UCRL-SM-236144
Version 2.1

Mili 1.0 I/O Library

A Mesh I/O Library

Programmer's Reference Manual

Ivan R. Corey
Elsie Pierce
Doug Speck

January 08, 2008

Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA 94550

Disclaimers

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Table of Contents

Revision History

Document Version	Revision Date	Originator(s)	Revision Description
1.0	April 24, 2003	Doug Speck	Initial version
2.0	October 25, 2007	I. R. Corey, Elsie Pierce	Updated with Surface Class and TI features
2.1	January 08, 2008	I. R. Corey,	Added Read&Write Examples

1.0 INTRODUCTION

Mili is a high-level **m**esh **I**/O **li**brary intended to support computational analysis and post-processing on unstructured finite element meshes. The support that Mili provides for finite element analysis manifests itself in two primary ways: self-describing databases with high-level, finite element analysis-cognizant semantics, and cross-platform data portability. Mili provides analysis codes with the capability to completely describe and label the state data, which will be output to disk. This descriptive information is encoded in state record descriptors. Mili allows applications to define an arbitrary number of these descriptors, engendering the ability to change output data formats on a state-by-state basis. Mili has both C and FORTRAN interfaces and provides for database portability by supporting both big- and little-endian binary formats while permitting applications to target either when creating a new database. Although much of Mili's application programming interface (API) is devoted to managing state data, there are also several general-purpose calls to write arbitrary scalars, arrays, and text data into a database.

A Mili database is actually a family of files identified by a common root prefix in each filename. The root name is specified by the application when the family is opened. Creation and naming of individual files in the family is managed entirely by Mili, although the application can set some parameters, which affect this process. Although the internal organization of a family is intended to be opaque to users, it is useful to know that Mili families have two branches - one for state data, the other for non-state data. "State" data to Mili is that data described by state record formats and written with the state data output calls. State data filenames consist of the root name followed by a numeric suffix (actually a zero-based file count, i.e., "test00", "test01", etc.). All other data is "non-state" data as far as Mili is concerned, although arbitrary numeric data written by an application can certainly be state-dependent in the context of the application. Non-state data filenames consist of the root name followed by an alphabetic suffix (this suffix is a zero-based base-26 file count using upper case letters as digits, i.e., "testA", "testB",...,"testZ", "testBA", etc.).

2.0 OVERVIEW

When an application creates a Mili database, it sets some overall parameters for the databases that are maintained for all subsequent output operations to the database. These parameters include the endianness of the data stored in the database (which defaults to the endianness of the host computer) and a precision limit which sets an overall cap on the number of bytes used to store numeric data in a database (four or eight bytes, defaulting to eight). The precision limit is essentially a convenience feature to permit double-precision applications to offload responsibility to Mili for converting data to a lower precision as it is written to disk. Similarly, by explicitly setting the endianness of a database, an application can target a preferred platform for subsequent post-processing of a database and offload responsibility to Mili for converting the endianness of output data. Setting the endianness of a database does not preclude subsequent processing on

any platform since Mili will also automatically convert data to the native endianness of the host platform when reading a database.

Mili provides for limited concurrent access to a database. Any number of processes can have read access on a database, but only one process can have write or append access on a database. This concurrency is implemented through the use of a UNIX file lock applied to a temporary "lock" file which Mili creates for the process which successfully obtains write (append) access.

With the 1.10 release of Mili file locking can now be disabled (*mf_filelock_disable*) since there it has limited application in the environments where Mili is being used. There has also been problems with the UNIX file locking implementation on some parallel platforms.

Mili manages state data through the use of state record formats. Applications define one or more state record formats, which describe the contents and layout of data, as it will be handed off to Mili during the analysis. Prior to writing data for a state, the application initializes the state by declaring its simulation time and the state record format. Thus, although each state record format describes a fixed length entity, the amount and layout of data written can vary from state to state by specifying different formats.

Mili defines a set of object "superclasses" for which an analysis code might produce state data. The simplest of these is the "unit" superclass, which has no semantics or topology implicitly associated with it and so acts merely as an organizing placeholder for grouping data. The unit superclass might be used to represent data for particles, for example, which are part of a simulation but topologically free of an enclosing mesh. Another superclass is the mesh itself, with which global state data can be associated. There also exists a material superclass for associating state data on a material-by-material basis. The rest of the superclasses derive from mesh nodes and common finite elements. The element superclasses carry with them no explicit qualities other than their names and nodal quantities (although a specific nodal order must be observed when actually specifying element connectivities), but a topology is implied so that applications operating on the data can interpret it correctly. For example, both the **`m_quad`** and **`m_tet`** superclasses require four nodal connectivities. The **`m_quad`** superclass is intended to represent "flat" quad or shell elements, though, while the **`m_tet`** superclass is intended to represent three-dimensional tetrahedral elements. A post-processor would need to distinguish between the two superclasses in order to correctly conduct calculations on the elements or to render them. The complete set of object superclasses is summarized in Table 1.

Class name	Node Quantity	Object type
m_unit	n/a	Arbitrary
m_node	1	Node
m_truss	2	Truss
m_beam	3	Oriented beam
m_tri	3	Triangle
m_quad	4	Quadrilateral or Shell
m_tet	4	Tetrahedron
m_pyramid	5	Pyramid
m_wedge	6	Wedge or Prism
m_hex	8	Hexahedron or Brick
m_mat	n/a	Material
m_mesh	n/a	Mesh
m_surface	n/a	Mesh

Table 1. Mili object superclasses

Mili superclasses are not used directly in a meaningful way. Rather, mesh object classes are defined as named descendants of the superclasses and are themselves utilized. In this way, multiple instances of each superclass can be created and treated completely independently.

The primary example of this is the use of element classes. By using element classes derived from superclasses, a Mili database can support a mesh model with multiple independent groups of elements, each numbered globally one-to-n, which share a common nodal topology. For example, a finite element analysis code might support both hexahedral volume elements and thick shell elements, both defined by eight nodal connectivities and, therefore, having a Mili superclass of **`m_hex`**. Without superclass specialization, Mili would have no way of distinguishing the two groups of eight-node elements and they would have to be lumped together (by the analysis code) into one globally numbered group of elements. With specialization, the volume elements could go into a "Brick" class, for example, and the thick shells could go into a "Thick Shell" class, allowing them to be differentiated for output and subsequent post-processing as two one-to-n globally numbered groups of elements.

A more common occurrence than having multiple element classes of a single superclass, and one of the primary motivations for developing Mili, is a mesh in which a single class of elements contains elements of different materials. Since different computational models may support these materials, there may be different sets of state variables available by material. Mili allows elements of a single class to be subdivided on output for the purpose of customizing the set of

output state variables. While material distinctions are a natural application for such subdivision, the criteria actually used to control the subdivision are entirely up to the application programmer.

An important aspect of a Mili database is the overall organization of state data with respect to the mesh objects for which the data is being output. Mili state data is organized as either "object-ordered" or "result-ordered". With object-ordered state data, the state variables for each individual object are physically adjacent, forming an object "vector". With result-ordered state data, all instances of a state variable from all contributing objects are physically adjacent, forming a result array, which has one entry for each object. Figure 1 graphically illustrates these orthogonal data organizations for a collection of four elements and two state variables (The ascending number sequences of Figure 1 are only for illustration. Mili imposes no constraints on the order of variables or mesh objects in record formats). Note that the physical organization of state data for the mesh itself (i.e., global state data) is actually the same whether specified as object-ordered or result-ordered. For the superclass **`m_mesh`** there can only be one object (the mesh) so the result arrays in a result-ordered state record would each have only one entry, and

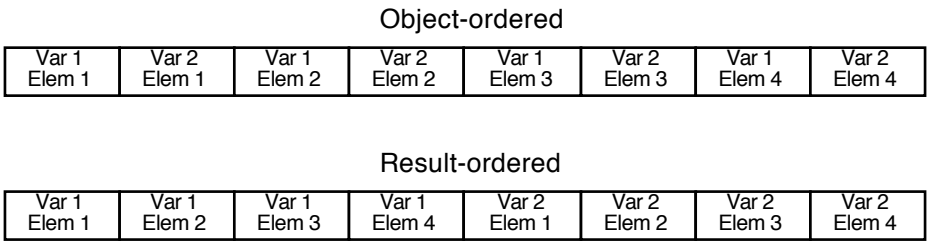


Figure 1. State record organization variations

the set of contiguous result arrays would be identical to the single object vector from an object-ordered state record. Mili does not perform organization translations on output, so applications are responsible for ordering the state data properly before handing it off to Mili. Mili does perform organization translations on input, so applications can, for example, request result-ordered data from object-ordered file records. The default organization is object-ordered.

2.1 Formatting State Data

The most complex task of creating a Mili database is formatting the state data. This three-step process includes defining the state variables, defining the mesh geometry and mesh object classes, and defining state record formats. These steps are described in more detail below. The only order imposed on these operations is that state record formats can only be defined after state variables and mesh geometry have been defined.

2.1.1 State Variable Definition

The process of defining the state variables creates a dictionary of definitions. A definition includes everything necessary to read, write, label, and otherwise operate on a given state variable. A key point to remember about a state variable definition is that it carries with it no information about an actual state record format, just as a word definition in a dictionary does not

dictate which sentences might employ the word. In this way, state variable definitions can be re-used in multiple state record formats.

There are four fundamental types of state variables in Mili: scalars, vectors, arrays, and vector arrays. These types comprise the possible values of a state variable descriptive parameter known as the aggregate type. A scalar is simply a single floating point or integer value. A vector is just a one-dimensional array in which the individual elements are referenced by a name rather than an array index. An array is an n-dimensional, contiguous collection of scalars in which individual elements are referenced by one or more indices. A vector array is an array in which the individual elements are vectors, with each vector field being referenced by name, not index value (although a particular vector, being an element of an array, is itself indexed). Computationally, any data that can be represented as an array can be represented as a vector array with one fewer array dimension and a vector size equal to the most minor dimension of the array. Likewise any vector array can be represented as a regular array with one additional minor dimension that dimension having a size equal to the vector length.

Regardless of aggregate type, state variables all have three pieces of descriptive information in common: a numeric type (floating point or integer), a short name, and a long name. The short name is intended to be a concise, mnemonic label, which is easy to manipulate (such as when entering it on a command line). The long name, or title, is intended to be used as a label for menus and graphic output from a post-processor. Array and vector array state variables also have integer descriptors describing the number of dimensions and the size of each dimension, and vectors and vector arrays require the vector size plus short names and long names for each vector field.

Short names are particularly important in state variable definitions because all short names are entered into a common name space *and must be unique*. Since state variable definitions can be used repeatedly to format multiple state records, this should not present a significant constraint to applications. The constraint, as it exists, is that a variable with a given short name can have only one long name, one numeric type, and one aggregate type.

2.1.2 Mesh Object Class and Geometry Definition

The goals of this step are to define the mesh geometry and to identify all objects, which make up the mesh so that state data can be output for them. Before any identification of actual mesh objects can occur, their object class names must be declared and associated with a particular mesh and object superclass. Classes are thus instantiations of superclasses on a particular mesh. A single Mili database can have multiple meshes defined within it, and identical object class names can be defined for each mesh if desired.

“Identification” means different things for objects belonging to different superclasses. In particular, mesh nodes (superclass **`m_node`**) are “identified” by numbering them and by specifying initial coordinates, and mesh elements (derived from various superclasses depending on element topology) are identified by numbering them and by specifying nodal connectivities; these steps, taken together, define the mesh geometry. Objects derived from the other superclasses (**`m_unit`**, **`m_mat`**, and **`m_mesh`**) lack the specialized requirements of nodes and elements and are simpler to define. Objects derived from those superclasses must only be

numbered. The Mili API provides specialized calls for identifying and defining mesh objects attendant to the requirements of the particular superclasses.

As was stated above, the mesh geometry is defined by declaring all nodal coordinates and element connectivities. Each of these straightforward tasks can be done in a piecemeal fashion, relieving the application from the necessity of having all nodal coordinates or element connectivities assembled in memory at once. As no checking is performed to verify that nodes referenced by element connectivities have themselves been defined, the two tasks can be performed in any order, or be interleaved.

Material and part numbers are specified for elements at the same time their connectivities are declared. At this time, there is no explicit linkage between the material numbers specified during element definition and material identifiers that would be declared for an object class derived from the **`m_mat`** superclass. However, this association is intended and may become explicit in future versions of the library, so it would be prudent for application programmers to create the material class and identify its members prior to referencing material numbers during element definition.

2.1.3 State Record Format Definition

Having defined a set of state variables and the mesh object classes, the fundamental Mili task of defining state record formats by associating sets of state variables with sets of mesh objects can proceed. In Mili, a state record format is never actually created as a complete entity in one action. Rather, an empty format descriptor is opened and then filled by defining one or more subrecords for the state record. Defining a subrecord accomplishes two important tasks. First, it binds a set of state variables to a set of mesh objects (nodes, elements, materials, etc.). This binding declares what state data will be output for which objects in each output state. Second, it specifies the order of the data in the subrecord. This order is controlled by the order of specification of both state variables and objects in the call to define the subrecord and by the gross organization, object-ordered or result-ordered, of the subrecord. The overall ordering of subrecords in the state record is determined by the order of calls to define the subrecords.

Mili places no limit on the number of state record formats, which are created. When writing out actual state data, the application must declare a state record format identifier with each new state so that Mili (and any database post-processing application) knows which format to apply while operating on the state data.

2.2 Mili API Utilization to Create a Finite Element Simulation Database

This section illustrates the basic programming steps (and pertinent function calls) necessary to generate a simulation database using Mili. Aside from opening and closing a database, the basic steps can be loosely categorized into two phases: definition and data output. In typical use these phases would be non-overlapping, but Mili does not require this. An application could, for example, execute the definition operations through definition of a state record format, then proceed to write data using that format, then define a new format and write more data in that format, etc. The golden rule is that things (elements, meshes, state record formats, etc.) must be

defined before they are used. Mili databases can be re-opened and appended to, so the process(es) creating a Mili database need not be persistent.

Figure 2 illustrates a hierarchy, in tree form, of programming steps to create a Mili state database. The tree has FORTRAN API function calls at its leaves (C API usage would be identical, i.e., just substitute "mc" for "mf" in the routine names). The text above each subtree describes what is accomplished on that subtree. By traversing the tree in a depth-first, left-to-right order, the leaves as they're visited generate the intended sequence of Mili library calls. The left-to-right sequence of activities on any given horizontal span can be looped over an arbitrary number of times depending on the needs of the application and, in particular, the mesh(es) being defined for the database. Where multiple function calls exist on a given leaf, no order is implied regarding their use. All calls illustrated may not be necessary depending on the structure of the mesh and the data being output.

This tree generates only one of an endless number of possible call sequences, but it should serve as an excellent starting point for most applications.

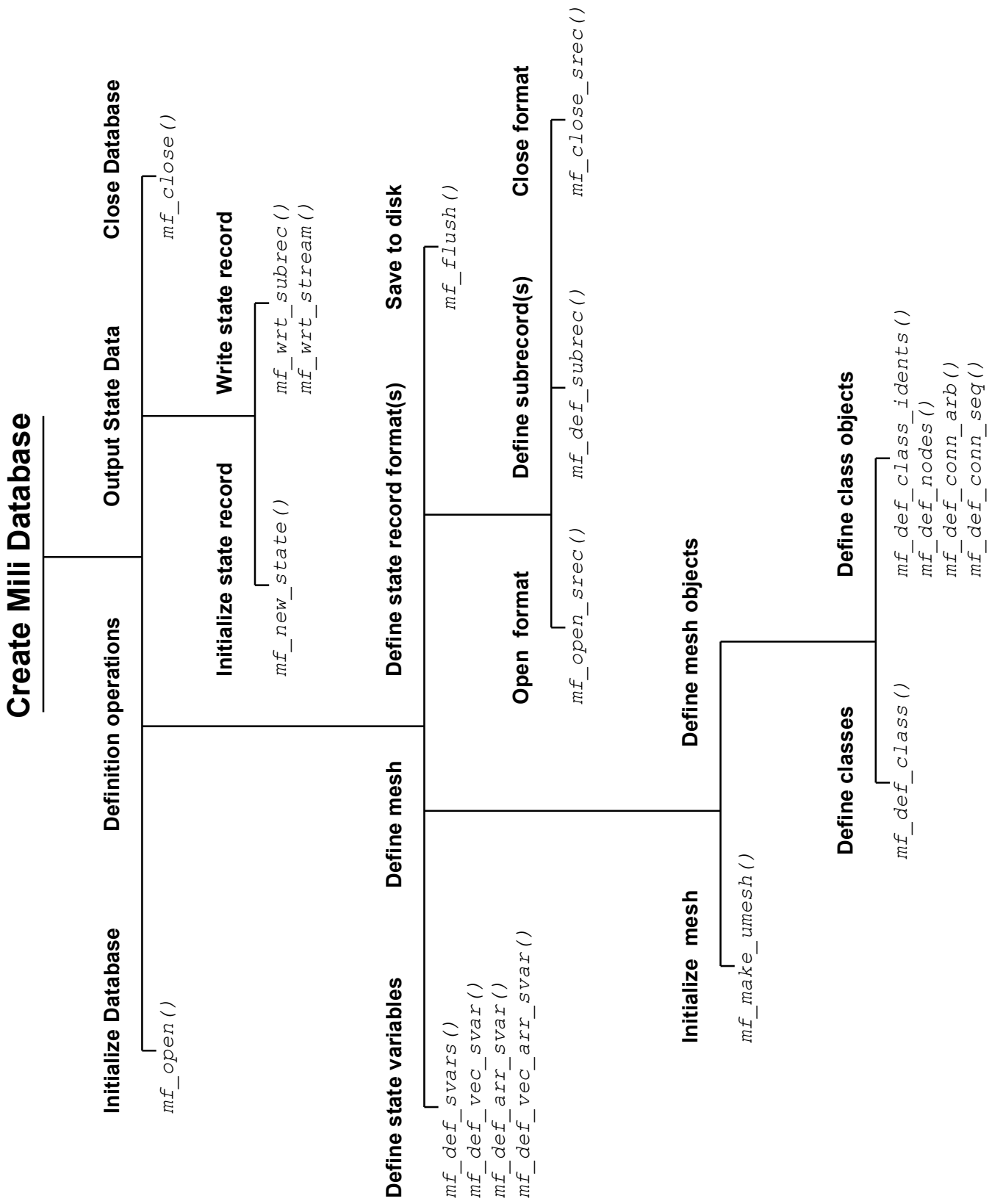


Figure 2. Calling heirarchy to create a Mili database

2.2.1 An Example

See *Appendix B* for a detailed example on writing and reading a Mili Database.

2.3 Time Independent Data

Mill provides a variety of functions to read and write time independent data or TI data. TI data are fields that are not associated with any particular state and they are kept in a file ending in "_TI" (for example test_TI). The following types of TI data are supported: scalars of int, float, double, and character strings, and arrays of int, float, double, and character strings. TI variables are identified by name and are associated with a superclass.

2.4 Surface class objects

A surface class is a collection of faces that are grouped and named. Each surface consists of a set of faces that can have multiple results attached to them. Mili now provides functions for defining and loading the surface connectivity.

2.5 Tips and Tools for Developing with Mili

There are several mechanisms available to provide development and debugging assistance to the application developer using Mili. First is the evaluation, within the calling application, of the status returned from each Mili call. With very few exceptions, a non-zero status value is an indication that Mili detected an error condition. Mili provides a function, **`mc_print_error`**(), (or **`mf_print_error`**() in Fortran), that takes as input a status value returned from any Mili call. If the status is non-zero, Mili will print a diagnostic to the UNIX standard error stream (stderr). The application may optionally provide a text string to **`mc_print_error`**(), the contents of which will be prepended to the Mili diagnostic if one is generated, as a further means of identifying the origin of the call that returned an error condition.

There are a few locations in the code where Mili detects unusual or potentially erroneous conditions, but which don't lend themselves to communication via a return status. In these cases, Mili writes a message directly to stderr, but only if an environment variable "MILI_VERBOSE" has been defined at the time the database is opened. Note that this functionality is not enabled per database. It is turned on for all subsequent Mili activity by the process, regardless of the number of databases in use, whenever MILI_VERBOSE exists at the time that any Mili database is opened; it is similarly turned off for all subsequent Mili activity when MILI_VERBOSE does not exist when any database is opened.

A third tool for development support is the Mili non-state data dump utility, "md". The md utility is built when the Mili library is compiled. Md takes a Mili database as input and unpacks the mesh description and other non-state data, writing it to stdout as a formatted text listing. Various options are available to constrain the output. Executing md without any arguments causes it to write out a help listing of the available options.

3.0 MILI FORTRAN API

mf_close()

Close an open Mili file family.

Synopsis:

```
mf_close( fam_id, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

If `fam_id` is invalid, `status` is returned non-zero.

See also:

C API: `mc_close()`

mf_close_srec()

Close an open state record format definition, precluding the addition of further subrecords in the format and marking the format as available for output to disk at the next non-state data flush or database close.

Synopsis:

```
mf_close_srec( fam_id, srec_id, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`srec_id`	Input	INTEGER	State record format identifier.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

See also:

C API: `mc_close_srec()`

mf_def_arr_svar()

Define an array state variable.

Synopsis:

```
mf_def_arr_svar( fam_id, type, order, dimensions, name,
                 title, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`type`	Input	INTEGER	Mili data type. Possible values: **`m_real`**, **`m_int`**
`order`	Input	INTEGER	Number of array dimensions (must be less than or equal to **`m_max_array_dims`**).
`dimensions`	Input	INTEGER array	Array of dimension sizes for the state variable array being defined. Size: (`order`)
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	State variable short name.
`title`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	State variable long name.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

`name` should be a brief, mnemonic handle for the array.
`title` should be a handle suitable for labeling a plot or menu.

All state variable (scalar or array) short names must be unique. An attempt to re-enter an extant name will cause `mf_def_arr_svar()` to fail.

Although each state variable is unique, each one can be referenced by any number of subrecord definitions (see `mf_def_subrec()`).

Example:

```
integer fam_id, status
integer dimensions(3)
data dimensions / 3, 3, 3 /

! Open family...
```

```
      mf_def_arr_svar( fam_id, m_real, 3, dimensions, 'ev',
     +                 'Strain Vector', status )
      mf_print_err( 'mf_def_arr_svar in myprog', status )
```

See also:

C API: `mc_def_arr_svar()`
File mili_fparam.h

mf_def_class()

Define a mesh object class from a superclass.

Synopsis:

```
mf_def_class( fam_id, mesh_id, superclass, short_name,
              long_name, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh identifier.
`superclass`	Input	INTEGER	Mesh object superclass to which new class will belong.
`short_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Short name of class being defined.
`long_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Long name (title) of class being defined.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

A class must be defined before any operation, which references the class, can occur. Class references occur during geometry definition with calls to `mf_def_nodes()`, `mf_def_conn_arb()`, and `mf_def_conn_seq()`, during state record format definition with calls to `mf_def_subrec()`, and during information query calls to `mf_query_family()`.

Each class short name must be unique for a particular mesh, but if multiple meshes exist within a single Mili database family, short names can be reused by more than one mesh. The intent of having both a short name and long name is to provide a string which is convenient to type on a command line plus a string that is suitable as a menu title or label in rendered output.

Example:

See also:

C API: `mc_def_class()`
File mili_fparam.h

mf_def_class_idents()

Define a continuous sequence of object identifiers for a mesh object class.

Synopsis:

```
mf_def_class_idents( fam_id, mesh_id, class_name, start,
                     stop, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh identifier.
`class_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of mesh object class; must have been previously defined with a call to `mf_def_class()`.
`start`	Input	INTEGER	Identifier of first object in sequence.
`stop`	Input	INTEGER	Identifier of last object in sequence.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

In order to bind state variables to mesh objects during state record format definition, it is necessary to have identified the objects being bound. Routine `mf_def_class_idents()` exists to define objects for classes, which are not of necessity defined elsewhere. As of this writing, this would be objects in classes derived from the superclasses **`m_unit`**, **`m_mat`**, and **`m_mesh`**. Other objects are defined via specialized routines that reflect the particular semantics of the objects being defined, i.e., nodes are defined with calls to `mf_def_nodes()`, during which initial nodal coordinates are also defined. Similarly, elements are defined using subroutines `mf_def_conn_arb()` and `mf_def_conn_seq()`, during which nodal connectivities are identified.

Although there is no explicit linkage within Mili, identifiers for a class derived from **`m_mat`**, i.e., material numbers, should exactly correspond to material numbers associated with elements in calls to `mf_def_conn_arb()` and `mf_def_conn_seq()`, as this would be the intended association of state variables bound to **`m_mat`** objects during state record format definition.

It is not necessary to identify all objects for a class in a single call to `mf_def_class_idents()`.

Example:

See also:

C API: `mc_def_class_idents()`
File mili_fparam.h

mf_def_conn_arb()

Define element connectivities for an arbitrary list of elements in an unstructured mesh.

Synopsis:

```
mf_def_conn_arb( fam_id, mesh_id, class, qty, elem_ids,
                 connects, status )
```

Arguments:

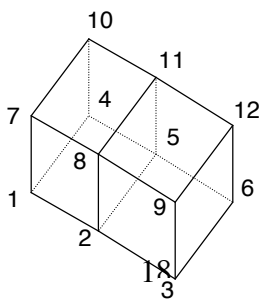
`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh identifier.
`class`	Input	CHARACTER*n ($n \le$ **`m_max_name_len`**)	Name of element class. Must have been previously defined with a call to `mf_def_class()`. The superclass associated with `class` must be one of the *element* mesh object types.
`qty`	Input	INTEGER	Number of elements referenced in array `elem_ids`.
`elem_ids`	Input	INTEGER array	Array of element identifiers. Size: (`qty`)
`connects`	Input	INTEGER array	Array of element connectivities, including material numbers as last field for each row. Size: (m,`qty`) where m is dependent on the mesh object superclass associated with parameter `class`.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Subroutine `mf_def_conn_arb()` is identical to subroutine `mf_def_conn_seq()` except for the way in which element identifiers are specified. In `mf_def_conn_arb()`, the elements are listed explicitly in array `elem_ids` and may be randomly ordered. See `mf_def_conn_seq()` for additional discussion of element connectivity definition.

Example:

Given the following two hex-element mesh with node numbers as shown:



```
      integer fam_id, mesh_id, status
      integer elem_ids
      data elem_ids / 1, 2 /
      integer connects(9,2)
      data connects / 1, 2, 5, 4, 7, 8, 11, 10, 1,
     +                2, 3, 6, 5, 8, 9, 12, 11, 1 /
      character*(m_max_name_len) elem_class
      data elem_class / 'Bricks' /

      ! Open Mili family, make mesh, define nodes...

      mf_def_class( fam_id, mesh_id, m_hex, elem_class )
      mf_def_conn_arb( fam_id, mesh_id, elem_class, 2,
     +                 elem_ids, connects, status )
      mf_print_err( 'mf_def_conn_arb in myprog', status )
```

See also:

`mf_def_conn_seq()`
C API: `mc_def_conn_arb()`
File mili_fparam.h

mf_def_conn_labels()

Define element labels (one label element) for the specified element class.

Synopsis:

```
mf_def_conn_labels( fam_id, mesh_id, class, qty, labels,
status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh identifier.
`class`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of element class. Must have been previously defined with a call to `mf_def_class()`. The superclass associated with `class` must be one of the *element* mesh object types.
`qty`	Input	INTEGER	Number of labels to write.
`labels`	Input	INTEGER array	Array of element labels. Size: (m,`qty`) where m is dependent on the mesh object superclass associated with parameter `class`.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Example:

Given the following two hex-element mesh with node numbers as shown:

```
integer fam_id, mesh_id, status
integer labels(2)
data la / 10, 20 /
character*(m_max_name_len) elem_class
data elem_class / 'Bricks' /

! Open Mili family, make mesh, define nodes...

mf_def_class( fam_id, mesh_id, m_hex, elem_class )
mf_def_conn_labels( fam_id, mesh_id, elem_class,
+                   labels, status )
mf_print_err( 'mf_def_conn_labels in myprog', status )
```

See also:

`mf_def_conn_seq()`
`mf_def_node_labels()`
`mf_load_conn_labels()`
C API: `mc_def_conn_labels()`
File mili_fparam.h

mf_def_conn_seq()

Define element connectivities for a continuously numbered sequence of elements in an unstructured mesh.

Synopsis:

```
mf_def_conn_seq( fam_id, mesh_id, class, start_elem,
                 stop_elem, connects, status )
```

Arguments:

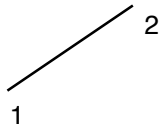
`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh identifier.
`class`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of element class. Must have been previously defined with a call to `mf_def_class()`. The superclass associated with `class` must be one of the *element* mesh object types.
`start_elem`	Input	INTEGER	Element number of first element in sequence.
`stop_elem`	Input	INTEGER	Element number of last element in sequence.
`connects`	Input	INTEGER array	Array of element connectivities plus material numbers and part numbers. Size: (m,`qty`) where m is dependent on the mesh object superclass associated with parameter `class`.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

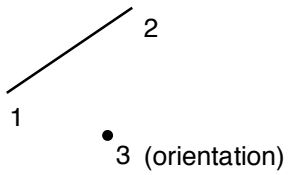
The purpose of subroutine `mf_def_conn_seq()` is to specify element connectivities for an element class. Element classes provide a way of having separately numbered (i.e., one-to-n) groups of elements, which share the same element superclass (**`m_hex`**, for example). This allows Mili to independently support new element models, which share a nodal topology with other elements. All elements must belong to a named class, even if there is only one element type in the mesh. Connectivities for the elements of a single class may be defined across multiple calls to `mf_def_conn_seq()`.

The array `connects` contains one row for each node referenced by the element plus two additional rows for the element material number and a part number. Array `connects` contains one column for each element being defined in the call. The nodes for each element must be referenced in a particular order as follows:

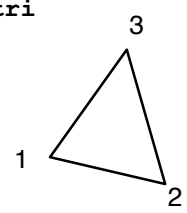
m_truss



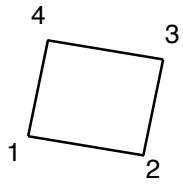
m_beam



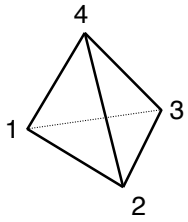
m_tri



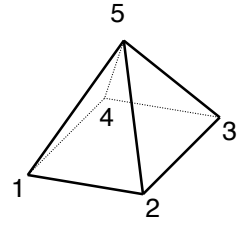
m_quad



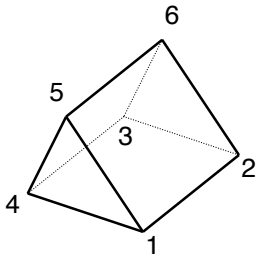
m_tet



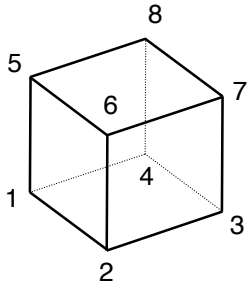
m_pyramid



m_wedge



m_hex



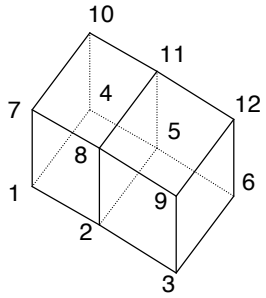
Allowing for material number and part number as the last fields for each element, then, the number of rows in array `connects` by element mesh object superclass is as follows:

m_truss	4
m_beam	5
m_tri	5
m_quad	6
m_tet	6
m_pyramid	7

m_wedge	8
m_hex	10

Example:

Given the following two hex-element mesh with node numbers as shown:




```
      integer fam_id, mesh_id, status
      integer connects(10,2)
      data connects / 1, 2, 5, 4, 7, 8, 11, 10, 1, 1,
     +                2, 3, 6, 5, 8, 9, 12, 11, 1, 1 /
      character*(m_max_name_len) elem_class
      data elem_class / 'Bricks' /

      ! Open Mili family, make mesh, define nodes...

      mf_def_class( fam_id, mesh_id, m_hex, elem_class )
      mf_def_conn_seq( fam_id, mesh_id, m_hex, elem_class, 1,
     +                 2, connects, status )
      mf_print_err( 'mf_def_conn_seq in myprog', status )
```

See also:

`mf_def_conn_arb()`

C API: `mc_def_conn_seq()`

mf_def_conn_surface()

Defines the connectivity via a list of node numbers for a new surface – the id number for the New surface is returned.

Synopsis:

```
mf_def_conn_surface( fam_id, mesh_id, short_name,
                     qty_facets, surf_data, surf_id,
                     status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh identifier.
`short_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Surface object class name.
`qty_facets`	Input	INTEGER	Number of facets in this surface.
`surf_data`	Output	Integer array	Array of nodal connection numbers.
`surf_id`	Output	Integer	Surface id number.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

See also:

`mf_load_surface()`
C API: `mc_def_conn_surface()`

mf_def_nodes()

Define initial coordinates for a continuously numbered sequence of nodes in an unstructured mesh.

Synopsis:

```
mf_def_nodes( fam_id, mesh_id, class_name, start_node,
              stop_node, coords, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Unstructured mesh identifier
`class_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of node class. Must have been previously defined with a call to `mf_def_class()`. The superclass associated with `class` must be `m_node`.
`start_node`	Input	INTEGER	Number of first node in sequence.
`stop_node`	Input	INTEGER	Number of last node in sequence. Should be greater than or equal to `start_node`.
`coords`	Input	REAL array	Array of nodal coordinates. Size: (`dims`,*qty*) where *qty* = `stop_node` - `start_node` + 1 and `dims` is either two or three, as defined in preceding call to `mf_make_umesh()` which returned `mesh_id`.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Initial nodal coordinates for a mesh can be defined in one or multiple calls to `mf_def_nodes()`. While nodes specified in an individual call to `mf_def_nodes()` must be continuous and ascending, there is no ordering constraint between groups of nodes across multiple calls to `mf_def_nodes()`. An application may overwrite previously written coordinates by calling `mf_def_nodes()` with values of `start_node` and `stop_node`, which identify a subset of nodes written in a single previous call to `mf_def_nodes()`.

Note that, since the application defines the class, some convention must be established to associate connectivities for mesh elements (see `mf_def_conn_arb()` and `mf_def_conn_seq()`) with a particular node class. For example, an application could require that at least one node class, of a particular name, exist to physically instantiate the connectivities of all elements in a mesh.

Defining nodal coordinates is one of two necessary steps to defining an unstructured mesh (the other step being to define the element connectivities). There is no crosschecking between these two tasks so the relative order in which they are completed is not constrained by Mili.

Example:

```
      integer fam_id, mesh_id, status
      real coords(3,8)
      data coords / 0.0, 0.0, 0.0,  1.0, 0.0, 0.0,  0.0, 1.0,
      0.0,
     +              1.0, 1.0, 0.0,  0.0, 0.0, 1.0,  1.0, 0.0, 1.0,
     +              0.0, 1.0, 1.0,  1.0, 1.0, 1.0 /
      character*(m_max_name_len) node_class
      data node_class / 'Nodal' /

      ! Open Mili family...

      mf_make_umesh( fam_id, 'Test mesh', 3, mesh_id, status )
      mf_def_class( fam_id, mesh_id, m_node, node_class )
      mf_def_nodes( fam_id, mesh_id, node_class, 1, 8, coords,
     +              status )
      mf_print_err( 'mf_def_nodes in myprog', status )
```

See also:

C API: `mc_def_nodes()`

mf_def_node_labels()

Define nodal labels.

Synopsis:

```
mf_def_node_labels( fam_id, mesh_id, class_name, qty,
                    labels, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Unstructured mesh identifier
`class_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of node class. Must have been previously defined with a call to `mf_def_class()`. The superclass associated with `class` must be `m_node`.
`qty`	Input	INTEGER	Number of labels to write.
`labels`	Input	INTEGER array	Array of nodal labels. Size: Number of nodes in the class.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Nodal labels are optional.

Example:

```
integer fam_id, mesh_id, status
integer labels(4)
data labels / 20, 30, 40, 50 /

character*(m_max_name_len) node_class
data node_class / 'Nodal' /

! Open Mili family...

mf_make_umesh( fam_id, 'Test mesh', 3, mesh_id, status )
mf_def_class( fam_id, mesh_id, m_node, node_class )
mf_def_node_labels( fam_id, mesh_id, node_class, labels,
+                   status )
mf_print_err( 'mf_def_node_labels in myprog', status )
```

See also:

`mf_def_conn_labels()`, `mf_load_conn_labels()`
`mf_load_node_labels()`

C API: `mc_def_node_labels()`

mf_def_subrec()

Define a state subrecord by binding state variables to mesh objects.

Synopsis:

```
mf_def_subrec( fam_id, srec_id, name, org, qty_st_vars,
               st_var_names, class, format, qty,
               object_ids, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`srec_id`	Input	INTEGER	State record format identifier from `mf_open_srec()`.
`name`	Input	CHARACTER*n ($n \le$ **`m_max_name_len`**)	Name (label) for subrecord.
`org`	Input	INTEGER	Organization of data. Possible values are **`m_object_ordered`** or **`m_result_ordered`**.
`qty_st_vars`	Input	INTEGER	Quantity of state variables bound to subrecord definition.
`st_var_names`	Input	CHARACTER*n array ($n \le$ **`m_max_name_len`**)	Array of state variable short names. Variables must have been previously defined for the family. Size: (`qty_st_vars`).
`class`	Input	CHARACTER*n ($n \le$ **`m_max_name_len`**)	Element class short name.
`format`	Input	INTEGER	Format of mesh object identifiers passed in array `object_ids`. Acceptable values are **`m_list_obj_fmt`** or **`m_block_obj_fmt`**.
`qty`	Input	INTEGER	Number of explicitly listed mesh objects or number of first/last object blocks in array `object_ids` (for `format` equal to **`m_list_obj_fmt`** or **`m_block_obj_fmt`**, respectively).
`object_ids`	Input	INTEGER array	If `format` is **`m_list_obj_fmt`** then `object_ids` is a 1D array (size: [`qty`]) listing each mesh object being bound to the subrecord. If `format` is **`m_block_obj_fmt`** then `object_ids` is a 2D array (size: [2,`qty`]) giving first and last

			identifiers of continuously numbered ranges of mesh objects being bound to the subrecord.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

A state record format is defined through one or more calls to `mf_def_subrec()`. Each call to `mf_def_subrec()` declares the set of state variables, which will be written for a particular set of nodes or elements (or the mesh as a single entity in the case of global state variables) belonging to a single subclass. The array `object_ids` identifies the mesh objects for which state data will be output in the subrecord and also specifies the order of their data.

The order in which calls are made to `mf_def_subrec()` for a particular state record defines the overall order for state data within a state record. For example, if one call to `mf_def_subrec()` defines state variables for all nodes in a mesh and a second call defines state variables for all beams in a mesh, then the format of actual written state data should be nodal data followed immediately by beam data.

In a similar vein, the order of state variable names in array `st_var_names` defines the overall order for data within a given subrecord. If the subrecord is organized as **`m_object_ordered`** then state variables for each mesh object are to be written contiguously and these object "vectors" are ordered according to the overall sequence specified in array `object_ids`. If the subrecord is organized as **`m_result_ordered`** then the data for each state variable is to be written contiguously across all objects, resulting in one array for each state variable. The order of these result arrays is dictated by the order of the state variable names. The order of instances of a given state variable within an individual result array is dictated by the object sequence specified in array `object_ids`.

Array `object_ids` can be formatted in either of two ways to allow flexibility for applications. An example of the **`m_list_obj_fmt`** would be a one-dimensional array containing ten elements numbered "26, 27, 28, 1, 2, 3, 4, 5, 34, 19". A **`m_block_obj_fmt`** array specifying an identical collection and order of elements would be an array with two rows and four columns containing the pairs "(26,28), (1,5), (34,34), (19,19)".

Note: If a subrecord is organized as `m_object_ordered`, the data types of all state variables bound to the subrecord definition must be the same. The type of an array variable is simply the numeric type of each element, so this restriction does not preclude interleaving arrays with scalars in a subrecord.

When defining a subrecord for a class derived from the **`m_mesh`** superclass (i.e., global mesh state data), the quantity of object blocks should be specified as zero. The contents of `format` and `object_ids` will be ignored, but may be specified as **`m_dont_care`**

in the actual call for clarity. The organization of the subrecord (**m_object_ordered** or **m_result_ordered**) is immaterial to the arrangement of the data as it is written for a global data subrecord. That is, result arrays will have only one element so the contiguous collection of result arrays will be identical to a single object vector.

It is important to conceptually distinguish the subrecord name from the subclass name. The subrecord name identifies an output order and a binding between a collection of state variables and a collection of elements. The subclass name identifies the group of elements to which the bound elements belong. In cases where a given mesh data class (**m_quad**, **m_hex**, etc) has only one subclass and one subrecord, it may be useful and less confusing to use the same name for both the subrecord and the subclass.

Example:

```
      integer fam_id, srec_id, status

      character*(m_max_name_len) global_name, nodal_name
      data global_name, nodal_name / 'Global', 'Nodal' /

      character*(m_max_name_len) global_short_names(4)
      data global_short_names / 'ke', 'pe', 'te', 'div' /

      character*(m_max_name_len) nodal_short_names(6)
      data nodal_short_names / 'posx', 'posy', 'posz', 'velx'
     +    'vely', 'velz' /

      integer node_blocks(2,4)
      data node_blocks / 200,275,  1,125,  500,545,  700,1000 /

      ! Open Mili family, initialize mesh, make state record...

      mf_def_subrec( fam_id, srec_id, global_name,
     +               m_object_ordered, 4, global_short_names
     +               global_name, m_dont_care, 0, m_dont_care,
     +               status )
      mf_print_err( 'mf_def_subrec, Global', status )

      mf_def_subrec( fam_id, srec_id, 'Node subset 1',
     +               m_object_ordered, 6, nodal_short_names,
     +               nodal_name, m_block_elem_fmt, 4,
     +               node_blocks, status )
      mf_print_err( 'mf_def_subrec, Nodal set 1', status )
```

See also:

C API: `mc_def_subrec()`
file mili_fparam.h

mf_def_svars()

Define one or more scalar state variables.

Synopsis:

```
mf_def_svars( fam_id, qty_vars, types, names, titles, status
)
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`qty_vars`	Input	INTEGER	Quantity of state variables being defined.
`types`	Input	INTEGER array	Mili data type for each scalar being defined. Possible values are: **`m_real`**, **`m_int`**.
			Size: (`qty_vars`)
`names`	Input	CHARACTER*n array	Short names for each of the scalar variables.
		($n \le$ **`m_max_name_len`**)	Size: (`qty_vars`)
`titles`	Input	CHARACTER*n array	Long names for each of the scalar variables.
		($n \le$ **`m_max_name_len`**)	Size: (`qty_vars`)
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Short names (array `names`) should be brief, mnemonic handles for the variables. Long names (array `titles`) should be handles suitable for labeling a plot or menu.

All state variable short names must be unique. An attempt to re-enter an extant name will cause a warning to be issued, but the call to `mf_def_svars()` will not fail. The danger, if such a warning is not heeded, is that all the associated information for the second entry attempt (such as the title and data type) will be ignored since no new entry is made, and all references to the variable will utilize only the original entry.

Although each state variable is unique, each one can be referenced by any number of subrecord definitions (see `mf_def_subrec()`).

Example:

```
integer fam_id, status

integer types(6)
data types / 6 * m_real /

character*(m_max_nam_len) names, titles
data names / 'sxx', 'syy', 'szz', 'sxy', 'syz', 'szx' /
```

```
      data titles / 'Sigma-xx', 'Sigma-yy', 'Sigma-zz', 'Sigma-
      xy',
     +              'Sigma-yz', 'Sigma-zx' /


      ! Open Mili family...

      mf_def_svars( fam_id, 6, types, names, titles,
     +              status )
      mf_print_err( 'mf_def_svars in myprog', status )
```

See also:

mf_def_subrec()
C API: mc_def_svars()
File mili_fparam.h

mf_def_vec_arr_svar()

Define a vector array state variable.

Synopsis:

```
mf_def_vec_arr_svar( fam_id, type, order, dimensions,
                     field_qty, name, title, field_names,
                     field_titles, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`type`	Input	INTEGER	Mili data type. Possible values: **`m_real`**, **`m_int`**
`order`	Input	INTEGER	Number of array dimensions (must be less than or equal to **`m_max_array_dims`**).
`dimensions`	Input	INTEGER array	Array of dimension sizes for the state variable array being defined. Size: (`order`)
`field_qty`	Input	INTEGER	Number of fields in vector.
`name`	Input	CHARACTER*n ($n \le$ **`m_max_name_len`**)	State variable short name.
`title`	Input	CHARACTER*n ($n \le$ **`m_max_name_len`**)	State variable long name.
`field_names`	Input	CHARACTER*n array ($n \le$ **`m_max_name_len`**)	Short names for each of the fields. Size: (`field_qty`)
`field_titles`	Input	CHARACTER*n array ($n \le$ **`m_max_name_len`**)	Long names for each of fields. Size: (`field_qty`)
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Vector arrays are provided to format state data, which are managed like arrays of one-dimensional vectors but for which it is desired to reference the vector fields explicitly by name rather than by array index (this could impact, for example, the information that is available in a post-processor menu of available results). In memory, the vector array state variable is identical to an array state variable of order `(order + 1)` in which the size of the first dimension is `field_qty`.

`name` and `field_names` should be brief, mnemonic handles.

`title` and `field_titles` should be handles suitable for labeling a plot or menu.

The field names are entered into the same name space as all state variable short names and so should be unique. If a field name already exists as a state variable, definition of the list array is allowed to continue as long as the extant state variable is a scalar of the same numeric type as the list field.

Although each vector array state variable is unique, each one can be referenced by any number of subrecord definitions (see `mf_def_subrec()`). Also, since the fields themselves are registered as independent state variables, they may be referenced as individual scalars when formatting state records.

Example:

```
      integer fam_id, status
      integer dimensions(1)
      data dimensions / 2000 /
      character*(m_max_nam_len) field_names(3), field_titles(3)
      data field_names / 'velx', 'vely', 'velz' /
      data field_titles / 'X Velocity', 'Y Velocity', 'Z
      Velocity'/

      ! Open family...

      mf_def_vec_arr_svar( fam_id, m_real, 1, dimensions, 3,
      'vel',
     +                     'Velocity Vector', field_names,
     +                     field_titles, status )
      mf_print_err( 'mf_def_vec_arr_svar in myprog', status )
```

See also:

C API: `mc_def_vec_arr_svar()`
File mili_fparam.h

mf_def_vec_svar()

Define a vector state variable.

Synopsis:

```
mf_def_vec_svar( fam_id, type, field_qty, name, title,
                 field_names, field_titles, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`type`	Input	INTEGER	Mili data type. Possible values: **`m_real`**, **`m_int`**
`field_qty`	Input	INTEGER	Number of fields in vector.
`name`	Input	CHARACTER*n ($n \le$ **`m_max_name_len`**)	State variable short name.
`title`	Input	CHARACTER*n ($n \le$ **`m_max_name_len`**)	State variable long name.
`field_names`	Input	CHARACTER array	Short names for each of the fields. Size: (`field_qty`)
`field_titles`	Input	CHARACTER array	Long names for each of fields. Size: (`field_qty`)
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Vectors are provided to format state data, which are managed like one-dimensional arrays but for which it is desired to reference the components explicitly by name rather than by array index (this could impact, for example, the information that is available in a post-processor menu of available results). In memory, the vector state variable is identical to a one-dimensional array state variable of size `field_qty`.

`name` and `field_names` should be brief, mnemonic handles.
`title` and `field_titles` should be handles suitable for labeling a plot or menu.

The field names are entered into the same name space as all state variable short names and so should be unique. If a field name already exists as a state variable, definition of the list array is allowed to continue as long as the extant state variable is a scalar of the same numeric type as the list field.

Although each vector state variable is unique, each one can be referenced by any number of subrecord definitions (see `mf_def_subrec()`). Also, since the fields themselves

are registered as independent state variables, they may be referenced as individual scalars when formatting state records.

Example:

```
integer fam_id, status
character*10 field_names(3), field_titles(3)
data field_names / 'velx', 'vely', 'velz' /
data field_titles / 'X Velocity', 'Y Velocity', 'Z
Velocity'/

! Open family...

mf_def_vec_svar( fam_id, m_real, 3, 'vel', 'Velocity
Vector',
+                field_names, field_titles, status )
mf_print_err( 'mf_def_vec_svar in myprog', status )
```

See also:

C API: `mc_def_vec_svar()`
File mili_fparam.h

mf_delete_family()

Delete a Mili file family.

Synopsis:

```
mf_delete_family( root_name, path, status )
```

Arguments:

`root_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Mili family root name.
`path`	Input	CHARACTER*n ($n \leq$ **`m_max_path_len`**)	Path to directory where family is to be located. May be absolute or relative to the current working directory.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

The Mili family specified for deletion need not be opened by the calling process, but the family will be closed first if it is. Mili will not detect if another process has the family open in read-only mode, so caution must be exercised in the use of this subroutine. If a different process concurrently has write access on the family, `mf_delete_family()` will fail with a non-zero return status.

If Mili cannot find any files belonging to the specified family (i.e., have names consisting of the given root followed by a purely numeric suffix or followed by a purely upper-case alphabetic suffix), it will return a status of zero even though it didn't delete any files. Thus, Mili will not give an indication of an incorrect path or root name specification by the calling application.

See also:

C API: `mc_delete_family()`

mf_filelocking_disable()

Disables the legacy Mili file locking mechanism that generated a lock file to ensure there was only one writer to a file family. File locking is removed from Mili completely in future releases since Mili does not support multiple-writers to the same file family.

Synopsis:

```
mf_( void )
```

Arguments: None.

Notes:

See also:

`mf_filelocking_enable()`

C API: `mc_filelocking_disable()`

mf_filelocking_enable()

Enables the legacy Mili file locking mechanism that generated a lock file to ensure there was only one writer to a file family. The default is for file locking to be enabled and it can only be disabled by calling *mf_filelocking_disable()*.
File locking with is removed from Mili completely in future releases since Mili does not support multiple-writers to the same file family.

Synopsis:

```
mf_filelocking_enable( void )
```

Arguments: None.

Notes:

See also:

`mf_filelocking_disable()`
C API: `mc_filelocking_enable()`

mf_flush()

Write any cached information and flush buffers for state data files or non-state data files.

Synopsis:

```
mf_flush( fam_id, file_type, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`file_type`	Input	INTEGER	Type of data file to flush to disk. Possible values are **`m_state_data`** or **`m_non_state_data`**
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Depending on the value of parameter `file_type`, `mf_flush()` flushes the I/O buffers for any currently opened state-data file or non-state data file. Any data that is normally buffered in memory, such as state record format descriptions, is written out prior to flushing the buffers (state data is never cached by Mili, but the C standard I/O library upon which Mili is layered is buffered).

Because of the requirements for managing non-state information, the current non-state data file is closed after a flush operation. Any subsequent output of non-state information (such as a named parameter, a new state variable definition, or a new state record format descriptor), will be written to a new non-state data file

The flush operation is performed automatically when `mf_close()` is called, but applications may wish to flush at other times to guard against file family corruption in the event of a system crash. For example, it might be prudent to flush non-state data prior to writing state data so that state record formats will be available even if the analysis run does not terminate normally.

See also:

C API: `mc_flush()`

mf_get_class_info()

Return the class name and size of a mesh object class distinguished within its superclass by a unique index.

Synopsis:

```
mf_get_class_info( fam_id, mesh_id, superclass,
                   class_index,
                   short_name, long_name, object_qty,
                   status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh numeric identifier.
`superclass`	Input	INTEGER	The Mili superclass identifier.
`class_index`	Input	INTEGER	An integer on [0, *qty* - 1], where *qty* is the quantity of classes in the specified superclass for the specified mesh as returned from a call to `mf_query_family()`
`short_name`	Output	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Object class short name.
`long_name`	Output	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Object class long name.
`object_qty`	Output	INTEGER	Quantity of objects in class.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Utility function.

Example:

See also:

C API: `mc_get_class_info()`

mf_get_metadata()

Returns metadata fields for a file family.

Synopsis:

```
mf_get_metadata( fam_id, mili_version, host, arch,
                 timestamp, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mili_version`	Output	CHARACTER*_n_	Mili version string.
`host`	Output	CHARACTER*_n_	Host name where file was created.
`arch`	Output	CHARACTER*_n_	Architecture name where file was created.
`timestamp`	Output	CHARACTER*_n_	Date and time of file creation.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Utility function.

Example:

See also:

C API: `mc_get_mesh_id()`

mf_get_mesh_id()

Return the numeric identifier for the named mesh.

Synopsis:

```
mf_get_mesh_id( fam_id, name, mesh_id, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n	Mesh name.
		($n \leq$ **`m_max_name_len`**)	
`mesh_id`	Output	INTEGER	Mesh numeric identifier.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Utility function.

Example:

```
integer fam_id, mesh_id, status

! Open Mili family...

mf_get_mesh_id( fam_id, 'My mesh', mesh_id, status )
mf_print_err( 'mf_get_mesh_id in myprog', status )
```

See also:

C API: `mc_get_mesh_id()`

mf_get_simple_class_info()

Return the class name and start and stop identifier numbers for a simple mesh object class distinguished within its superclass by a unique index.

Synopsis:

```
mf_get_simple_class_info( fam_id, mesh_id, superclass,
                          class_index, short_name,
                         long_name,
                          start_ident, stop_ident, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh numeric identifier.
`superclass`	Input	INTEGER	The Mili superclass identifier.
`class_index`	Input	INTEGER	An integer on [0, *qty* - 1], where *qty* is the quantity of classes in the specified superclass for the specified mesh as returned from a call to `mf_query_family()`
`short_name`	Output	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Object class short name.
`long_name`	Output	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Object class long name.
`start_ident`	Output	INTEGER	First object identifier in class.
`stop_ident`	Output	INTEGER	Last object identifier in class.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Utility function.

Example:

See also:

C API: `mc_get_simple_class_info()`

mf_limit_filesize()

Set the maximum number of bytes allowable in any state-data file.

Synopsis:

```
mf_limit_filesize( fam_id, filesize, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`filesize`	Input	INTEGER	Maximum number of bytes per file rounded to the nearest state record size.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

The number of bytes allowed per file is rounded to the nearest state record size. Setting the filesize limit will override the state number limit (set with mf_limit_states) if both are set.

See also:

C API: `mc_limit_states()`

mf_limit_states()

Set the maximum number of states allowable in any state-data file.

Synopsis:

```
mf_limit_states( fam_id, limit, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`limit`	Input	INTEGER	Maximum state quantity.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

By default, the per-file state limit is set to one hundred. Applications that change this should call `mf_limit_states()` *prior to writing any state data and should not change it thereafter*. If the current partition scheme is not **`m_state_limit`** (currently the only scheme available), this call has no effect.

See also:

C API: `mc_limit_states()`

mf_load_conns()

Read element connectivities, materials, and part numbers for an element mesh object class from a file family into memory.

Synopsis:

```
mf_load_conns( fam_id, mesh_id, class_name, connects,
               materials, parts, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh id number.
`class_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Element object class.
`connects`	Output	INTEGER array	Array of element connectivities. Size: (n,qty) where n is the number of nodal connectivities required for the element object superclass which the specified element class is derived from (see Table 1) and qty is the number of elements in the class as returned from a call to `mf_get_class_info()`.
`materials`	Output	INTEGER array	Array of material numbers. Size: (qty) where qty is the number of elements in the class as returned from a call to `mf_get_class_info()`.
`parts`	Output	INTEGER array	Array of part numbers. Size: (qty) where qty is the number of elements in the class as returned from a call to `mf_get_class_info()`.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

See also:

`mf_get_class_info()`
C API: `mc_load_conns()`

mf_load_conn_labels()

Read in element labels for a specified class.

Synopsis:

```
mf_load_conn_labels( fam_id, mesh_id, class_name, labels,
                     status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Unstructured mesh identifier
`class_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of element class. Must have been previously defined with a call to `mf_def_class()`.
`labels`	Output	INTEGER array	Array of element labels. Size: Number of elements in the class.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Element labels are optional. The caller must allocate space for the labels.

Example:

```
integer fam_id, mesh_id, status
integer labels(4)

character*(m_max_name_len) tet_class
data tet_class / 'Tet' /

! Open Mili family...

mf_make_umesh( fam_id, 'Test mesh', 3, mesh_id, status )
mf_def_class( fam_id, mesh_id, m_tet, tet_class )
mf_load_conn_labels( fam_id, mesh_id, tet_class, labels,
+                    status )
mf_print_err( 'mf_def_conn_labels in myprog', status )
```

See also:

`mf_def_conn_labels()`
`mf_def_node_labels()`
`mf_load_node_labels()`
C API: `mc_load_conn_labels()`

mf_load_nodes()

Read nodal coordinates for a node object class from a file family into memory.

Synopsis:

```
mf_load_nodes( fam_id, mesh_id, class_name, coords, status)
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh id number.
`class_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Nodal object class.
`coords`	Output	REAL array	Array of nodal coordinates. Size: (*dims*,*qty*) where *dims* are the dimensionality of meshes in the family and *qty* is the number of nodes in the class as returned from a call to `mf_get_class_info()`.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

See also:

`mf_get_class_info()`
C API: `mc_load_nodes()`

mf_load_node_labels()

Read in nodal labels.

Synopsis:

```
mf_load_node_labels( fam_id, mesh_id, class_name, labels,
                     status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Unstructured mesh identifier
`class_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of node class. Must have been previously defined with a call to `mf_def_class()`. The superclass associated with `class` must be `m_node`.
`labels`	Output	INTEGER array	Array of nodal labels. *Size:* Number of nodes in the class.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Nodal labels are optional. The caller must allocate space for the labels.

Example:

```
integer fam_id, mesh_id, status
integer labels(4)

character*(m_max_name_len) node_class
data node_class / 'Nodal' /

! Open Mili family...

mf_make_umesh( fam_id, 'Test mesh', 3, mesh_id, status )
mf_def_class( fam_id, mesh_id, m_node, node_class )
mf_load_node_labels( fam_id, mesh_id, node_class, labels,
+                    status )
mf_print_err( 'mf_def_node_labels in myprog', status )
```

See also:

`mf_load_conn_labels()`

C API: `mc_load_node_labels()`

mf_load_surface()

Reads in the connectivity definition for the specified surface number.

Synopsis:

```
mf_load_surface( fam_id, mesh_id, surf_id, short_name,
                 conns, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh id number.
`Surf_id`	Input	INTEGER	Surface id number.
`short_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Surface object class.
`conns`	Output	Integer array	Array of nodal connection numbers.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

See also:

`mf_def_conn_surface()`

C API: `mc_load_surface()`

mf_make_umesh()

Create an empty unstructured mesh descriptor.

Synopsis:

```
mf_make_umesh( fam_id, name, dims, mesh_id, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n	Mesh name.
		($n \leq$ **`m_max_name_len`**)	
`dims`	Input	INTEGER	Quantity of spatial dimensions in the mesh (two or three).
`mesh_id`	Output	INTEGER	Mesh numeric identifier.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

A mesh must be created and defined prior to defining state record formats for it. This call creates an empty mesh descriptor, which is then filled by defining nodal coordinates and element connectivities for the mesh.

See also:

`mf_def_nodes()`
`mf_def_conn_arb()`
`mf_def_conn_seq()`
C API: `mc_make_umesh()`

mf_new_state()

Prepare a family to receive data for a new state.

Synopsis:

```
mf_new_state( fam_id, srec_id, time, file_sequence,
              file_state_index, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`srec_id`	Input	INTEGER	State record format identifier.
`time`	Input	REAL	Time value associated with new state record.
`file_sequence`	Output	INTEGER	The filename sequence number of the state data file into which the new state will be written.
`file_state_index`	Output	INTEGER	The zero-based index of the new state within the state data file where it will be written.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

`mf_new_state()` must be called once prior to writing state data for each new state.

See also:

C API: `mc_new_state()`

mf_open()

Open a Mili file family.

Synopsis:

```
mf_open( root_name, path, control, fam_id, status )
```

Arguments:

`root_name`	Input	CHARACTER*n ($n \le$ **`m_max_name_len`**)	Mili family root name.
`path`	Input	CHARACTER*n ($n \le$ **`m_max_path_len`**)	Path to directory where family is to be located. May be absolute or relative to the current working directory.
`control`	Input	CHARACTER*n ($n \le$ **6**)	Access control string (see *Notes* below).
`fam_id`	Output	INTEGER	Mili family identifier.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

`control` is a character string composed of zero or more pairs of characters. The first character of each pair specifies a control function; the second character of the pair supplies the value for the control function. By convention, the control function specifiers are upper case letters and the possible values are lower case letters.

Each control function has a default value depending on whether or not the Mili family identified by `root_name` and `path` already exists. Thus, it is possible to leave the control string empty and let Mili supply all the control function values. In general, Mili's defaults are designed to permit the least amount of modification possible on incoming data or to an existing family. The table below describes the control functions, their possible values, and their defaults.

Control Function	Description	Values		When Default?
`A`	Mili family access mode	`w`	Write access; delete family if extant	When specified family does not exist
		`r`	Read access	When specified family does exist
		`a`	Append access	Never
`P`	Precision limit – sets maximum precision of state	`d`	Double precision (64 bits)	Always

	data stored in family (higher precision state variables are automatically converted); *applicable only when a family is created and fixed thereafter*	`s`	Single precision (32 bits)	Never
`E`	Endianness of data written to family (translation applied automatically when necessary); *applicable only when a family is created and fixed thereafter*	`n`	Native endianness of host computer	Always
		`b`	Big endian	Never
		`l`	Little endian	Never

Notes

- Defaults are considered only when the control function is not specified by the `control` parameter.
- Functions `P` and `E` are applied and fixed for a family when it is first created. Specifications of either function are ignored on subsequent opens of the family (unless opened for write access, which causes the family to be created anew).

File deletions occur on write-access opening if there are files present in the target directory that fall into either of the following two categories: (1) they have file names consisting of the root name followed by a purely numeric suffix (i.e., Mili state-data files), in any sequence; (2) they have file names consisting of the root name followed by a purely upper-case alphabetic suffix (i.e., Mili non-state-data files). The intent of this logic is to remove any existing files in the specified Mili family while permitting the existence of other files that have names starting with the same root. Any file with the same root name followed by characters which are non-numeric and non-alphabetic or mixed alphabetic and numeric character strings or lower-case will not be deleted, as these are not names which will be produced by Mili.

Mili supports a limited form of concurrent access in that multiple processes can have one family open simultaneously. However, only one process can have write access, so attempts to open an existing family with "write" or "append" access will fail when another process has already established either of those access types.

Previous versions of Mili supported a "`mode`" string argument, which was formatted differently than the current `control` argument and offered a subset of the current control functions (although some of the functionality documented for the `mode` string was in fact never implemented). For backward compatibility with applications that implement the previous `mode` string format, Mili detects occurrences of the previous format and translates them into the new format as shown in the table below.

`mode` **format**	**Translated** `control` **format**
r	Ar
rs	Ar
rd	Ar
w	AwPd
ws	AwPsEb
wd	AwPdEb
a	Aa
as	AaPsEb
ad	AaPdEb

The constraints for the P and E control functions still apply, i.e., they are utilized only when a family is first created and are fixed thereafter regardless of the `control` specification on subsequent opens.

Mili also provides backward compatibility between the current version of the library and existing Mili databases that were created with recent versions of the library, which supported the "`mode`" string argument. Although there are binary differences between the current and older database formats, they are few and well defined and it is possible for Mili to unambiguously translate from the old to the new. Users of the Mili dump utility "md" should be aware that dumps of old-format databases with a current copy of "md" will be indistinguishable from dumps of new-format databases.

Example:

```
integer fam_id, status

mf_open( 'test', '.', 'AwPs', fam_id, status )
mf_print_err( 'mf_open in myprog', status )
```

See also:

C API: `mc_open()`

mf_open_srec()

Create an empty state record format descriptor.

Synopsis:

```
mf_open_srec( fam_id, mesh_id, srec_id, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh identifier (from `mf_make_umesh()`).
`srec_id`	Output	INTEGER	State record format descriptor identifier.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Each mesh may have multiple state record formats defined. The state record format is actually described through one or more calls to `mf_def_subrec()`. Contents of a state record format, created by calls to `mf_def_subrec()`, will not be available to be saved to disk until the state record format is closed with a call to `mf_close_srec()`.

See also:

`mf_def_subrec()`, `mf_close_srec()`
C API: `mc_open_srec()`

mf_partition_state_data()

Force closure of the current state data file to create a partition in the state data file sequence.

Synopsis:

```
mf_partition_state_data( fam_id, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Subroutine `mf_partition_state_data()` can be used to override the current family partitioning scheme and create a file sequence partition at an arbitrary point in the state record stream. This is intended to support analysis restarts, but applications can use it at any time to override the automatic partitioning provided by Mili. The application bears responsibility for ensuring that the current state is written completely to disk prior to calling `mf_partition_state_data()`. Incomplete states will not be accessible by applications attempting to read the database.

See also:

`mf_restart_file()`
C API: `mc_partition_state_data()`

mf_print_error()

Print a diagnostic associated with a Mili subroutine return status.

Synopsis:

```
mf_print_error( preamble, rval )
```

Arguments:

`preamble`	Input	CHARACTER*n ($n \leq$ **`m_max_preamble_len`**)	Application text to be prepended (with a colon) to the diagnostic message.
`rval`	Input	INTEGER	Status return value from a previous call to a Mili Fortran subroutine.

Notes:

If (and only if) `rval` is non-zero, indicating an error condition in the Mili subroutine call that returned it, `mf_print_error()` prints to standard error the text in `preamble` followed by a colon and an explanation of the error condition detected.

There is no return status from `mf_print_error()`.

See also:

C API: `mc_print_error()`

mf_query_family()

Request information about a Mili database family.

Synopsis:

```
mf_query_family( fam_id, req_type, num_args, char_arg,
                 data, rval )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`req_type`	Input	INTEGER	FORTRAN parameter identifying the information being requested (see *Notes*).
`num_args`	Input	INTEGER or REAL	Array of numbers (dependent on `req_type`, above) providing additional information, which may be needed to resolve request.
`char_arg`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Character string providing additional information, which may be needed to resolve request.
`data`	Output	Request dependent (see *Notes*, below)	Output from request.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

The possible values of the `req_type` argument, their meanings and type of data returned, and the associated definitions for `num_args` and `char_arg` are as follows:

req_type	**Output/Inputs**
`m_class_superclass`	Superclass for a given class (INTEGER)
`num_args`	`num_args(1)` = id of mesh (INTEGER)
`char_arg`	Mesh object class name
`m_class_exists`	Non-zero if named class exists, else zero (INTEGER)
`num_args`	`num_args(1)` = id of mesh (INTEGER)
`char_arg`	short name of candidate class
`m_lib_version`	Version number of library as a string (CHARACTER*(**`m_max_name_len`**))[1]
`num_args`	**`m_dont_care`**
`char_arg`	**`m_blank`**
`m_multiple_times`	Simulation time for an arbitrary sequence of state records

(REAL)

num_args	num_args(1) = quantity of desired state times (INTEGER) num_args(2) = num_args(qty + 1) = indices of states for which times are desired; where "qty" is the value placed in num_args(1); each index must be a number on [1, qty_states] (INTEGER)
char_arg	**m_blank**

m_next_state_loc Array (length 2) containing, in order, a filename sequence number and file state index for the state following that identified by an input filename sequence number and file state index (INTEGER, INTEGER). **Note** – if the input state indices identify the last state currently in the family, the output indices will correctly identify the file and state for the next state initiated by a subsequent call to mf_new_state(). However, an intervening call to mf_partition_state_data(), mf_restart_at_state(), or mf_restart_at_file() before the next mf_new_state() call may invalidate these next state indices. In such a case, another **m_next_state_loc** query (after the intervening partition or restart call) with the same input will return outputs that are again correct.

num_args	num_args(1) = input filename sequence number; must identify an extant state data file in the family (INTEGER) num_args(2) = input file state index; must identify an extant state (zero-based) in the file identified by num_args(1) (INTEGER)
char_arg	**m_blank**

m_qty_class_in_sclass Quantity of mesh object classes derived from a particular superclass (INTEGER)

num_args	num_args(1) = id of mesh (INTEGER) num_args(2) = superclass (INTEGER)
char_arg	**m_blank**

m_qty_dimensions Dimensionality of mesh(es) in the family (INTEGER)

num_args	**m_dont_care**
char_arg	**m_blank**

m_qty_elem_conn_defs Quantity of element connectivity definition entries in a particular class (INTEGER)

num_args	num_args(1) = id of mesh (INTEGE)
char_arg	Element class name

m_qty_elems_in_def — Quantity of elements defined in a particular connectivity definition entry for a particular element class (INTEGER)

- num_args
  - `num_args(1)` = id of mesh (INTEGER)
  - `num_args(2)` = index of desired connectivity definition; must be a number on [0, qty_conn_defs - 1], where "qty_conn_defs" is the data returned by the `mf_query_family()` call with `req_type` = **m_qty_elem_conn_defs** (INTEGER)
- char_arg
  - Element class name

m_qty_meshes — Quantity of meshes defined in a family (INTEGER)

- num_args
  - **m_dont_care**
- char_arg
  - **m_blank**

m_qty_node_blks — Quantity of nodal coordinate blocks defined for a particular mesh in a family (INTEGER)

- num_args
  - `num_args(1)` = id of mesh (INTEGER)
- char_arg
  - Node class name

m_qty_nodes_in_blk — Quantity of nodes defined in a particular coordinates block (INTEGER)

- num_args
  - `num_args(1)` = id of mesh (INTEGER)
  - `num_args(2)` = index of desired coordinate block; must be a number on [0, qty_node_blks - 1], where "qty_node_blks" is the data returned by the `mf_query_family()` call with `req_type` = **m_qty_node_blks** (INTEGER)
- char_arg
  - Node class name

m_qty_srec_fmts — Quantity of state record formats defined in a family (INTEGER)

- num_args
  - **m_dont_care**
- char_arg
  - **m_blank**

m_qty_states — Quantity of complete states currently written to a family (INTEGER)

- num_args
  - **m_dont_care**
- char_arg
  - **m_blank**

m_qty_subrecs — Quantity of subrecord bindings in a particular state record format (INTEGER)

- num_args
  - `num_args(1)` = id of desired state record format (INTEGER)
- char_arg
  - **m_blank**

m_qty_subrec_svars — Quantity of state variables bound to a particular subrecord definition (INTEGER)

num_args	num_args(1) = id of desired state record format (INTEGER) num_args(2) = index of desired subrecord; must be a number on [0, qty_subrecs - 1], where "qty_subrecs" is the data returned by the mf_query_family() call with req_type = **m_qty_subrecs** (INTEGER)
char_arg	**m_blank**
m_qty_svars	Quantity of state variable definitions in a family (INTEGER)
num_args	num_args(1) = id of mesh (INTEGER)
char_arg	**m_blank**
m_series_srec_fmts	State record format identifier for a continuous series of state data records identified by first and last indices (INTEGER)
num_args	num_args(1) = index of first desired state; must be a number on [1, qty_states], where "qty_states" is the data returned by the mf_query_family() call with req_type = **m_qty_states** (INTEGER) **num_args(2)** = index of last desired state; must be a number on [1, qty_states] and must be greater than num_args(1) (INTEGER)
char_arg	**m_blank**
m_series_times	Simulation times for a continuous series of state records (REAL)
num_args	num_args(1) = index of first desired state; must be a number on [1, qty_states], where "qty_states" is the data returned by the mf_query_family() call with req_type = **m_qty_states** (INTEGER) num_args(2) = index of last desired state; must be a number on [1, qty_states] (INTEGER) The two indices can specify an ascending or descending series order.
char_arg	**m_blank**
m_srec_fmt_id	State record format identifier for a particular state data record (INTEGER)
num_args	num_args(1) = index of desired state; must be a number on [1, qty_states], where "qty_states" is the data returned by the mf_query_family() call with req_type = **m_qty_states** (INTEGER)
char_arg	**m_blank**
m_srec_mesh	Mesh identifier associated with a state record format (INTEGER)
num_args	num_args(1) = id of state record format (INTEGER)
char_arg	**m_blank**
m_state_of_time	State indices, each on [1, qty_states], of the two states bounding the input simulation time value; if input time matches a state time

	exactly, that state's index will be the first of the two (unless input time matches the last state time) (INTEGER)
num_args	num_args(1) = specified simulation time (REAL)
char_arg	**m_don't_care**
m_state_time	Simulation time for the specified state record (REAL)
num_args	num_args(1) = index of desired state; must be a number on [1, qty_states], where "qty_states" is the data returned by the mf_query_family() call with req_type = **m_qty_states** (INTEGER)
char_arg	**m_blank**
m_subrec_class	Class name for objects in a subrecord (CHARACTER)
num_args	num_args(1) = id of desired state record format num_args(2) = index of desired subrecord; must be a number on [0, qty_subrecs - 1], where "qty_subrecs" is the data returned by the mf_query_family() call with req_type = **m_qty_subrecs** (INTEGER)
char_arg	**m_blank**

[1]For the **m_lib_version** query, Mili only writes the characters of its internally stored version string to the application's CHARACTER variable; the application bears responsibility for initializing (blank-filling) the variable prior to making the query.

See also:

C API: mc_query_family()

mf_read_results()

Read state variables from a subrecord into result-ordered arrays.

Synopsis:

```
mf_read_results( fam_id, state, subrec_id, qty, results,
                 data, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`state`	Input	INTEGER	State number (one-based) from which to read data.
`subrec_id`	Input	INTEGER	Zero-based index of subrecord containing requested state variables.
`qty`	Input	INTEGER	Quantity of requested state variables.
`results`	Input	CHARACTER array	State variable short names identifying requested results. Names of vector or array variables can optionally include additional text, which specifies a subset of the variable (see *Notes* below).
`data`	Output	Size and numeric type as required by requested results.	Buffer to receive results. Individual state variable arrays are appended sequentially into the buffer. See *Notes* below regarding alignment constraints.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Array `results` contain character strings, which identify the results being requested. All results must come from the same subrecord. Each result character string consists of a state variable short name. For state variables having aggregate types of vector, array, or vector array, there may be additional text appended to the short name, which specifies a subset of the variable for collection. For example, if a vector "vel" is defined which has components "x", "y", and "z", it is possible to retrieve any one of the components individually with the appropriate specification. The formats for the subset specification are as follows:

Vector	*short_name*[*component_name*]
Array	*short_name*[$index_1$, $index_2$,...]
Vector array	*short_name*[*component_name*, $index_1$, $index_2$,...]

Array indices are positive integers starting with one and are listed in column-major order, i.e., $index_1$ specifies the most minor array dimension. It is not necessary to specify an

index for each dimension of arrays and vector arrays, allowing whole dimensions to be collected if desired. However, any dimension indices left out (including the *component_name* for vector arrays) must be contiguous and must start with the most minor dimension of the array. Conversely, to specify a value for the most minor dimension of an array variable, all other dimension indices must be specified. Thus it is **not** possible to specify, for example, a *component_name* alone for a vector array and collect all instances of that component from each instance of the variable in a subrecord.

The memory referenced by the `data` parameter is treated as a byte-array into which the state variable result arrays are sequentially written. Thus, depending on the requested variables, `data` may contain arrays of scalars, arrays of vectors, arrays of arrays, or arrays of vector arrays, each of floating point or integer type and 32-bit or 64-bit size. Each successive variable array is written at the next available address in `data` that is a multiple of the size (four or eight bytes) dictated by the Mili type (`m_real`, `m_int`, etc.) of the state variable, with padding interleaved as necessary between state variable arrays. The logic Mili follows to generate the locations to write each variable array is:

Init current destination address to start of `data`.
Loop over requested state variables:
    Round-up current destination address to next multiple of size of current variable numeric type.
    Save current destination address for current variable array.
    Increment current destination address by the number of bytes required for the current variable array.

In the typical case, where all variables are of the same numeric type and size, `data` can be defined/allocated as that type and all arrays will be adjacent with no padding required. If mixing 32- and 64-bit types there may be padding between arrays depending on the alignment of `data` and the quantity of entries in the variable arrays. Note that padding is utilized based on *evaluation of an absolute address, not the relative offset from the beginning of* `data`. The calling application bears responsibility for ensuring that `data` is large enough to hold all variable arrays and any required padding.

Routine `mf_read_results()` provides the necessary transposition for data in `m_object_ordered` subrecords.

See also:

C API: `mc_read_results()`

mf_restart_at_file()

Prepare a family to receive data for a new state at the beginning of a particular family sequence file.

Synopsis:

```
mf_restart_at_file( fam_id, file_sequence, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`file_sequence`	Input	INTEGER	Filename sequence number (zero-based) specifying the state data file to begin writing with the next new state.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Subroutine `mf_restart_at_file()` prepares the Mili file family to receive data for a new state by overwriting an existing file in the family or appending a new one to the existing sequence of state data files. Data for the next new state written after a call to `mf_restart_at_file()` will be written to a sequence of files starting with the file specified by `file_sequence`, regardless of any extant state data files (although a value for `file_sequence` may not be specified which would create a gap in the name sequence of existing state data files). **Any existing file identified by `file_sequence` and all subsequent continuously numbered state data files in the family will be deleted by a call to `mf_restart_at_file()`.**

Note that the `file_sequence` is not necessarily equivalent to a zero-based count of the state data files present in a family. The Mili library will open an existing Mili family database in which the state data filename sequence does not start with zero. For example, an acceptable Mili family with a root name of "test' could have members "testA", "test08", "test09", "test10", and "test11". With such a database, acceptable values for `file_sequence` in a call to `mf_restart_at_file()` would be integers in the range 8-12.

See also:

C API: `mc_restart_at_file()`

mf_restart_at_state()

Prepare a family to receive new state data by overwriting at an existing state.

Synopsis:

```
mf_restart_at_state( fam_id, file_sequence,
                     file_state_index, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`file_sequence`	Input	INTEGER	Filename sequence number (zero-based) specifying the state data file in which the next new state will be written.
`file_state_index`	Input	INTEGER	Zero-based state index specifying where to begin writing subsequent state data. If `file_sequence` identifies a new file for the family, `file_state_index` must be zero.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Subroutine `mf_restart_at_state()` prepares the Mili file family to receive new data by overwriting an existing state and any subsequent existing states. **The existing state specified by `file_state_index` and all other states and files following it in a continuous sequence will be deleted by a call to `mf_restart_at_state()`.** The state identified by the combination of `file_sequence` and `file_state_index` must be an existing state or represent the next state in the existing sequence of states in the database. If it represents the next state in the existing sequence (i.e., effectively dictating an append instead of an overwrite) the identified state may fall within the last existing file in the family (if that file has not already reached its maximum number of states) or it may be the first state in a new file in the sequence.

See also:

C API: `mc_restart_at_state()`

mf_set_buffer_qty()

Set the number of states' data, which is buffered in memory for a particular mesh object class (or all classes).

Synopsis:

```
mf_set_buffer_qty( fam_id, mesh_id, class_name, buffer_qty,
                   status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mesh_id`	Input	INTEGER	Mesh numeric identifier.
`class_name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of mesh object class for which input buffer quantity is being set. May be empty, in which case all classes are affected.
`buffer_qty`	Input	INTEGER	The quantity of input buffers to use for the specified class(es). May be zero or any positive integer (subject to memory limitations).
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Mili allows multiple states' data to be buffered in memory on a subrecord-by-subrecord basis. The buffers are managed in a circular queue, such that when a requested state is not found in a read-result request, the oldest buffer is overwritten by reading in the new state. Buffering the data allows for significant performance gains under common access scenarios, for example when an application is repeatedly interpolating between two states. By default, two buffers are created when a database is opened. Mili's buffering can be turned off by setting a buffer quantity of zero. On the other hand, if a database is known to be smaller than available memory, the buffer quantity could be set to the quantity of states in the database and after the first access of each state, further "reads" would occur at memory access speeds.

See also:

C API: `mc_set_buffer_qty()`

mf_suffix_width()

Set minimum numeric suffix width for Mili state-data file names.

Synopsis:

```
mf_suffix_width( fam_id, width, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`width`	Input	INTEGER	Minimum suffix width for state-data file names. Must be a positive, non-zero value.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

By default, Mili generates state-data file names by appending a minimum two-character numeric string, starting with “00”, to the family root name. This call allows applications to reset the minimum width of the numeric string.

See also:

C API: `mc_suffix_width()`

mf_wrt_array()

Write a named array to a Mili family.

Synopsis:

```
mf_wrt_array( fam_id, type, name, order, dimensions, data,
              status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`type`	Input	INTEGER	Mili data type of the array entries. Possible values are **`m_real`** and **`m_int`**.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of the array.
`order`	Input	INTEGER	Quantity of dimensions in the array being written.
`dimensions`	Input	INTEGER array	An array containing the size of each dimension in the array being written. Size:(`order`)
`data`	Input	(dependent on `type` parameter above)	Array of values for the named array. Size and mapping must agree with **`order`** and `dimensions`.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

This call writes an array of application data into the file family. It is intended to be used for unique arrays of data. Arrays, which are state-dependent and re-generated with each state, should be declared with `mf_def_arr_svar()` and written as state data.

Example:

See also:

`mf_def_arr_svar()`
C API: `mc_wrt_array()`
File mili_fparam.h

mf_wrt_scalar()

Write a named scalar value to a Mili family.

Synopsis:

```
mf_wrt_scalar( fam_id, type, name, value, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`type`	Input	INTEGER	Mili data type of scalar. Possible values are **`m_real`** and **`m_int`**.
`name`	Input	CHARACTER*n	Name of the scalar.
		($n \leq$ **`m_max_name_len`**)	
`value`	Input	(dependent on `type` above)	Value of the scalar.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Example:

See also:

C API: `mc_wrt_scalar()`
File mili_fparam.h

mf_wrt_stream()

Write state data as a stream of values.

Synopsis:

```
mf_wrt_stream( fam_id, type, quantity, data, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`type`	Input	INTEGER	Mili type of values being written. Possible values are **`m_real`** and **`m_int`**.
`quantity`	Input	INTEGER	Quantity of values being written.
`data`	Input	(dependent on `type` above)	One-dimensional array of values being written.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

This interface provides the lowest overhead within the I/O library for writing state data since it forces the application to bear all responsibility for formatting the data and sequencing the output calls correctly. `mf_wrt_stream()` simply copies the array of data values into the current state record at the current position. If one or more calls to `mf_wrt_stream()` should write more data than the state record format allows for, the excess will be overwritten on the next call to `mf_new_st()` and subsequent state data writes.

All data values must be of the same base type, since any type conversion that is necessary is determined by the single `type` parameter. This allows, for example, **`m_real`** scalars to be interleaved with **`m_real`** arrays in a single output buffer.

Example:

See also:

C API: `mc_wrt_stream()`
File mili_fparam.h

mf_wrt_string()

Write a named string to a Mili family.

Synopsis:

```
mf_wrt_string( fam_id, name, value, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n	Name of the string.
		($n \leq$ **`m_max_name_len`**)	
`value`	Input	CHARACTER*n	The string to be written.
		($n \leq$ **`m_max_string_len`**)	
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

See also:

C API: `mc_wrt_string()`
File mili_fparam.h

mf_wrt_subrec()

Write part of a subrecords state data.

Synopsis:

```
mf_wrt_subrec( fam_id, name, start, stop, data, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Subrecord name used in `mf_def_subrec()`.
`start`	Input	INTEGER	First unit (object vector or result array) of the subrecord being written during this call.
`stop`	Input	INTEGER	Last unit (object vector or result array) of the subrecord being written during this call.
`data`	Input	(dependent on record format)	Data values for the subrecord.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

This interface allows an application to randomly access a subrecord for output. The data being written consists of an integral number of units, where a unit is dependent on the organization of state data for the family. If the family is **`m_object_ordered`**, then a unit is the vector of state variables for a single mesh object, and the total quantity of units for the subrecord is equal to the number of nodes or elements bound to the subrecord definition (or one if the subclass bound to the subrecord is "`Global`"; see `mf_def_subrec()`). If the family is **`m_result_ordered`**, then a unit is a result array consisting of all instances of a single state variable for the current state and subrecord. In this latter case, the quantity of units for the subrecord is equal to the number of state variables bound to the subrecord definition. [Note: if a state variable is an array or list array, then a result array for a **`m_result_ordered`** family is an array of arrays (list arrays).]

It is important to note that `start` and `stop` are order numbers, not numeric identifiers of particular nodes or elements. The mapping between order numbers and identifiers is defined when the subrecord is formatted with `mf_def_subrec()`.

All units written in one call to `mf_wrt_subrec()` must consist of data of the same type. This is not a consideration for **`m_object_ordered`** families, since for those families all state variables are required to be of the same base type. But for **`m_result_ordered`** families, this does preclude packing integer data alongside floating point data for a single output operation.

This interface can be used in conjunction with `mf_wrt_stream()` to write state data for a single state. However, it should be noted that a call to `mf_wrt_stream()` immediately following a call to `mf_wrt_subrec()` will write data immediately following the last unit written by `mf_wrt_subrec()`.

Example:

See also:

`mf_def_subrec()`
C API: `mc_wrt_subrec()`
File mili_fparam.h

3.1 MILI FORTRAN TI API

This section describes all functions implemented in Mili release 1.11 that support reading and writing time independent data (TI).

mf_ti_check_arch()

Checks to make sure that TI files were generated on same architecture as the Mili mesh files. A status of -1 is returned if there is an architecture conflict.

Synopsis:

```
mf_ti_check_arch( fam_id, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Example:

```
integer fam_id, status

mf_ti_check_arch( fam_id, status )
mf_print_err( 'mf_ti_check_arch in myprog', status )
```

See also:

C API: `mc_ti_get_metadata()`
File: ti.h

mf_ti_def_class()

This function will define TI data class attributes for all subsequent writing. It only applies to writing.

Synopsis:

```
mf_def_class( fam_id, meshid, state, matid, superclass,
              meshvar, nodal, short_name, long_name, status
              )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`meshid`	Input	INTEGER	Mili mesh identifier.
`state`	Input	Integer	State at which data item was written.
`matid`	Input	Integer	Material number for TI number is applicable.
`superclass`	Input	CHARACTER*n	Superclass of data item.
`meshvar`	Input	Bool_type	If TRUE then this data items is a mesh variable and of mesh length– either nodal or an element field.
`nodal`	Input	CHARACTER*n	If TRUE then this item is a nodal variable, other wise it is an element variable. This field is only relevant if meshvar is also TRUE.
`short_name`	Input	CHARACTER*n	Short class name.
`long_name`	Input	CHARACTER*n	Long class name.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

This function is also named `mc_ti_set_class()` – the arguments are the same as for `mc_ti_def_class(). If the TI variables to be written are not associated with a state, class, or superclass, then these fields should be set to NULL.`

Example:

See also:

`mc_ti_get_class()`

C API: `mc_ti_def_class()`

File: ti.h

mf_ti_disable()

This function will prevent future writing of TI data during a run. It will just return (doing nothing) if TI reading/writing is not enabled.

Synopsis:

```
mf_ti_disable( void )
```

Arguments: None

Notes:

This function would be called if you want to stop writing TI data in the middle of a run.

Example:

```
mc_ti_enable_only()
```

See also:

`mf_ti_enable()`
`mf_ti_enable_only()`
C API: `mc_ti_disable()`
File ti.h

mf_ti_enable()

Synopsis:

```
mf_ti_enable( void )
```

Arguments: None

Notes:

This function would be called if you want to start writing TI data along with the Mili mesh data.

Example:

See also:

`mf_ti_disable()`
`mf_ti_disable_only()`
C API: `mc_ti_enable()`
File ti.h

mf_ti_enable_only()

Synopsis:

```
mf_ti_enable_only( void )
```

Arguments: None

Notes:

This function would be called if you want to start writing TI data but no mesh data (only TI files will be generated if this function is called).

Example:

See also:

`mf_ti_enable()`
`mf_ti_disable()`
C API: `mc_ti_enable_only()`
File ti.h

mf_ti_get_data_attributes()

This function will return all class attributes for a TI data field.

Synopsis:

```
mf_ti_get_data_attributes( fam_id, meshid, name, class,
                           state, matid, superclass,
                           meshvar, nodal, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`meshid`	Input	INTEGER	Mesh Id for this class
`name`	Input	CHARACTER*_n_	Name of the TI variable.
`class`	Input	CHARACTER*_n_	Class name for the TI variable.
`state`	Output	INTEGER	State that TI variable was written.
`matid`	Output	INTEGER	Material associated with TI variable *(if any)*.
`superclass`	Output	Integer	Superclass of data item.
`IsMeshvar`	Output	Bool_type	If TRUE then this data items is a mesh variable and of mesh length– either nodal or an element field.
`IsNodal`	Output	CHARACTER*_n_	If TRUE then this item is a nodal variable, other wise it is an element variable. This field is only relevant if meshvar is also TRUE.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

This function is also named `mc_ti_get_class()` – the arguments are the same as for `mc_ti_get_data_attributes().` If the TI variable is not associated with a mesh then meshid should be set to a -1.

Example:

See also:

`mf_ti_set_class()`
`mf_ti_get_class()`
C API: `mc_ti_def_class()`

File: ti.h

mf_ti_htable_search_wildcard()

Searches for variables in the TI hashtable using up to three keys. A match occurs if the TI name contains a partial match to one of the three keys input. The matches are returned in *return_list*. If list_len is >0 then matches are returned at position *list_len*. This allows for multiple searches to be appended to the same list.

Synopsis:

```
mf_ti_htable_search_wildcard( fam_id, list_len,
                      allow_duplicates,
                      key1, key2, key3, return_list,
                      status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`list_len`	Input	INTEGER	Current length of result list.
`allow_duplicates`	Input	Bool_type	If TRUE then duplicate names are returned if duplicates are found.
`key1, key2, key3`	Input	CHARACTER*_n_	Search keys.
`return_list`	Input/ Output	CHARACTER*_n[]_	List of names found that match the search keys.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

This function is also named `mc_htable_search_wildcard()` – the arguments are the same as for `mc_ti_htable_search_wildcard()`.
The function `mc_htable_search_wildcard()` search the standard Mili hash tables. The caller must allocate space for the return list.

See also:

`mf_htable_search_wildcard()`
C API: `mc_ti_htable_search_wildcafrd()`
File: ti.h

mf_ti_read_array()

Reads an array of any type (except character strings) from the TI data file.

Synopsis:

```
mf_ti_read_array( fam_id, name, value, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of TI array to read.
`value`	Output	INTEGER	Pointer to the array read.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

See also:

`mf_ti_wrt_array()`
`mf_ti_def_class()`
C API: `mc_ti_read_array()`
File: ti.h

mf_ti_read_scalar()

Reads a scalar of any type (except character strings) from the TI data file.

Synopsis:

```
mf_ti_read_scalar( fam_id, name, value, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of TI variable to read.
`value`	Output	INTEGER	Pointer to the variable read.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

See also:

`mf_ti_wrt_array()`
`mf_ti_def_class()`
C API: `mc_ti_read_scalar()`
File: ti.h

mf_ti_read_string()

Reads a string variable from the TI data file.

Synopsis:

```
mf_ti_read_scalar( fam_id, name, value, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of TI variable to read.
`value`	Output	CHARACTER*n	Pointer to the variable read.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

See also:

`mf_ti_wrt_string()`
`mf_ti_def_class()`
C API: `mc_ti_read_string()`
File: ti.h

mf_ti_undef_class()

This funtion resets the current TI class definition that was set with the last call to `mf_ti_def_class()`.

Synopsis:

```
mf_ti_undef_class( fam_id, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

Example:

See also:

`mf_ti_def_class()`
`mf_ti_get_class()`
C API: `mc_ti_undef_class()`
File ti.h

mf_ti_wrt_array()

Writes an array of any type (except character strings) to the TI data file using the currently defined class.

Synopsis:

```
mf_ti_wrt_array( fam_id, type, name, value, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`type`	Input	Any Mili number type	Data type of the scalar to be written.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of TI array.
`value`	Input	INTEGER	Pointer to the array to write.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

See also:

`mc_ti_read_array()`
`mc_ti_def_class()`
C API: `mc_ti_wrt_array()`
File: ti.h

mf_ti_wrt_scalar()

Writes a single scalar of any type to the TI data file using the currently defined class.

Synopsis:

```
mf_ti_wrt_scalar( fam_id, type, name, value, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`type`	Input	Any Mili number type	Data type of the scalar to be written.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of TI scalar.
`value`	Input	INTEGER	Scalar to write.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

See also:

C API: `mc_ti_wrt_scalar()`
`mc_ti_read_scalar()`
`mc_ti_def_class()`
File: ti.h

mf_ti_wrt_string()

Writes a single character string to the TI data file using the currently defined class.

Synopsis:

```
mf_ti_wrt_string( fam_id, name, value, status )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of TI scalar.
`value`	Input	CHARACTER*n	String to write.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

See also:

`mc_ti_read_string()`
`mc_ti_def_class()`
C API: `mc_ti_wrt_string()`
File: ti.h

Appendix A

Mili Defined Constants

FORTRAN constants

- Data types

Name	Meaning
m_string	Character string
m_real	Floating point (size of REAL)
m_real4	Floating point, 4-byte
m_real8	Floating point, 8-byte
m_int	Integer (size of INTEGER)
m_int4	Integer, 4-byte
m_int8	Integer, 8-byte

- Object superclasses

Name	Meaning
m_unit	Generic object superclass
m_node	Node
m_truss	Truss
m_beam	Beam
m_tri	Triangle
m_quad	Quadrilateral or Shell
m_tet	Tetrahedron
m_pyramid	Pyramid
m_wedge	Wedge or Prism
m_hex	Hexahedron or Brick
m_mat	Material subdomain
m_mesh	Entire mesh
m_surface	A surface area of a mesh

- State data organization variations

Name	Meaning
m_object_ordered	Object-ordered state records
m_result_ordered	Result-ordered state records

- Database information categories

Name	Meaning
`m_state_data`	All data described by state record formats
`m_non_state_data`	All data which is not `m_state_data`

- Argument constraints

Name	Meaning
`m_max_svars`	Maximum number of state variables defined in one call to `mf_def_svars()`; value = 200
`m_max_array_dims`	Maximum number of array dimensions; value = 6
`m_max_name_len`	Maximum length (bytes) of a name string; value = 256
`m_max_path_len`	Maximum length (bytes) of a directory path specification string; value = 128
`m_max_preamble_len`	Maximum length (bytes) of a `mf_print_err()` preamble string; value = 128
`m_max_string_len`	Maximum length (bytes) of a named string written with `mf_wrt_string()`; value = 512

- Element enumeration formats for `mf_def_subrec()`

Name	Meaning
`m_list_obj_fmt`	Element numbers listed explicitly in 1D array
`m_block_obj_fmt`	Elements listed implicitly by sequence of first elem/last elem ranges in 2-by-n array

- Database query request types for `mf_query_family()`

Name	Information requested
`m_class_superclass`	Superclass of a class
`m_class_exists`	Existence (yes/no) of named class
`m_lib_version`	Library version string
`m_multiple_times`	Simulation times for multiple unordered states
`m_next_state_loc`	Location indices for state following input state
`m_qty_class_in_sclass`	Quantity of classes in a superclass
`m_qty_dimensions`	Dimensionality of mesh(es) in family
`m_qty_elem_conn_defs`	Quantity of element connectivity definition entries for a particular class
`m_qty_elems_in_def`	Quantity of elements referenced in a particular element connectivity definition entry
`m_qty_meshes`	Quantity of meshes defined in a family
`m_qty_node_blks`	Quantity of node definition blocks for a particular nodal object class
`m_qty_nodes_in_blk`	Quantity of nodes referenced in a particular

Name	Information requested
	node definition block
m_qty_srec_fmts	Quantity of state record formats defined in a family
m_qty_states	Quantity of state data records currently written to a family
m_qty_subrecs	Quantity of subrecords defined for a particular state record format
m_qty_subrec_svars	Quantity of state variables bound to a subrecord definition
m_qty_svars	Quantity of state variables defined in a family
m_series_srec_fmts	State record format identifiers for a continuous series of state records
m_series_times	Simulation times for a continuous series of state records
m_srec_fmt_id	State record format identifier for a particular state record
m_srec_mesh	Mesh identifier associated with a state record format
m_state_of_time	Indices of states bounding an input time value
m_state_time	Simulation time for a particular state record
m_subrec_class	Class name for objects in a subrecord

- Family file partitioning methods

Name	Meaning
m_state_limit	Partition state-data files on the basis of the quantity of states written to a file; non-state files are unlimited

- Miscellaneous

Name	Meaning
m_blank	Placeholder for character parameters that won't be evaluated
m_dont_care	Placeholder for numeric parameters that won't be evaluated

Appendix B

Detailed Mili Example

Consider the simple three-dimensional mesh shown in Figure 3. It consists of two hexahedral volume elements, each defined by eight nodal connectivities, and four truss elements, each defined by two nodal connectivities, with the nodes numbered as shown. Assume a computational analysis of the mesh under some load is to generate data, which will be written out using a single state record format. The output data is to include the position of the nodes, a six-component stress tensor for the volume elements, and axial force for the truss elements.

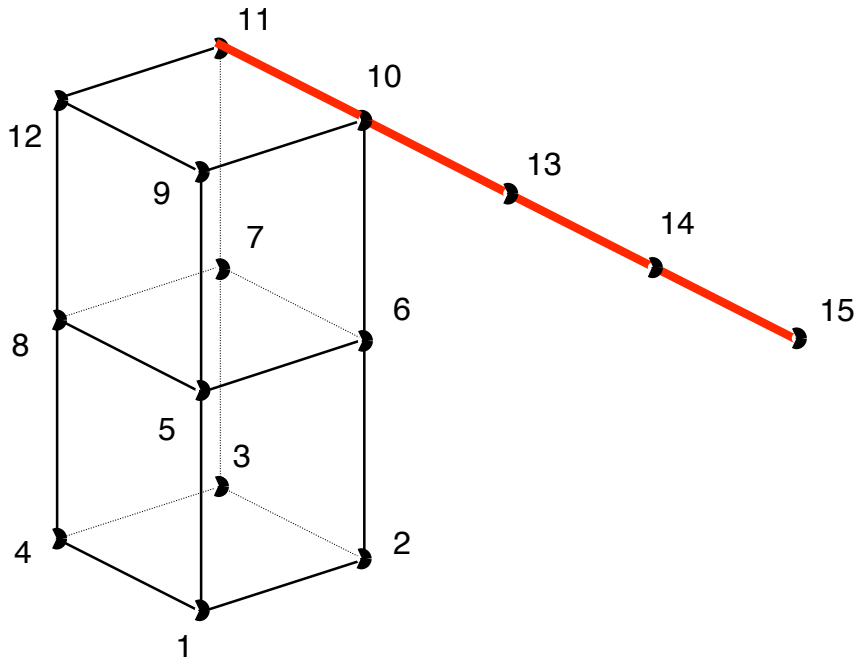


Figure 3. Simple mesh with Hex and Truss elements

A traversal of Figure 2 in consideration of this mesh produces the following sequence of FORTRAN calls (for the sake of illustration, assume the analysis will complete after three iterations):

Write Example (Fortran)

```
c.....State variables per mesh object type
c.....All short names MUST be unique
c.....Nodal state variables
      character*(m_max_name_len) nodal_short(12), nodal_long(12)
      data
     +    nodal_short /
     +   'nodpos', 'nodvel', 'nodacc', 'ux', 'uy', 'uz',
     +        'vx', 'vy', 'vz', 'ax', 'ay', 'az' /,
     +    nodal_long /
     +        'Node Position', 'Node Velocity', 'Node Acceleration',
     +        'X Position', 'Y Position', 'Z Position',
     +        'X Velocity', 'Y Velocity', 'Z Velocity',
     +        'X Acceleration', 'Y Acceleration', 'Z Acceleration' /

c.....Hex state variables
      character*(m_max_name_len) hex_short(8), hex_long(8)
      data
     +    hex_short /
     +        'stress', 'eeff', 'sx', 'sy', 'sz', 'sxy', 'syz',
      +       'szx' /,
      +    hex_long /
      +        'Stress', 'Eff plastic strain', 'Sigma-xx', 'Sigma-yy',
      +        'Sigma-zz', 'Sigma-xy', 'Sigma-yz', 'Sigma-zx' /

c.....Truss state variables
      character*(m_max_name_len) truss_short(1), truss_long(1)
      data
     +    truss_short /
     +        'stress'/,
          truss_long /
      +       'Stress'/

c.....State var types; they're all floats, so just define one array as big
c.....as the largest block of svars that will be defined in one call
      integer svar_types(7)
      data svar_types / 7*m_real /

c.....Group names, to be used when defining connectivities and when
c.....defining subrecords.
      character*(m_max_name_len) global_s, global_l, nodal_s, nodal_l,
     +                           hexs_s, hexs_l, material_s, truss_s,
     +                           truss_l, material_l
          data global_s, global_l, nodal_s, nodal_l, hexs_s, hexs_l,
     +    material_s, material_l /
     +    'g', 'Global', 'node', 'Nodal', 'h', 'Bricks', 'm', 't',
     +    'Truss', 'Material' /

c....MESH GEOMETRY
c.....Nodes 1:15
      real node_1_15(3, 15)
      data node_1_15/
    + 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0. 0.0,
    + 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0. 1.0,
    + 0.0, 0.0, 2.0, 1.0, 0.0, 2.0, 1.0, 1.0, 2.0, 0.0, 1.0. 2.0,
```

```
     + 2.0, 1.0, 2.0,
     + 3.0, 1.0, 2.0,
     + 4.0, 1.0, 2.0/

c.....Hexes 1:2
c.....  8 nodes Part#=1 / Material#=2
c
       integer hex_1_2(10, 2)
       data hex_1_2/
     + 1,2,3,4,5,6,7,8,1,1,    ! Hex1 (Part 1, Mat 1)
     + 5,6,7,8,9,10,11,12,1,2/ ! Hex2 (Part 1, Mat 2)
     + 1,2,3,4,5,6,7,8,1,1, ! Truss1 (Part 1, Mat 3)

c.....Trusses 1:4
c..... 2 nodes per truss Part#=1 / Material#=3
c
       integer truss_1_4(10, 4)
       data truss_1_4/
     + 11,10,1,3, ! Truss1 (Part 1, Mat 3)
     + 10,13,1,3, ! Truss2 (Part 1, Mat 3)
     + 13,14,1,3, ! Truss3 (Part 1, Mat 3)
     + 14,15,1,3/ ! Truss4 (Part 1, Mat 3)

c.....Open a Mili datafile for output
       integer fid
       call mf_open( 'test_problem', 'data_dir' 'AwPs', fid, stat )
       If (stat .ne. 0)
           call mf_print_error('mf_open - Write', stat)
           stop
       endif

c.....MUST define state variables prior to binding subrecord definitions

c      Note: 1) Re-use of svar_types array since everything is of type m_real
c            2) Variables are defined in groups by object type only so that
c               the "short" name arrays can be re-used during subrecord
c               definitions.

       call mf_def_vec_svar( fid, m_real, 3, nodal_short(1),
     +                       nodal_long(1), nodal_short(4),
     +                       nodal_long(4), stat )
       call mf_print_error( 'mf_def_vec_svar', stat )

       call mf_def_vec_svar( fid, m_real, 3, nodal_short(2),
     +                       nodal_long(2), nodal_short(7),
     +                       nodal_long(7), stat )
       call mf_print_error( 'mf_def_vec_svar', stat )

       call mf_def_vec_svar( fid, m_real, 3, nodal_short(3),
     +                       nodal_long(3), nodal_short(10),
     +                       nodal_long(10), stat )
       call mf_print_error( 'mf_def_vec_svar', stat )
       call mf_def_vec_svar( fid, m_real, 6, hex_short(1),
     +                       hex_long(1), hex_short(3),
     +                       hex_long(3), stat )
       call mf_print_error( 'mf_def_vec_svar', stat )
       call mf_def_svars( fid, 1, svar_types, hex_short(2),
     +                    hex_long(2), stat )
```

```
       call mf_print_error( 'mf_def_svars', stat )

c.....MUST create & define the mesh prior to binding subrecord definitions
c.....Initialize the mesh.
      call mf_make_umesh( fid, 'Hex mesh', 3, mesh_id, stat )
      call mf_print_error( 'mf_make_umesh', stat )

c.....Create the mesh object classes by deriving the classes from
c.... superclasses.

c.....Create simple mesh object classes
      call mf_def_class( fid, mesh_id, m_mesh, global_s, global_l, stat )
      call mf_print_error( 'mf_def_class (global)', stat )

      call mf_def_class_idents( fid, mesh_id, global_s, 1, 1, stat )
      call mf_print_error( 'mf_def_class_idents (global)', stat )

c.....Define the material class.
      call mf_def_class( fid, mesh_id, m_mat, material_s, material_l, stat )
      call mf_print_error( 'mf_def_class (material)', stat )

      call mf_def_class_idents( fid, mesh_id, material_s, 1, 3, stat )
      call mf_print_error( 'mf_def_class_idents (material)', stat )

c.....Create complex mesh object classes.  First create each class from a
c……...superclass as above, then use special calls to supply appropriate
c……….attributes.
c…. (i.e., coordinates for nodes and connectivities for elements).
c.....Define a node class.
      call mf_def_class( fid, mesh_id, m_node, nodal_s, nodal_l, stat )
      call mf_print_error( 'mf_def_class (nodal)', stat )
      call mf_def_nodes( fid, mesh_id, nodal_s, 1, 16, node_1_16, stat )
      call mf_print_error( 'mf_def_nodes', stat )

c.....Define element classes
c.....Connectivities are defined using either of two interfaces,
c     mf_def_conn_arb() or mf_def_conn_seq().  The first references
c     element identifiers explicitly in a 1D array (in any order);
c     the second references them implicitly by giving the first and last
c     element identifiers of a continuous sequence.
c     Note that all connectivities for a given element type do not have
c     to be defined in one call (for example., hexs).
c     NO checking is performed to verify that nodes referenced in
c     element connectivities have actually been defined in the mesh.

c.....Define a hex class
      call mf_def_class( fid, mesh_id, m_hex, hexs_s, hexs_l, stat )
      call mf_print_error( 'mf_def_class (hexs)', stat )
      call mf_def_conn_seq( fid, mesh_id, hexs_s, 1, 2,
     +                      hex_1_2, stat )
      call mf_print_error( 'mf_def_conn_seq', stat )

c.....Define a truss class
      call mf_def_class( fid, mesh_id, m_truss, truss_s, truss_l, stat )
      call mf_print_error( 'mf_def_class (truss)', stat )
      call mf_def_conn_seq( fid, mesh_id, truss_s, 1, 4,
     +                      truss_1_4, stat )
      call mf_print_error( 'mf_def_conn_seq', stat )
```

```
c.....MUST create and define the state record format
c.....Create a state record format descriptor.
      call mf_open_srec( fid, mesh_id, srec_id, stat )
      call mf_print_error( 'mf_open_srec', stat )

c....Nodal data
      obj_blocks(1,1) = 1
      obj_blocks(2,1) = 15
      call mf_def_subrec( fid, srec_id, nodal_l, m_result_ordered, 3,
     +                    nodal_short, nodal_s, m_block_obj_fmt, 1,
     +                    obj_blocks, stat )
      call mf_print_error( 'mf_def_subrec', stat )

c....Hex data, brick class
      obj_blocks(1,1) = 1
      obj_blocks(2,1) = 2
      call mf_def_subrec( fid, srec_id, hexs_l, m_object_ordered, 2,
     +                    hex_short(1), hexs_s, m_block_obj_fmt, 1,
     +                    obj_blocks, stat )
      call mf_print_error( 'mf_def_subrec', stat )
      call mf_close_srec( fid, srec_id, stat )
      call mf_print_error( 'mf_close_srec', stat )

c....Truss data, Truss class
      obj_blocks(1,1) = 1
      obj_blocks(2,1) = 4
      call mf_def_subrec( fid, srec_id, truss_l, m_object_ordered, 2,
     +                    truss_short(1), truss_s, m_block_obj_fmt, 1,
     +                    obj_blocks, stat )
      call mf_print_error( 'mf_def_subrec', stat )
      call mf_close_srec( fid, srec_id, stat )
      call mf_print_error( 'mf_close_srec', stat )

c...SHOULD commit before writing state data so the family will be
c...readable if a crash occurs during the analysis.
      call mf_flush( fid, m_non_state_data, stat )
      call mf_print_error( 'mf_commit', stat )
c     call mf_limit_states( fid, 500, stat )
c     call mf_print_error( 'mf_limit_states', stat )

c....Write three state records, each with one call.
      time = 0.0
      call mf_new_state( fid, srec_id, time, suffix, st_idx, stat )
      call mf_print_error( 'mf_new_state', stat )

      call mf_wrt_stream( fid, m_real, 165, state_record1, stat )
      call mf_print_error( 'mf_wrt_stream', stat )

      time = 1.0
      call mf_new_state( fid, srec_id, time, suffix, st_idx, stat )
      call mf_print_error( 'mf_new_state', stat )

      call mf_wrt_stream( fid, m_real, 165, state_record2, stat )
      call mf_print_error( 'mf_wrt_stream', stat )

      time = 2.0
```

```
      call mf_new_state( fid, srec_id, time, suffix, st_idx, stat )
      call mf_print_error( 'mf_new_state', stat )
      call mf_wrt_stream( fid, m_real, 165, state_record3, stat )
      call mf_print_error( 'mf_wrt_stream', stat )

c.....MUST close the Mili family when finished
      call mf_close( fid, stat )
      call mf_print_error( 'mf_close', stat )
```

Read Example (C)

** The reader functionality using *mc_read_state_data()* and *mc_read_non_state_data()* will not be operational until release Mili 8.1

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include "mili_internal.h"
int
main( int argc, char *argv[] )
{
    int fid;
    Bool_type status;
    Mili_analysis *dbid;

    fid = dbid->db_ident;

    /* Open a Mili database for input */
    status = mc_open( “test”, “datadir”, "r", &fid );

   /* Read non-state data (mesh and constants) */
   status = mc_read_non_state_data( &dbid );

   /* Read non-state data (mesh and constants) */
   status = mc_read_state_data( state, &dbid );

   mc_close( fid );
}
```