different from C/C++ pointers, in that it is a structure containing the descriptor of the data that are referenced, and the exact size of the structure is compiler dependent and may vary with the types that it references. These two data types are particularly useful in conjunction with **COM_allocate_array** to store pointers to some objects, which allows a module to eliminate global variables completely, so that they can take advantage of Charm++.

When COM allocates a F90 pointer, it allocates a piece of memory that is large enough to hold any type of F90 pointers. A F90 application code can copy a pointer to or from COM using **COM_SET_POINTER** and **COM_GET_POINTER**, respectively.

Fortran: **SUBROUTINE COM_SET_POINTER**(ATTR, PTR, ASSO)
CHARACTER(*), INTENT(IN) :: ATTR
<TYPE>, POINTER :: PTR
EXTERNAL ASSO

Fortran: **SUBROUTINE COM_GET_POINTER**(ATTR, PTR, ASSO)
CHARACTER(*), INTENT(IN) :: ATTR
<TYPE>, POINTER :: PTR
EXTERNAL ASSO

These functions are particularly useful for registering the context variable of member functions, similar to registering COM_Object associated with the C++ member functions. For that reason, COM also provides two F90 interface functions, **COM_set_object** and **COM_get_object**, which are essentially aliases of **COM_set_pointer** and **COM_get_pointer**. The argument ASSO is a user-defined subroutine which looks like follows.

```
SUBROUTINE ASSOCIATE_POINTER( attr, ptr)
   <TYPE>, POINTER  :: attr
   <TYPE>, POINTER  :: ptr

   ptr => attr
END SUBROUTINE ASSOCIATE_POINTER
```

Because the arguments of **COM_set_pointer** and **COM_get_pointer** are pointers whose types are unknown to COM, the user must explicitly define the prototypes of these functions in the application codes using the specific data types.

0.7.3 Inheritance

Inheritance is a key concept of object-oriented programming. In current *IMPACT* release, inheritance is very useful under a few situations. First, the orchestration module (*SIM*) sometimes needs to create intermediate data associated with a window owned by another module. Inheritance allows *SIM* to extend the window by adding additional DataItems, or altering the definitions of some of the DataItems. Second, a module (e.g., *Rocburn* in *Rocstar Multiphysics*) may need to operate on a subset of the mesh of another module (e.g., *Rocflo* or *Rocflu*). COM facilitates such special needs by allowing a window to inherit (a subset of) another window without incurring the memory overhead of data duplication. Furthermore, *SIM* sometimes needs to split user-defined windows into separate windows based on boundary-condition types, so that they can be handled differently (such as written into separate files for visualization).

COM supports two types of inheritance: using and cloning. For the former, COM does not duplicate the dataset; for the latter, COM does. For each type, it allows inheriting the mesh from a parent window to a child window in two modes. First, the mesh can be inherited as a whole. Second, only a subset of panes that satisfy a certain criterion are inherited. The following two subroutines support these two modes of use-inheritance, respectively.

C++:	**COM_use_dataitem**(const std::string wName_to, const std::string wName_from, int with_ghost=1, const char *aName=NULL, int val=0)
C:	**COM_use_dataitem**(const char *wName_to, const char *wName_from, int with_ghost=1, const char *aName=NULL, int val=0)
Fortran:	**SUBROUTINE COM_USE_DATAITEM**(WNAME_TO, WNAME_FROM, WITH_GHOST,ANAME, VAL) CHARACTER(*), INTENT(IN) :: WNAME_TO, WNAME_FROM, ANAME INTEGER, INTENT(IN) :: VAL, WITH_GHOST

OPTIONAL WITH_GHOST, ANAME, VAL

C++: **COM_clone_dataitem**(const std::string wName_to, const std::string wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

C: **COM_clone_dataitem**(const char *wName_to, const char *wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

Fortran: **SUBROUTINE COM_CLONE_DATAITEM**(WNAME_TO, WNAME_FROM, WITH_GHOST,ANAME, VAL)
CHARACTER(*), INTENT(IN) :: WNAME_TO, WNAME_FROM, ANAME
INTEGER, INTENT(IN) :: VAL, WITH_GHOST
OPTIONAL WITH_GHOST, ANAME, VAL

In the arguments, the wName are window names and the aName are DataItem names. The argument with_ghost indicates whether the ghost nodes and elements should be inherited. The next argument is a panel DataItem of integer type, and only the panes whose corresponding values of the DataItem equal to the argument val will be inherited. In practice, aName is most likely to correspond to a boundary-condition type for panes, and val correspond to a boundary condition ID. Note that if aName is empty (i.e., either a NULL pointer or an empty string) and val is nonzero, then condition is considered to be "paneID==val", so that only the pane whose ID is equal to val is inherited.

If a child window needs to contain panes of more than one boundary-condition types, then a user can call **COM_use_dataitem** multiple times with different boundary condition ID. Note that in both routines, the child window does not duplicate memory space for the mesh but inherit the memory addresses of the parent window. If a pane in the parent window does not exist in the target window, a new pane is inserted into the derived window if the DataItem being inherited contains the element connectivities (i.e,. "conn", "mesh", or "all"), or the pane is ignored for other types of elements. If the DataItem being inherited already exists in the child window, then the data type and layout of the new DataItem must be the same as the existing one, and COM will overwrite other information of the existing DataItem with the new DataItem. Note that one must not delete or redefine an DataItem that is being used by another window.

A related function of inheritance is **COM_copy_dataitem**, which copies data from one DataItem onto another.

C++: **COM_copy_dataitem**(const std::string wName_to, const std::string wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

C++: **COM_copy_dataitem**(const int wName_to, const int wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

C: **COM_copy_dataitem**(const char *wName_to, const char *wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

Fortran: **SUBROUTINE COM_COPY_DATAITEM**(WNAME_TO, WNAME_FROM, WITH_GHOST,ANAME, VAL)
CHARACTER(*), INTENT(IN) :: WNAME_TO, WNAME_FROM, ANAME
INTEGER, INTENT(IN) :: VAL, WITH_GHOST
OPTIONAL WITH_GHOST, ANAME, VAL

0.7.4 Deletion of Entities

When a window is not needed anymore, it should be destroyed by calling **COM_delete_window**, which takes the window name as its only argument.

C: **COM_delete_window**(const std::string wName)

C: **COM_delete_window**(const char *wName)

Fortran: **SUBROUTINE COM_DELETE_WINDOW**(WNAME)
CHARACTER(*), INTENT(IN) :: WNAME

This subroutine allows COM to clean up its internal data created for a window. It also automatically deallocates all the datasets allocated using **COM_allocate_array** or **COM_resize_array** but not yet deallocated.

Furthermore, one can delete a single pane from a window by calling **COM_delete_pane**, which takes the window name and a pane ID as its arguments.

C++: **COM_delete_pane**(const std::string wName, int pandID)

C: **COM_delete_pane**(const char *wName, int pandID)

Fortran: **SUBROUTINE COM_DELETE_PANE**(WNAME, PANEID)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(IN) :: PANEID

One can also delete an existing DataItem (except for predefined DataItems) by calling **COM_delete_dataitem**.

C++: **COM_delete_dataitem**(const std::string aName)

C: **COM_delete_dataitem**(const char *aName)

Fortran: **SUBROUTINE COM_DELETE_DATAITEM**(ANAME)
CHARACTER(*), INTENT(IN) :: ANAME

The only keyword that can be used with **COM_delete_dataitem** is "data", which will removed all user-defined DataItems and leave only the mesh. Note that after deleting some panes or DataItems, one must call **COM_window_init_done** on all processes collectively before using the window. In addition, deleting a window, pane, or DataItem may invalidate DataItem and function handles and the structure of inheritance, so they should be used with extreme care.

0.8 Information Retrieval

0.8.1 Window and panes

Typically, data registered by application modules need to be accessed only by service modules through the C++ interface described in the Developers Guide. However, some application modules (e.g., *Rocburn*) need to obtain the information about a window created by another module (e.g., *Rocflo*/*Rocflu*). *IMPACT* provides functions to support information retrieval, under the assumption that the caller knows about the DataItem names and base data types of the DataItems.

C: **COM_get_communicator**(const char *wName, MPI_Comm *comm)

Fortran: **SUBROUTINE COM_GET_COMMUNICATOR**(WNAME, COMM)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(OUT) :: COMM

This subroutine obtains the MPI communicator of a window.

The following subroutine obtains the IDs of the panes in a window local to a process:

C/C++: **COM_get_panes**(const char *wName, int *np, int **pane_ids=NULL, int rank=myrank)

C++: **COM_get_panes**(const char *wName, vector<int> &pane_ids, int rank=myrank)

Fortran:
```
SUBROUTINE COM_GET_PANES(WNAME, NP, PANE_IDS, RANK)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(OUT) :: NP
OPTIONAL, INTEGER, POINTER :: PANE_IDS(:)
OPTIONAL, INTEGER, INTENT(IN) :: RANK
```

It sets the number of panes to np and loads an array of IDs into pane_ids, whose memory is allocated by COM and should be deallocated by calling **COM_free_buffer** (except for the vector interface). The rank is in the scope of the MPI communicator of the window. If the rank is not present or is -2, then the default value is that of the current process. If the rank is -1, then the function will load the panes on all the processes within the communicator. Note that this function can only be called **after** calling **COM_window_init_done**.

C/C++: **COM_get_dataitems**(const char *wName, int *na, char **names)

C++: **COM_get_dataitems**(const char *wName, int *na, string &names)

Fortran:
```
SUBROUTINE COM_GET_DATAITEMS( WNAME, NA, NAMES)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(OUT) :: NA
CHARACTER, POINTER :: NAMES(:)
```

It sets na to be the number of DataItems in the window and allocates a space-delimited string names to store the names of the DataItems. Except for the string interface, names must be deallocated by calling **COM_free_buffer** (see below) after use.

C/C++: **COM_get_connectivities**(const char *wName, const int *pid, int *nc, char **names)

C++: **COM_get_connectivities**(const char *wName, const int *pid, int *nc, string &names)

Fortran:
```
SUBROUTINE COM_GET_CONNECTIVITIES( WNAME, PID, NC, NAMES)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(IN) :: PID
INTEGER, INTENT(OUT) :: NC
CHARACTER, POINTER :: NAMES(:)
```

It sets nc to be the number of connectivity tables in a pane and allocates a space-delimited string names to store the names of the connectivity tables. Again, names must be deallocated by calling **COM_free_buffer** (except for the string interface) after use.

C: **COM_free_buffer**(char (or int) **buf)

Fortran:
```
SUBROUTINE COM_FREE_BUFFER( BUF)
CHARACTER (or INTEGER), POINTER :: BUF(:)
```

0.8.2 DataItem and Connectivity

One can obtain the information of an DataItem by calling **COM_get_dataitem**, whose arguments correspond to those of **COM_new_dataitem**.

C: **COM_get_dataitem**(const char *aName, char *loc, COM_Type *type, int *ncomp, char *unit, int n)

C++: **COM_get_dataitem**(const char *aName, char *loc, COM_Type *type, int *ncomp, string *unit)

Fortran:
```
SUBROUTINE COM_GET_DATAITEM(ANAME, LOC, TYPE,
NCOMP, UNIT)
CHARACTER(*), INTENT(IN) :: ANAME
CHARACTER*1, INTENT(OUT) :: LOC
INTEGER, INTENT(OUT) :: TYPE, NCOMP
CHARACTER(*) :: UNIT
```

One can also use **COM_get_dataitem** on a connectivity, which will set ncomp to the number of nodes per element for that particular type of element.

One can also check the status of a window, a pane, an DataItem, or a connectivity table by calling the function **COM_get_status**.

C: int **COM_get_status**(const char *aName, int paneID)

Fortran:
```
FUNCTION COM_GET_STATUS(ANAME, PANEID)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID
INTEGER :: COM_GET_STATUS
```

If aName is a window name (i.e., containing no '.') and paneID is 0, then it checks whether the window exists, and returns 0 if so and -1 otherwise. If aName is a window name and paneID is nonzero, then it checks whether the given pane exist in the window, and returns 0 if so and -1 otherwise. If aName is in the form of "window.DataItem", then it checks the status of the given DataItem, and returns one of the following values:

- -1: does not exist;
- 0: exists but not initialized;
- 1: set by the user using set_array or set_object;
- 2: set by the user using set_array_const;
- 3: use from another DataItem;
- 4: allocated using resize_array, allocate_array, or cloned from another DataItem.

If an DataItem uses another, one can get the full name (window.DataItem) of its parent DataItem using the following interface:

C/C++: **COM_get_parent**(const char *waName, int paneid, char **parent)

C++: **COM_get_parent**(const char *waName, int paneid, string &parent)

Fortran: **SUBROUTINE COM_GET_PARENT**(WANAME, PANEID, PARENT)
CHARACTER(*), INTENT(IN) :: WANAME
INTEGER, INTENT(IN) :: PANEID
CHARACTER, POINTER :: PARENT(:)

The storage for the parent name will be allocated by COM and must be freed using **COM_free_buffer** after usage, except for the C++ interface.

0.8.3 Sizes

The following function can be used to obtain the sizes of an DataItem. Note that the size corresponds to the total size (including ghost items).

C++: **COM_get_size**(const std::string aName, int pane_id, int *size, int *ng=NULL)

C: **COM_get_size**(const char *aName, int pane_id, int *size, int *ng=NULL)

Fortran: **SUBROUTINE COM_get_size**(ANAME, PANE_ID, SIZE, NG)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANE_ID
INTEGER, INTENT(OUT) :: SIZE, NG
OPTIONAL :: NG

Note that for structured meshes, its dimensions should be obtained using **COM_get_array_const** instead of **COM_get_size**.

0.8.4 Arrays

One can obtain an array by either obtaining a pointer to the array, or copying the data into a user provided buffer. The first mode is provided by **COM_get_array** and **COM_get_array_const**, which can be used to obtain a pointer to an array registered or allocated in *COM*.

C: **COM_get_array**(const char *aName, int paneID, void **addr,
int *strd=NULL, int *cap=NULL)

Fortran: **SUBROUTINE COM_GET_ARRAY**(ANAME, PANEID, ADDR,
STRD, CAP)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID
<TYPE>, POINTER :: ADDR
INTEGER, INTENT(OUT) :: STRD, CAP
OPTIONAL :: STRD, CAP

C: **COM_get_array_const**(...)

Fortran: **SUBROUTINE COM_GET_ARRAY_CONST**(...)

Note that if the DataItem-name and the pane-ID do not identify a unique array, then a NULL pointer will be returned. Furthermore, if an array was registered with **COM_set_array_const**, then it can be retrieved only by **COM_get_array_const**. For the Fortran interface, only scalar, 1-D and 2-D pointers are allowed. If a scalar pointer is used, the data itself must be a scalar and the argument STRD and CAP must not be present. If a 1-D pointer is given, then the size of the array will be STRD*CAP. If a 2-D pointer is given, then the sizes of the array will be (STRD,CAP) if STRD is no smaller than the number of components of the DataItem (NCOMP), or be (CAP,NCOMP) if STRD is 1.

The second mode is provided by **COM_copy_array**.

C: **COM_copy_array**(const char *aName, int paneID, void *val, int v_strd=0, int v_size=0, int offset=0)

Fortran: **SUBROUTINE COM_GET_ARRAY**(ANAME, PANEID, VAL, V_STRD, V_SIZE, OFFSET)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, V_STRD, V_SIZE, OFFSET
<TYPE>, POINTER :: VAL
OPTIONAL :: V_STRD, V_SIZE, OFFSET

It copies up to v_size items of the DataItem starting from the offset-th item of the DataItem into the given buffer with stride v_strd. If v_strd=0 (the default value), then the number of components will be used as the stride. If v_size=0 (the default value), then the number of items will be used as the size. The default value of offset is 0. Note that a runtime error occurs if offset is negative or offset+v_size is larger than the actual capacity of the DataItem.

0.8.5 Bounds

The lower and upper bounds of a specific DataItem can be obtained by calling **COM_get_bounds**.

C: **COM_get_bounds**(const char *aName, int pane_id, void *lbnd, void *ubnd, int is_soft=0)

Fortran: **SUBROUTINE COM_get_bounds**(ANAME, PANE_ID, LBND, UBND, IS_HARD)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANE_ID
<TYPE>, INTENT(OUT) :: LBND, UBND
INTEGER, INTENT(IN), OPTIONAL :: IS_SOFT

Furthermore, COM also provides functions to check the DataItems against pre-set bounds.

C: int **COM_check_bounds**(const char *aName, int pane_id, int nprint=0)

Fortran: INTEGER **FUNCTION COM_check_bounds**(ANAME, PANE_ID, NPRINT)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANE_ID
INTEGER, INTENT(IN), OPTIONAL :: NPRINT

If pane_id is 0, then the bounds will be checked on all panes; otherwise, they will be checked only on the pane with the given pane ID. If there are only soft-bound violations or no violations, then the function returns the number of soft-bound violations. If the verbose level of COM is 0, then no information will be printed. If the verbose level is nonzero and nprint is 0, then a summary of soft-bound violations will be printed. If nprint is greater than 0, then the first few (where the number to be printed is nprint) soft-bound violations for each DataItem in each individual pane will be printed. A violation of hard bounds will terminate the execution of the code, and a summary of hard-bound violations will be printed upon termination, along with information about any soft-bound violations if the verbose level is nonzero.

0.9 Sample Codes

A few sample application codes of COM are provided in the test subdirectories of a few service modules. In particular, sample codes are available in Simpal/test and SurfMap/test.