UCRL-SM-236144

Mili I/O Library

Programmer's Reference Manual

Authors/Developers

Kevin Durrenberger

Alumni

Ivan R. Corey

Elsie Pierce

Doug Speck

Version 13.1

July 1, 2013

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
8.1	March 20, 2009	I. R. Corey	Minor Edits – Fixed section headers and changed version number from 2.1 to 8.1
9.1	July 1, 2009	I. R. Corey	Added Mili Reader Functions(mr_*)
10.1	August 18, 2010	IR.Corey, Kevin Durrenberger	Performance improvements, enhancements to Mili read capability, Windows support, bug fixes.

Introduction

Mili is a high-level mesh I/O library and database developed to support computational analysis and post-processing on unstructured finite element meshes. Mili provides is self-describing database 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. Mili has both C and FORTRAN interfaces and provides for database portability by supporting both big- and little-endian binary formats.

1 Getting Started

1.1 External library requirements

Mili is a self-contained library and requires no external libraries to build.

1.2 Compiler Requirements

Mili supports building with the following compilers.

Compiler	Executable
Intel	icc and ifort
AIX	xlc and xlf
Gnu	gcc and gfortran

Table 1: Compiles and Executabes

1.3 Configuration Tools

Mili requires GNU configure(version 2.63 or later) to setup and create the Makefile. It also requires gmake (We currently use GNU Make 3.81), however earlier version should work.

2 Configuring and Building Mili.

Creating the Mili library consists of 2 steps, configuration and build

2.1 Configuration

In the most basic mode using the *configure* straight out of the box is sufficient. By default it will choose the compilers in the following order of Intel, AIX and GNU. If you wish to set a specific compiler then the standard configure flags CC={C compiler} and F77={fortran compiler}. For a complete list of available options run *configure --help* (see Appendix A). After configuration is complete Mili has created a subdirectory based on the system and login node with MILI- prepended to it. For example, if you had a Linux based system on a login node named zeus42 your file created would be MILI-Linux-ZEUS.

2.2 Building a Mili Database Access Library

Once configuration is complete, you will then need to build the code. You must change into the directory created by configure to run the build. After changing into this directory you can build either a debug or optimized version of the code.

gmake opt
or
gmake debug

For a complete list of build options see Appendix B

3 Creating a Mili Database

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 the Mili access library will also automatically convert data to the native endianness of the host platform when reading a database. The ability to set these parameters is defined in the function declaration for mf_open(). For the impatient, the entire source code to create a dataase can be found in Appendix C.

3.1 Initializing a database

Creating a Mili database starts with one function call mf_open() or mc_open(). However there is much after that which requires handling in specific order. The following diagram shows the process one must follow to create a Mili database using the Mili library API.

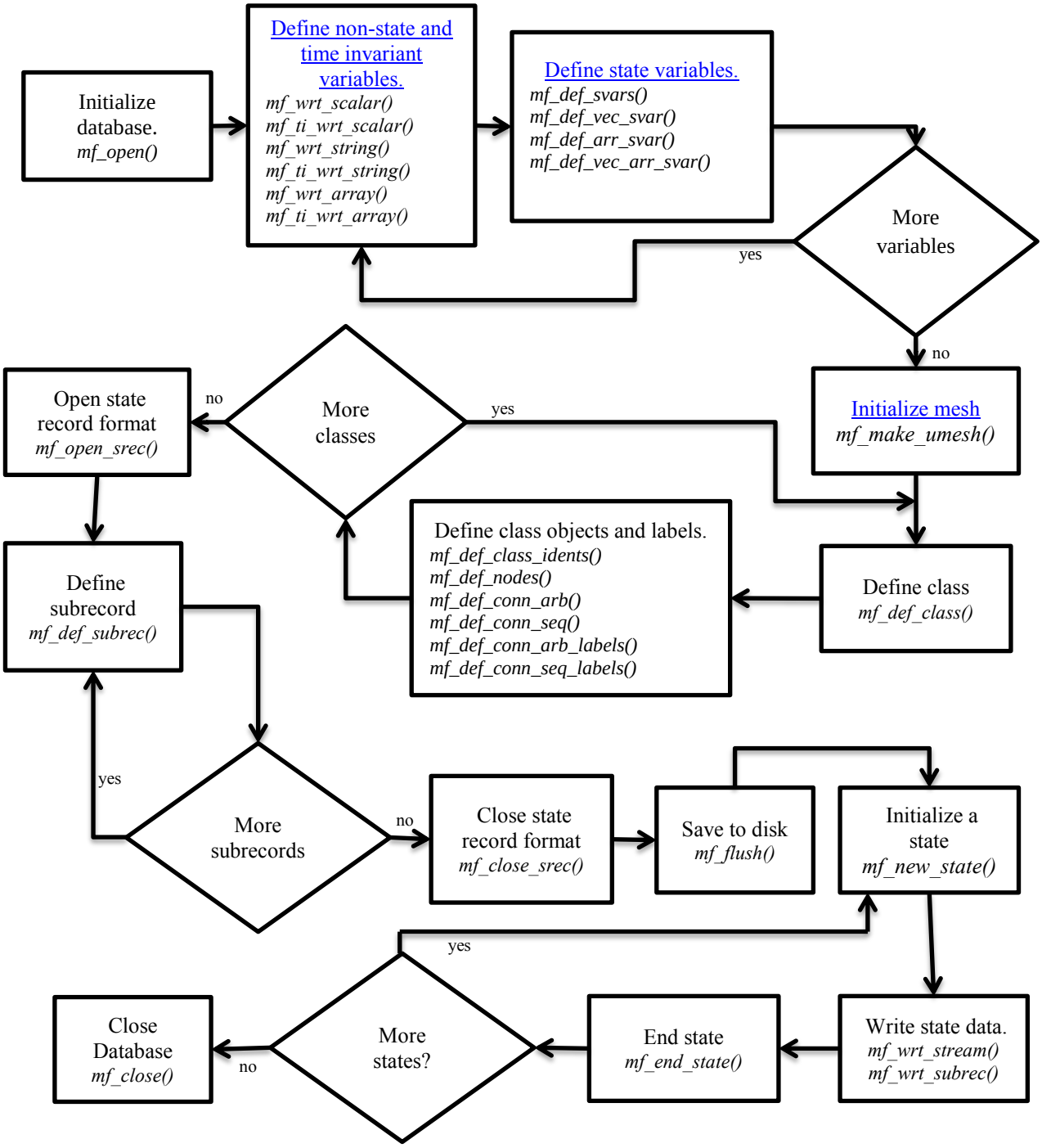


Table 2: Mili Creation Flow Chart

For our examples as we work through the process of building a Mili database we will use the following mesh. Which will consist of 15 nodes, 2 hex (brick) elements (black), and 4 truss elements (red). The full code to create this mesh is in appendix C.

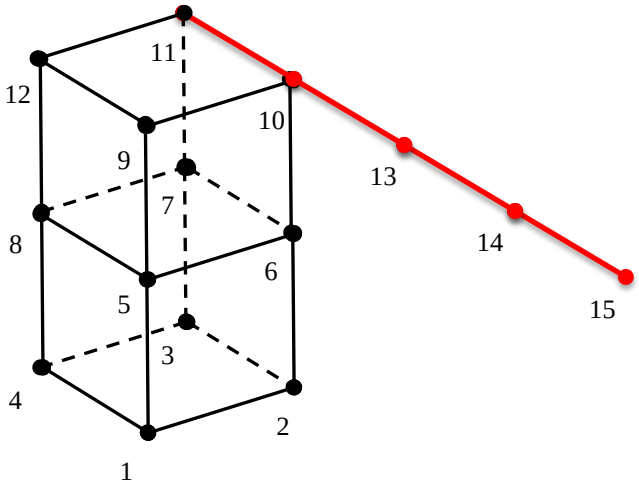


Figure 1: Simple mesh with Node, Hex and Truss elements

The first step will be to open the file for writing. We will create a file name "plot1" in the directory "/usr/files".

FORTRAN call:
mf_open("plot1", "/usr/files", "Aw", fam_id, status)

C call:
status = mc_open("plot1", "/usr/files", "Aw", fam_id);

Note: I will only the C function for the mf_open call. Afterwards, I will only show the FORTRAN calls and leave the reference to the function call to show the C calls.

This opens a file /usr/files/plot1 in write mode (Aw) with single precision. A complete list of the control string can be found in table 4. The value fam_id is an integer and will be used by all future calls to Mili to reference the correct database. The parameter status is set within Mili and should be checked on return for a valid function execution which if successful. Upon successful completion all functions should return 0. After any function call the following code should be done to check if the call was successful. Depending on the call you may want to clean up and exit or just clean up any allocated memory and continue.

FORTRAN call:

```
if (status .ne. 0)
```

```
    call mf_print_error("mc_open " , status)
    !Handle your cleanup and return
 endif
```

C call:

```
if( status != OK) {
   mc_print_error("mc_open", status);
   //Handle your clean up and return
}
```

OK is a constant defined in Mili and is equal to zero.

This check to confirm that a database was instantiated and the user should exit if this returns a none zero status as any subsequent call to Mili functions using the fam_id will return with an error telling the user that no such database exists.

We now have an open database for writing.
Some additional functions that you can use at this time prior to declaring your varibales are:

- mf_limit_states()
- mf_limit_filesize()

3.2 Defining Variables

3.2.1 Defining Non-state Variables

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.

In previous version of Mili there were Non-state variables and what is listed as time invariant (TI) data. The TI data is still available in the 13.1 release and the functions used to define them are still available so that existing code can still use the existing calls. The major change in the step is the removal of the files that end in TI_A, TI_B and so on. We determined that the TI variables were basically non-state variables and did not need the additional file in order to work.

There are some non-state variable created by Mili and internally use the same calls to create these "metadata" variables. They include the following

- lib version -- The version of the mili interface library used to create this database.
- host name -- the host name this database was created on. Usually a node name.
- arch name -- Created by calling uname –s
- date -- Original date of the creation of this database.
- username -- The person's account name that created the file
- states per file -- As states are created and added to the database this is updated.
- max size per file -- default is 10Mb, otherwise this is set by user.
- xmilics version -- If this is a combined database, it is the XmiliCS version.

Xmilics is the Livermore MDG codes combining tool for combining files created in parallel.

Some common things that are included in the databases are the title of the mesh and the number of processors used (nproc). I will use these to show examples on how to write what we call Mili parameters.

From the above open example we have the fam_id which was returned from Mili. We will need to use this when calling the write functions.

```
mf_wrt_string(fam_id,"title",title_text,status)

!as always we check the return status

if (status .ne. 0)
   call mf_print_error("mc_wrt_str(title) " , status)
  !Handle your cleanup and possibly return
endif

!the ti calls as of version 13.1 are interchangeable with the regular write calls
mf_ti_wrt_scalar(fam_id, m_int, "nproc", nproc, status)

if (status .ne. 0)
  call mf_print_error("mc_ti_wrt_scalar(nproc) " , status)
  !handle your cleanup and possibly return
endif
```

3.2.2 Defining State Variables (svar)

A state variable is simple any parameters that you will want to use in a given state. For example items such node positions may be written. In order to do so you must first define the variable using one of four different function calls: mf_def_svars(), mf_def_vec_svar(), mf_def_arr_svar(), mf_def_vec_arr_svar().

In our example problem we want to put out node positions, velocities and accelerations. We can define them using either mf_def_svars() and mf_def_vec_svars()

```
! define the names we are going to use.
! node 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' /


!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' /

!global state variables

 character*(m_max_name_len) global_short(1), global_long(1)
 data  global_short /'ke'/,
+      global_long /'Kinetic Energy'/

!Define using mf_def_vec_svars(). This is our preferred method.

call mf_def_vec_svar( fid, m_real, 3, nodal_short(1),
     +                       nodal_long(1), nodal_short(4),
     +                       nodal_long(4), stat )
if(stat .ne. 0)
    call mf_print_error( 'mf_def_vec_svar', stat )
    !clean up and return
endif

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

if(stat .ne. 0)
    call mf_print_error( 'mf_def_vec_svar', stat )
    !clean up and return
endif

call mf_def_vec_svar( fid, m_real, 3, nodal_short(3),
     +                       nodal_long(3), nodal_short(10),
     +                       nodal_long(10), stat )

if(stat .ne. 0)
    call mf_print_error( 'mf_def_vec_svar', stat )
    !clean up and return
endif

!or define the positions, velocities and accelerations as separate svars
!using mf_def_svars
```

```
call mf_def_svars( fid, m_real, 3, nodal_short(3), nodal_long(3),stat )

if(stat .ne. 0)
    call mf_print_error( 'mf_def_vec_svar', stat )
    !clean up and return
endif

call mf_def_svars( fid, m_real, 3, nodal_short(7), nodal_long(7),stat )

if(stat .ne. 0)
    call mf_print_error( 'mf_def_vec_svar', stat )
    !clean up and return
endif

call mf_def_svars( fid, m_real, 3, nodal_short(10), nodal_long(10),stat )

if(stat .ne. 0)
    call mf_print_error( 'mf_def_vec_svar', stat )
    !clean up and return
endif
```

We will define the additional using only one method

```
call mf_def_vec_svar( fid, m_real, 6, hex_short(1),
     +                      hex_long(1), hex_short(3),
     +                      hex_long(3), stat )

if(stat .ne. 0)
    call mf_print_error( 'mf_def_vec_svar', stat )
    !clean up and return
endif

call mf_def_svars( fid, m_real, 1, global_short(1), global_long(1),stat )

if(stat .ne. 0)
    call mf_print_error( 'mf_def_svars', stat )
    !clean up and return
endif
```

There are reasons for either method. The first method allows you to assign all the node positions to a subrecord by just giving the name “nodpos”. This is our code’s normal way of defining state variables, but you can define them however you would like.

Continue adding whatever state variables that you need (i.e. stress). Although it is not encouraged, you may define as many state variables that you want without using all of them. We recommend that you simply create whichever state variables that your mesh will require.

Now that you have defined you state variables it is now time to create a mesh and state record.

3.3 Defining a Mesh

Mili supports only unstructured meshes. To create a mesh is a single call which will return the mesh_id of the created mesh. The mesh_id is used in all subsequent calls to creating a mesh and state records.

```
call mf_make_umesh( fid, 'Hex mesh', 3, mesh_id, stat )

if(stat .ne. 0)
    call mf_print_error( 'mf_def_vec_svar', stat )
    !clean up and return
endif
```

We now need to define our classes based upon a Mili superclass. We can equate this to inheritance in many modern programing languages. The superclasses are in actuality just integer values to tell Mili how to read and store the incoming and outgoing data. Each Mili superclass has a number of node connections plus a material id as well as a part number. If the superclass has node connectivity, it is always given in the form of an array of size n = Node Connectivity + 2 with the first n-2 values being the nodes and the last 2 values being the material id and part number in that order. A Mili superclass can consist of both of mesh and non-mesh object shown in the table below.

FORTRAN class name	C class name	Node Connectivity	Object type
m_unit	M_UNIT	n/a	Arbitrary
m_node	M_NODE	1	Node
m_truss	M_TRUSS	2	Truss
m_beam	M_BEAM	3	Oriented beam
m_tri	M_TRI	3	Triangle
m_quad	M_QUAD	4	Quadrilateral or Shell
m_tet	M_TET	4	Tetrahedron
m_pyramid	M_PYRAMID	5	Pyramid
m_wedge	M_WEDGE	6	Wedge or Prism
m_hex	M_HEX	8	Hexahedron or Brick
m_mat	M_MAT	n/a	Material
m_mesh	M_MESH	n/a	Mesh
m_surface*	M_SURFACE	n/a	A surface containing other types of mesh objects.
m_particle	M_PARTICLE	1	Single Node Particle

Table 3 Mili object superclasses
*not yet implemented

Although the ordering of the connections is not enforced by Mili (It would be prohibitive for Mili to analyze all connectivities passed to it), the general rule of ordering is as follows:

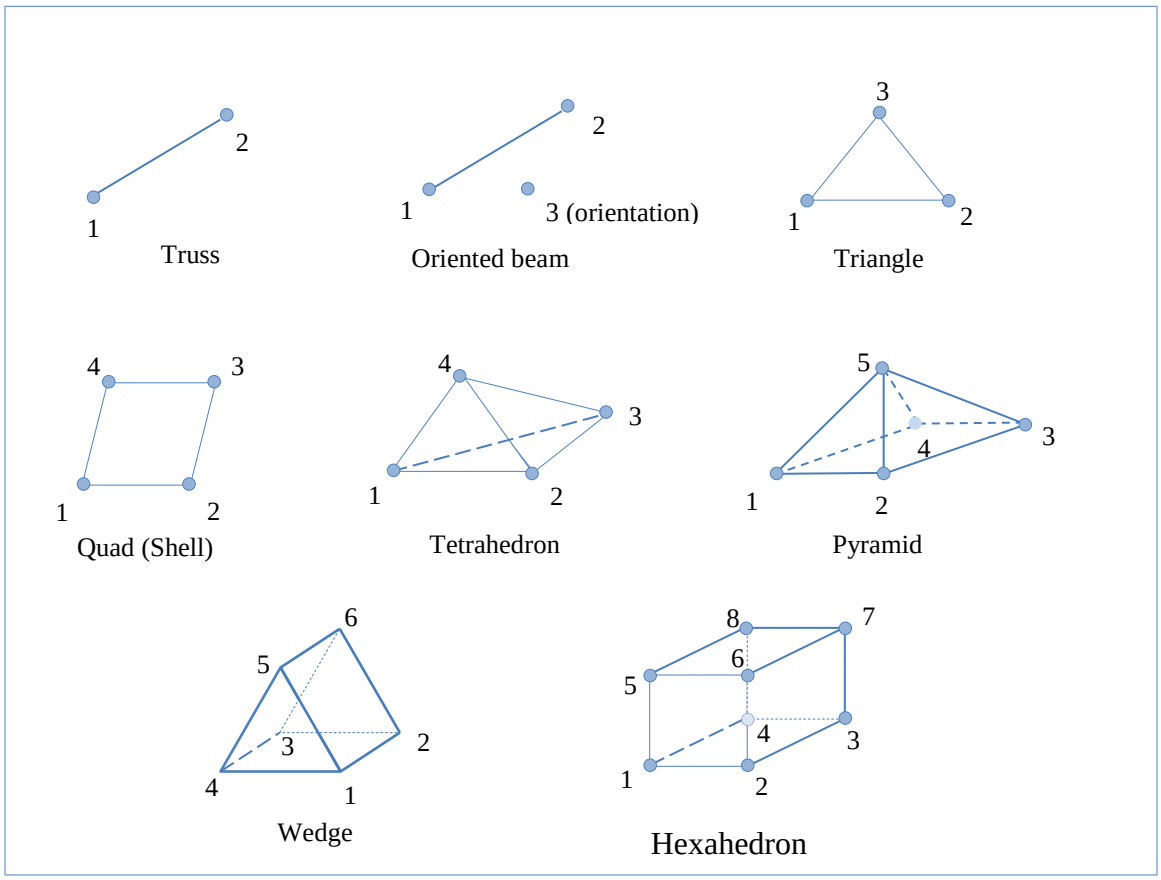


Figure 2: Node ordering

Now that we have the pieces required to create our mesh we can begin.

```
! Node definition, connectivities, and labels

 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/
```

```
 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)
+ 5,6,7,8,9,10,11,12,2,1/ ! Hex2 (Part 1, Mat 2)
```

```
integer hex_labels(2)

data hex_labels / 10,20 /

 integer truss_1_4(10, 4)
 data truss_1_4/
+ 11,12,2,1, ! Truss1 (Part 1, Mat 2)
+ 12,13,2,1, ! Truss1 (Part 1, Mat 2)
+ 13,14,2,1, ! Truss2 (Part 1, Mat 2)
+ 14,15,2,1, ! Truss3 (Part 1, Mat 2)

integer truss_labels(4)

data truss_labels / 1,2,3,4 /
```

Using our example from the beginning we start by defining the classes and the identities of each element that we will need.

```
!This will associate the shortname "node" with the m_node superclass

call mf_def_class( fid, mesh_id, m_node, nodal_s, nodal_l, stat )

! Nodes are the only connectivity which has a special function associated
! with and it is as follows:

call mf_def_nodes( fid, mesh_id, nodal_s, 1, 15, node_1_15, stat )

if(stat.ne.0)
      call mf_print_error( 'mf_def_nodes', stat )
endif

! globals are a special class that will apply thare values to all objects
! in the mesh

call mf_def_class( fid, mesh_id, m_mesh, global_s, global_l, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_class(global', stat )
endif

call mf_def_class_idents( fid, mesh_id, global_s, 1, 1, stat )

if(stat.ne.0)
    call mf_print_error( 'mf_def_class_idents (global)', stat )
endif

!.....Define the material class. You need at least one material

call mf_def_class( fid, mesh_id, m_mat, material_s, material_l, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_class (material)', stat )
```

```
endif

call mf_def_class_idents( fid, mesh_id, material_s, 1, 3, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_class_idents (material)', stat )
endif
```

All other mesh objects use the functions mf_def_conn_arb(), mf_def_conn_seq(), and mf_def_labels().It is now more common to use labels when defining connectivities using mf_def_conn_arb_labels() and mf_def_conn_seq_labels(). A label is an integer number assigned to an object to help in labelling that object. It can be any arbitrary number, but you cannot repeat anumber for a specific mech object.

```
!.....Define a hex class

call mf_def_class( fid, mesh_id, m_hex, hexs_s, hexs_l, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_class (hexs)', stat )
endif

call mf_def_conn_seq_label( fid, mesh_id, hexs_s, 1, 2,
     +                      hex_labels, hex_1_2, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_conn_seq', stat )
endif



!.....Define a truss class

call mf_def_class( fid, mesh_id, m_truss, truss_s, truss_l, stat )

if(stat.ne.0)
call mf_print_error( 'mf_def_class (truss)', stat )
endif

call mf_def_conn_seq( fid, mesh_id, truss_s, 1, 4,
     +                     truss_labels, truss_1_4, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_conn_seq', stat )
endif
```

That is the end of defining our mesh. The next task will be to define the state record.

3.4 Creating a State Record

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. Again, we will continue with our code from above.

```
call mf_open_srec( fid, mesh_id, srec_id, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_conn_seq', stat )
endif
```

The srec_id is passed back to the user to be used in the subrecord definitions and when the state data is written out.

When begin with our nodal data. We define the blocks of object ids that this subrecord applies to. In this case we are defining all of the “node” elements to the subrecord named “Nodal”. What this allows us to do is to group the x, y, and z nodal positions, velocities, and accelerations under a single subrecord. Remember we defined the nodpos, nodvel and nodacc as vector svars.

```
 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 )

if(stat.ne.0)
  call mf_print_error( 'mf_def_subrec(nodal_l)', stat )
endif
```

The m_result_ordered is telling Mili that we want to write the data in result order. What this means is that when the data is written out at each state we will write out all the node positions, then the node velocities, and then all the node accelerations. 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 3 graphically illustrates the difference between result_ordered and object ordered data.

Object Ordered							
Var1 Elem 1	Var 2 Elem 1	Var1 Elem 2	Var2 Elem 2	Var1 Elem 3	Var2 Elem 3	Var1 Elem 4	Var2 Elem 4
Result Ordered							
Var1 Elem 1	Var 1 Elem 2	Var1 Elem 3	Var1 Elem 4	Var2 Elem 1	Var2 Elem 2	Var2 Elem 3	Var2 Elem 4

Figure 3: State record organizations. 4 elements and 2 variables

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.

We will continue on defining the additional subrecords now.

```
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 )

 if(stat.ne.0)
  call mf_print_error( 'mf_def_subrec(hex_l)', stat )
 endif

 call mf_close_srec( fid, srec_id, stat )

 if(stat.ne.0)
   call mf_print_error( 'mf_close_srec()', stat )
 endif

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 )

 if(stat.ne.0)
   call mf_print_error( 'mf_def_conn_seq', stat )
 endif
```

3.5 Writing a State

Writing out state data can occur in 2 different ways. You must choose one one of them as they cannot be mixed. The first method is to use mf_wrt_stream() . This method puts the burden of data alignment for an entire state on the external code. It is the responsibility of the code to write out the data in the exact same order that the subrecords were delared. The second method for writing out state data is using the mf_wrt_subrec(). While this still requires that the user align the data correctly for the subrecord itself, it relieves the code the responsibility of aligning the data for the entire state. In either case the code is responsible for writing out all the pertinent data for each time step, even if it is just dummy data to avoid NaN's when reading the data in.

Our state records will be written from the following array which we are showing here. Of course normally these qould have been declared in the common block area of the FORTRAN code. We are just duplicating the arrays of data for simplicity.

```
 real state_record1(149)
 data state_record1/
+ 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, ! end of nodes positions
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, !end of node velocities
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ! end of node accelerations
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0/       ! end of "Stress" Subrecord

 real state_record2(149)
 data state_record2/
+ 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, ! end of nodes positions
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, !end of node velocities
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ! end of node accelerations
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0/       ! end of "Stress" Subrecord
 real state_record3(149)
 data state_record3/
+ 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, ! end of nodes positions
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, !end of node velocities
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ! end of node accelerations
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
+ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0/       ! end of "Stress" Subrecord
```

c....Write three state records, each with one call.

```
 time = 0.0
 call mf_new_state( fid, srec_id, time, suffix, st_idx, stat )

 if(stat.ne.0)
   call mf_print_error( 'mf_def_conn_seq', stat )
 endif

 call mf_wrt_stream( fid, m_real, 149, state_record1, stat )

 if(stat.ne.0)
   call mf_print_error( 'mf_def_conn_seq', stat )
 endif

 call mf_end_state(fid, srec_id, stat)

 if(stat.ne.0)
   call mf_print_error( 'mf_def_conn_seq', stat )
 endif

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

 if(stat.ne.0)
   call mf_print_error( 'mf_def_conn_seq', stat )
 endif

 call mf_wrt_stream( fid, m_real, 149, state_record2, stat )

 if(stat.ne.0)
```

```
  call mf_print_error( 'mf_def_conn_seq', stat )
endif

call mf_end_state(fid, srec_id, stat)

if(stat.ne.0)
  call mf_print_error( 'mf_def_conn_seq', stat )
endif

time = 2.0
call mf_new_state( fid, srec_id, time, suffix, st_idx, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_conn_seq', stat )
endif

call mf_wrt_stream( fid, m_real, 149, state_record3, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_conn_seq', stat )
endif

call mf_end_state(fid, srec_id, stat)

if(stat.ne.0)
  call mf_print_error( 'mf_def_conn_seq', stat )
endif
```

3.6 Closing a Database

When you finished your last state you will want to close the database and Mili clean up it memory footprint. This is a single call in Mili

```
call mf_close( fid, stat )

if(stat.ne.0)
  call mf_print_error( 'mf_def_conn_seq', stat )
endif
```

You now have a complete database. Although a very simple database, this illustrated the basic functions of how to create a database.

3.7 Appending to an existing database

Sometimes it becomes neesssary to restart a database. Either the time allotment on the code expired or the code ended abnormally prior to finishing. Mili has some limited capabilities for restaring at a specific timestep. To accomplish this you must know the state that you want to

restart at and the structure of the database must not have changed. We open the database in the append mode.

```
integer restart_state
integer fid
integer stat

restart_state = 3

call mf_open( 'test_problem', 'data_dir' 'Aa', fid, stat )

if (stat .ne. 0)
   call mf_print_error('mf_open - append', stat)
   stop
endif
```

Then you will need to set the timestep. The negative 1 tell Mili to find the state within the various state files for you. Previously, Mili required the user to know the state file index that was associated with the specific timestep.

```
mf_restart_at_state( fid, -1, restart_state, stat)

if (stat .ne. 0)
    call mf_print_error('mf_restart_at_state', stat)
    stop
endif
```

After that you would start writing the states as normal.

4 Reading a Database

Extracting data from a Mili database has been challenging for most codes. Initially we had one tool to open and read a database, that tool was Griz. The integration of Griz and Mili were and still are tightly coupled. I will provide some methods in the next sections to extract data from Mili at as high of a level that Mili provides currently.

For our examples we will use the C interface.

4.1 Opening a Database

To open a database for reading is simply to request it open in read only mode. This is done by passing the option "Ar" in the open function.

```
int fid;
int stat;
```

```
stat mc_open( 'test_problem', 'data_dir' 'Ar', &fid);

if (stat != OK)
{
   mc_print_error('mf_open - read', stat)
   exit(0)
}
```

You now have a database open for reading.

4.2 Getting Non-mesh Data

4.2.1 Retrieving Mili Parameters

If you remember from the database creation, Mili Parameters are those parameters that really have nothing to do with mesh itself, but are rather just information parameters pertaining to the creation of this database (i.e. title or mili version). There are really 2 approaches to extracting the data, first if you know what the database contains and you know exactly what you want to extract and second if you have no idea what the database contains and you will be just extracting everything.

4.2.1.1 Extracting known parameter data.

In this case you know exactly what the data is that you are looking for in the database. We will use 'title' as an example. 'title' is a known parameter of type string(char*).

```
char title[M_MAX_STRING_LEN]

stat = mc_read_string( dbid, "title",title );

if(stat != OK)
{
  mc_print_error("mc_read_string- title", stat);
  // Don't exit as it is not really error to have no title
}
```

Continue reading known variables with any of the following functions:

```
Return_value mc_read_scalar( /* Read a named scalar from the file family */
   Famid fam_id,             /* Mili family identifier */
   char *name,               /* Label of scalar */
   void *value );            /* Pointer to scalar */

Return_value mc_read_string( /* Read a named string from the file family */
   Famid fam_id,             /* Mili family identifier */
   char *name,               /* Label of string */
   char *value );            /* String data */
```

That is it for the parameters if you know the specific parameters and the type associated with them.

4.2.1.2 Extracting unknown parameter data

This process is a little cumbersome but pretty straight forward.

```
int num_entries;
int index;
int type;
int datalength;
int status;
char **param_name;

int *data_int = NULL;
long *data_long = NULL;
char *data_char = NULL;
float *data_float = NULL;
double *data_double = NULL;

/*Get the number of parameters. The "*" tells it to match all parameters*/
/*Note that passing NULL as the final parameter forces the function to
perform a count and not try to copy the data over*/
num_entries = mc_htable_search_wildcard(dbid, 0, FALSE,
                                        "*", "NULL", "NULL",
                                        NULL);

param_names = (char **) malloc(num_entries*sizeof(char *));

if (num_entries > 0 && param_names == NULL)
{
   return ALLOC_FAILED;
}

/* Load the parameter names */
num_entries = mc_htable_search_wildcard(dbid, 0, FALSE,
                                        "*", "NULL", "NULL",
                                        param_names);

/* You now have a list of all the Mili Parameters */

/* Now loop over the parameter names and extract the data */

for(index = 0; index < num_entries; index++)
{
  status = mc_mr_get_param_attributes(dbid, param_name[index], FALSE,
                                      &type, &datalength);
  if(status != OK)
  {
    mc_print_error("mc_mr_get_param_attributes()", status);
```

```
    continue; /*maybe exit if needed*/
  }

  switch (datatype) {
      case (M_INT):
         data_int = (void *) malloc( datalength*sizeof(int) );
         status = mc_read_scalar(fam_id, name,(void*)data_int);
         /* Check the return and process the data */
         free(data_int);
         break;

      case (M_INT8):
         data_long = (void *) malloc( datalength*sizeof(long) );
         status = mc_read_scalar(fam_id, name,(void*)data_long);
         /* Check the return and process the data */
         free(data_long);
         break;

      case (M_STRING):
         data_char = (void *) malloc( datalength*sizeof(char) );
         status = mc_read_string(fam_id,name,data_char);
         /* Check the return and process the data */
         free(data_char);
         break;

      case (M_FLOAT):
      case (M_FLOAT4):
         data_float = (void *) malloc( datalength*sizeof(float) );
         status = mc_read_scalar(fam_id, name,(void*)data_float);
         /* Check the return and process the data */
         free(data_float);
         break;

      case (M_FLOAT8):
         data_double = (void *) malloc( datalength*sizeof(double) );
         status = mc_read_scalar(fam_id, name,(void*)data_double);
         /* process the data */
         free(data_double);
         break;
   } /* switch */

}
```

5 Mili FORTRAN and C API

5.1 **mf_close()**

Close an open Mili file family.

Declaration:

FORTRAN:

mf_close(fam_id, status)

C:

int status mc_close(int fam_id)

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.

5.2 **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.

Declaration:

FORTRAN:
mf_close_srec(fam_id, srec_id, status)

C:
int status mc_close_srec(int fam_id, int srec_id)

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.

5.3 **mf_def_arr_svar()**

Define an array state variable.

Declaration:

FORTRAN:
mf_def_arr_svar(fam_id, type, order, dimensions, name, title, status)

C:
return int status
mc_def_arr_svar(int fam_id, int type, int *order, int *dimensions, char *name, char *title)

Parameters:

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. It must be unique and should be a brief, mnemonic handle for the array.
title	Input	CHARACTER*n ($n \leq$ **m_max_name_len**)	State variable long name. It should be a handle suitable for labeling a plot or menu.
status	Output	INTEGER	Returns 0 on success, else non-zero error code.

Note:

Although each state variable name 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 /

mf_def_arr_svar(fam_id, m_real, 3, dimensions, 'ev', 'Strain Vector', status)

```
if(status .ne. 0)
{
   mf_print_err( 'mf_def_arr_svar in myprog', status )
   !clean up memory and return
}
```

5.4 **mf_def_class()**

Define a mesh object class from a superclass.

Declaration:

FORTRAN:
mf_def_class(fam_id, mesh_id, superclass, short_name,
long_name, status)

C:
int status
mc_def_class(int fam_id, int superclass, char* short_name,
char* long_name)

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.

5.5 **mf_def_class_idents()**

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

Declaration:

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

C:
```
int status mc_def_class_idents( int fam_id, int mesh_id, char* class_name,
                                int start, int stop)
```

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().

5.6 **mf_def_conn_arb()**

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

Declararion:

FORTRAN:
mf_def_conn_arb(fam_id, mesh_id, class, qty, elem_ids,
connects, status)

C:
int status mc_def_conn_arb(int fam_id, int mesh_id, char* class, int qty,
int *elem_ids, int* connects)

Arguments:

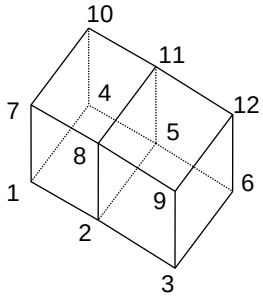
fam_id	Input	INTEGER	Mili family identifier.
mesh_id	Input	INTEGER	Mesh identifier.
class	Input	CHARACTER*n ($n \leq$ **m_max_name_len**)	Name of element class. Must have been previously defined with a call to mf_def_class(). The superclass associated with class must be one of the *element* mesh object types.
qty	Input	INTEGER	Number of 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 )
```

5.7 **mf_def_conn_arb_labels()**

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

Declaration:

FORTRAN:
mf_def_conn_arb_labels(fam_id, mesh_id, class, qty, elem_ids, labels, connects, status)

C:
int status mc_def_conn_arb_labels(int fam_id, int mesh_id, char* class, int qty, int* elem_ids, int* labels, int* connects)

Arguments:

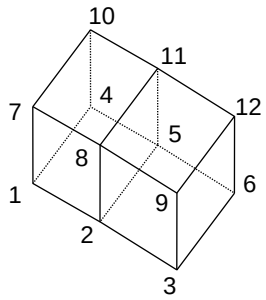
fam_id	Input	INTEGER	Mili family identifier.
mesh_id	Input	INTEGER	Mesh identifier.
class	Input	CHARACTER*n ($n \leq$ **m_max_name_len**)	Name of element class. Must have been previously defined with a call to mf_def_class(). The superclass associated with class must be one of the *element* mesh object types.
qty	Input	INTEGER	Number of elements referenced in array elem_ids.
elem_ids	Input	INTEGER array	Array of element identifiers. Size: (qty)
labels	Input	INTEGER array	Array of element labels. 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_labels() is identical to subroutine mf_def_conn_seq_labels() except for the way in which element identifiers are specified. In mf_def_conn_arb_labels(), the elements are listed explicitly in array elem_ids and may be randomly ordered. See mf_def_conn_seq_labels() 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 /
data labels / 100, 200 /
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_labels( fam_id, mesh_id, elem_class, 2,
     +                  elem_ids, labels, connects, status )

mf_print_err( 'mf_def_conn_arb_labels in myprog', status )
```

5.8 **mf_def_conn_labels()**

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

Declaration:

FORTRAN:
mf_def_conn_labels(fam_id, mesh_id, class, qty, labels, status)

C:
int status mc_def_conn_labels(int fam_id, int mesh_id, char* class,
int qty, int* labels)

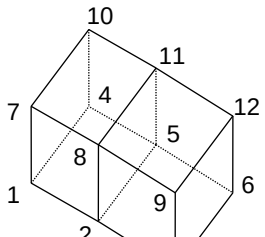
Arguments:

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 )
```

5.9 **mf_def_conn_seq_labels()**

Define element connectivity's with associated labels for a continuously numbered sequence of elements in an unstructured mesh.

Declaration:

FORTRAN:

```
mf_def_conn_seq_labels( fam_id, mesh_id, class, start_elem,
                        stop_elem, labels, connects, status )
```

C:

```
int status mc_def_conn_seq_labels( int fam_id, int mesh_id, char* class,
                                   int start_elem, int stop_elem, int* labels,
                                   int* connects)
```

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.
labels	Input	INTEGER array	Array of element label numbers.
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:

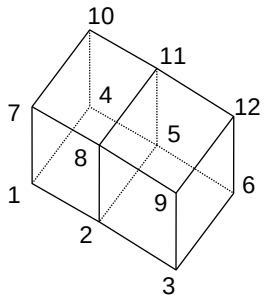
The purpose of subroutine mf_def_conn_seq() is to specify element connectivities and associated labels for an element class. Element classes provide a way of having separately numbered (i.e., one-to-n) groups of elements, which share the same element superclass (**m_hex**, for example). This allows Mili 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:

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:

Example:

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




```
integer fam_id, mesh_id, status
integer connects(10,2), labels(10)
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_labels( fam_id, mesh_id, m_hex, elem_class,
   + 1, 2, labels, connects, status )

if ( status .ne. 0) mf_print_err( 'mf_def_conn_seq in myprog', status )
```

5.10 **mf_def_conn_seq()**

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

Declaration:

FORTRAN:

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

C:

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

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.

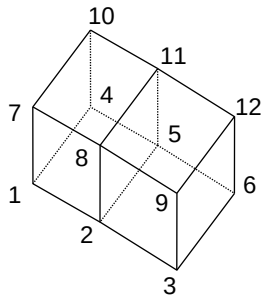
Notes:

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:

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 )
if( status .ne. 0) mf_print_err( 'mf_def_conn_seq in myprog', status )
```

5.11 **mf_def_conn_surface()** *Currently this is not implemented.*

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

Declaration:

FORTRAN:

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

C:

```
int status mc_def_conn_surface(int  fam_id,int  mesh_id,int  short_name,
                               int qty_facets, int*  conns,int  surf_id )
```

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.
`conns`	Input	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:

5.12 **mf_def_nodes()**

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

Declaration:

FORTRAN:
mf_def_nodes(fam_id, mesh_id, class_name, start_node, stop_node, coords, status)

C:
int status mc_def_nodes(int fam_id, int mesh_id, char* class_name, int start_ node,
int stop_node, int* coords)

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 )

if (status .ne. 0) mf_print_err( 'mf_def_nodes in myprog', status )
```

5.13 **mf_def_node_labels()**

Define nodal labels.

Declaration:

FORTRAN:
mf_def_node_labels(fam_id, mesh_id, class_name, qty, labels, status)

C:
int status mc_def_node_labels(int fam_id, int mesh_id, char* class_name,
int qty, int* labels)

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. If labels are not used a 1 to N ordering will be assumed.

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 )

if( status .ne. 0) mf_print_err( 'mf_def_node_labels in myprog', status )
```

5.14 **mf_def_subrec()**

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

Declaration:

FORTRAN:

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

C:

```
int status mc_def_subrec(int  fam_id, int srec_id, char* name, int org,
                         int qty_st_vars, char** st_var_names,
                         char* class,  int format, int qty, int* object_ids )
```

Arguments:

fam_id	Input	INTEGER	Mili family identifier.
srec_id	Input	INTEGER	State record format identifier from mf_open_srec().
name	Input	CHARACTER*n ($n \leq$ **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 ($n \leq$ **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 ($n \leq$ **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

5.15 **mf_def_surf_subrec()** *Not yet Implemented*

Define state subrecords on a surface mesh object.

Declaration:

```
FORTRAN:
mf_def_surf_subrec( fam_id, srec_id, name, org, qty_svars,
                    svar_names, class, idformat, qty,
                    obj_ids, flag, status )

C:

int status
mc_def_surf_subrec( int fam_id, int srec_id, char* name,
                    int org, int qty_svars, char** svar_names,
                    char* class, int idformat, int qty,
                    int* obj_ids, int* flag )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`srec_id`	Input	INTEGER	State record format identifier from `mf_open_srec()`.
`name`	Input	CHARACTER*n ($n \leq$ **`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_svars`	Input	INTEGER	Quantity of state variables bound to subrecord definition.
`svar_names`	Input	CHARACTER*n array ($n \leq$ **`m_max_name_len`**)	Array of state variable short names. Variables must have been previously defined for the family. Size: (`qty_svars`).
`class`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Element class short name.
`idformat`	Input	INTEGER	Format of mesh object identifiers passed in array `obj_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 obj_ids (for format equal to **m_list_obj_fmt** or **m_block_obj_fmt**, respectively).
obj_ids	Input	INTEGER array	If format is **m_list_obj_fmt** then obj_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.
flag	Input	INTEGER array	Surface variable display flags, either NULL (implies all are PER_OBJECT) or array of PER_OBJECT/PER_FACET values.
status	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

This is largely a wrapper for mf_def_subrec(). The difference is that the surface variable flag is supplied in this function (which defines subrecords on a surface mesh object) but is not supplied in mf_def_subrec(). The notes for mf_def_subrec() apply to this function as well.

5.16 **mf_def_svars()**

Define one or more scalar state variables.

Declaration:

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

C:
int status
mc_def_svars( int fam_id, int qty_vars, int* types,
              char** names, char** titles )
```

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 ($n \leq$ **`m_max_name_len`**)	Short names for each of the scalar variables. Size: (`qty_vars`)
`titles`	Input	CHARACTER*n array ($n \leq$ **`m_max_name_len`**)	Long names for each of the scalar variables. 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 )
```

5.17 **mf_def_vec_arr_svar()**

Define a vector array state variable.

Declaration:

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

C:

int status
mc_def_vec_arr_svar( int fam_id, int type, int order,
                     int* dimensions, int field_qty,
                     char* name, char* title,
                     char** field_names, char** field_titles )
```

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 ($n \leq$ **`m_max_name_len`**)	State variable short name.
`title`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	State variable long name.
`field_names`	Input	CHARACTER*n array ($n \leq$ **`m_max_name_len`**)	Short names for each of the fields. Size: (`field_qty`)
`field_titles`	Input	CHARACTER*n array ($n \leq$ **`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 )
```

5.18 **mf_def_vec_svar()**

Define a vector state variable.

Declaration:

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


C:
int status
mc_def_vec_svar( int fam_id, int type, int field_qty,
                 char* name, char* title, char** field_names,
                 char** field_titles )
```

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 ($n \leq$ **`m_max_name_len`**)	State variable short name.
`title`	Input	CHARACTER*n ($n \leq$ **`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 )
```

5.19 **mf_delete_family()**

Delete a Mili file family.

Declaration:

```
FORTRAN:
mf_delete_family( root_name, path, status )
C:
int status
mc_delete_family( root_name, path )
```

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.

5.20 **mf_filelock_disable()** *Deprecated*

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.

Declaration:

```
mf_filelock_disable( void )
```

Arguments: None.

5.21 **mf_filelock_enable()** *Deprecated*

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_filelock_disable().*

File locking with is removed from Mili completely in future releases since Mili does not support multiple-writers to the same file family.

Declaration:

```
FORTRAN:
mf_filelock_enable( void )

C:
mc_filelock_enable( )
```

Arguments: None.

5.22 **mf_flush()**

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

Declaration:

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

C:
int status
mf_flush( int fam_id, int file_type )
```

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.

5.23 **mf_get_class_info()**

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

Declaration:

```
FORTRAN:
mf_get_class_info( fam_id, mesh_id, superclass,
                   class_index, short_name, long_name,
                   object_qty, status )
C:
int status
mc_get_class_info( int fam_id, mesh_id, superclass,
                   class_index, short_name, long_name,
                   object_qty )
```

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.

5.24 **mf_get_metadata()**

Returns metadata fields for a file family.

Declaration:

```
FORTRAN:
mf_get_metadata( fam_id, mili_version, host, arch,
                 timestamp, xmilics_version, status )
C:
int status
mc_get_metadata( int fam_id, char* mili_version,
                 char* host, char* arch,
                 char* timestamp, char* xmilics_version )
```

Arguments:

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

Notes:

Utility function

5.25 **mf_get_mesh_id()**

Return the numeric identifier for the named mesh.

Declaration:

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

C:
int status
mc_get_mesh_id( int fam_id, char* name, int* mesh_id)
```

Arguments:

fam_id	Input	INTEGER	Mili family identifier.
name	Input	CHARACTER*n ($n \leq$ **m_max_name_len**)	Mesh name.
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 )
```

5.26 **mf_get_node_label_info()**

Returns information needed to set up call to mf_get_node_labels.

Declaration:

```
FORTRAN:
mf_get_node_label_info( fam_id, mesh_id, class_name, num_blocks,
                        num_labels, status )
C:
int status
mc_get_node_label_info( int fam_id, int mesh_id,
                        char* class_name, int* num_blocks,
                        int* num_labels )
```

Arguments:

fam_id	Input	INTEGER	Mili family identifier.
mesh_id	Input	INTEGER	Unstructured mesh identifier
class_name	Input	CHARACTER*n (n ≤ **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.
num_blocks	Output	INTEGER array	Number of tuples for each block range: {1,n} {1,m} ……
num_labels	Output	INTEGER array	Number of nodal labels.
status	Output	INTEGER	Returns 0 on success, else non-zero error code.

5.27 **mf_get_svar_mo_ids_on_class()**

Returns the mesh object ids associated with the requested svar defined on the requested class. This information is state independent and only needs to be called once. State variable information is state dependent and will need to be gathered once per state of interest.

Declaration:

```
FORTRAN:
mf_get_svar_mo_ids_on_class( fam_id, class_name, var_name,
                             blocks, status )
C:
int status
mc_get_svar_mo_ids_on_class( int fam_id, char* class_name,
                             char* var_name, int* blocks )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`class_name`	Input	CHARACTER*n ($n \leq$ **m_max_name_len)**	Object class name.
`var_name`	Input	CHARACTER*n ($n \leq$ **m_max_name_len)**	State variable name.
`blocks`	Output	INTEGER array	Length 2**num_blocks* where *num_blocks* is returned by mf_get_svar_size. A tuple for each block range $\{I, j\}$, $\{k, l\}$….
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

5.28 **mf_get_svar_on_class()**

Returns the state variable values for the requested variable associated with the requested class at the specified state.

Declaration:

```
FORTRAN:
mf_get_svar_on_class( fam_id, state, class_name, var_name, data,
status)

C:
int status
mc_get_svar_on_class( int fam_id, int state, char* class_name,
                      char* var_name, float* data)
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`state`	Input	INTEGER	State number (one-base) from which to read data.
`class_name`	Input	CHARACTER*n ($n \leq$ **m_max_name_len)**	Object class name.
`var_name`	Input	CHARACTER*n ($n \leq$ **m_max_name_len)**	State variable name.
`data`	Output	real*n	Length *size* of type *type* where *size* and *type* are returned by mf_get_svar_size.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

5.29 **mf_get_svar_size()**

Returns the number of values, their data type, and the number of mesh object blocks for the specified state variable associated with the specified class.

Declaration:

```
FORTRAN:
mf_get_svar_size( fam_id, class_name, var_name, num_blocks,
                  size, type, status )

C:
int status
mc_get_svar_size( int fam_id, char* class_name,
                  char* var_name, int* num_blocks,
                  int* size, int* type )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`class_name`	Input	CHARACTER*n ($n \leq$ **m_max_name_len)**	Object class name.
`var_name`	Input	CHARACTER*n ($n \leq$ **m_max_name_len)**	State variable name.
`num_blocks`	Output	INTEGER	Number of mesh object blocks associated with the specified variable.
`size`	Output	INTEGER	Number of data values associated with the specified variable.
`type`	Output	INTEGER	Type of data values associated with the specified variable.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

5.30 **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.

Declaration:

```
FORTRAN:
mf_get_simple_class_info( fam_id, mesh_id, superclass,
                          class_index, short_name, long_name,
                          start_ident, stop_ident, status )
C:
int status
mc_get_simple_class_info( int fam_id, int mesh_id,
                          int superclass, int class_index,
                          char* short_name, char* long_name,
                          int start_ident,int  stop_ident )
```

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.

5.31 **mf_limit_filesize()**

Set the maximum number of bytes allowable in any state-data file. Mili will start a new file once the maximum size is reached. This means that the size will be larger than the filesize limit set by this function.

Declaration:

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

C:
int status
mc_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.

5.32 **mf_limit_states()**

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

Declaration:

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

C:
int status
mf_limit_states( int fam_id, int limit)
```

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.

5.33 **mf_load_conns()**

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

Declaration:

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

C:
int status
mf_load_conns( int fam_id, int mesh_id, char* class_name,
               int* connects, int* materials,
               int* parts )
```

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.

5.34 **mf_load_conn_labels()**

Read in element labels for a specified class.

Declaration:

```
FORTRAN:
mf_load_conn_labels( fam_id, mesh_id, class_name, qty,
                     num_blocks, block_range, element_list,
                     labels, status)
C:
int status
mc_load_conn_labels( int fam_id, int mesh_id, char* class_name,
                     int qty, int* num_blocks, int* block_range,
                     int* element_list, int* labels)
```

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()`.
`qty`	Input	INTEGER	Number of labels expected.
`num_blocks`	Output	INTEGER	Number of label blocks where a block is a contigious range of labels.
`block_range`	Output	INTEGER array	Length 2*num_blocks. A tuple for each block range: {1,n} {1,m} ……
`element_list`		INTEGER array	Length qty. Local element index numbers for each label.
`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), element_list(4), qty

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

! Open Mili family...

qty = 4

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,
+qty,num_blocks, block_range, element_list, labels,
+status )

mf_print_err( 'mf_def_conn_labels in myprog', status )
```

5.35 **mf_load_nodes()**

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

Declaration:

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

C:
int status
mc_load_nodes( int fam_id, int mesh_id, char* class_name,
               float** coords)
```

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.

5.36 **mf_load_node_labels()**

Read in nodal labels.

Declaration:

```
FORTRAN:
mf_load_node_labels( fam_id, mesh_id, class_name, block_range,
                     labels, status )
C:
int status
mf_load_node_labels( int fam_id, int mesh_id, char* class_name,
                     int* block_range, int* labels )
```

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`.
`block_range`	Output	INTEGER array	Length 2*num_blocks. A tuple for each block range: {1,n} {1,m} ……
`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 but recommended. If you are creating parallel files then you should use labels. 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 )
```

5.37 **mf_load_surface()** ***Not implemented***

Reads in the connectivity definition for the specified surface number.

Declaration:

```
FORTRAN:
mf_load_surface( fam_id, mesh_id, surf_id, short_name,
                 conns, status )
C:
int status
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.
`short_name`	Input	CHARACTER*n (*n* <= **m_max_name_len**)	Surface object class.
`surf_id`	Input	INTEGER	Surface id number.
`conns`	Output	Integer array	Array of nodal connection numbers.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

5.38 **mf_make_umesh()**

Create an empty unstructured mesh descriptor.

Declaration:

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

C:
int status
mc_make_umesh( int fam_id, char* name, int dims, int* mesh_id )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Mesh name.
`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.

5.39 **mf_new_state()**

Prepare a family to receive data for a new state.

Declaration:

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

C:
int status
mc_new_state( int fam_id, int srec_id, float time,
              int* file_sequence, int* file_state_index )
```

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.

5.40 **mf_open()**

Open a Mili file family.

Declaration:

```
FORTRAN:
mf_open( root_name, path, control, fam_id, status )
C:
int status
mf_open( char* root_name, char* path, char* control,
         int* fam_id )
```

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.
`control`	Input	CHARACTER*n ($n \leq$ **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 data stored in family (higher precision state variables are automatically converted); *applicable only when a family is created and fixed thereafter*	d	Double precision (64 bits)	Always
		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

Table 4: Control srting definition

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).

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

Table 5: Control string conversion table old fromat to new format

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.

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.

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.

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.

Example:

```
integer fam_id, status


mf_open( 'test', '.', 'AwPs', fam_id, status )
if(status.ne.0)
mf_print_err( 'mf_open in myprog', status )
endif
```

5.41 **mf_open_srec()**

Create an empty state record format descriptor.

Declaration:

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

C:
int status
mf_open_srec( int fam_id, int mesh_id, int* srec_id )
```

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()`.

5.42 **mf_partition_state_data()**

This function is not recommended. Force closure of the current state data file to create a partition in the state data file sequence.

Declaration:

```
FORTRAN:
mf_partition_state_data( fam_id, status )
C:
int status
mf_partition_state_data( fam_id )
```

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.

5.43 **mf_print_error()**

Print a diagnostic associated with a Mili subroutine return status.

Declaration:

```
FORTRAN:
mf_print_error( preamble, rval )

C:
void
mf_print_error( char* preamble, int status )
```

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. This function only prints an error. It is the codes responsibility to clean up any allocated member after an error has occurred.

5.44 **mf_query_family()**

Request information about a Mili database family.

Declaration:

```
FORTRAN:
mf_query_family( fam_id, req_type, num_args, char_arg,
                      data, status )

C:
int status
mf_query_family( int fam_id, int req_type, int num_args,
                 char* char_arg, void* data )
```

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_mesh_name	Name of mesh with specified mesh id (CHARACTER*(**m_max_name_len**))
num_args	num_args(1) = id of mesh (INTEGER)
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_facets_in_surface	Quantity of facets in a surface (INTEGER)
num_args	num_args(1) = id of mesh (INTEGER)
char_args	**m_blank**
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_size	**Number of subrecords in the specified state 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_args	**m_blank**
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**
m_time_of_state	**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**

[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.

5.45 **mf_read_results()**

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

Declaration:

```
FORTRAN:
mf_read_results( fam_id, state, subrec_id, qty, results,
                 data, status )
C:
int status
mc_read_results( int fam_id, int state, int subrec_id,
                 int qty, int results, void* data )
```

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.

5.46 **mf_read_scalar()**

Read the named scalar parameter from the referenced family.

Declaration:

```
FORTRAN:
mf_read_scalar( fam_id, name, value, status )
C:
int status
mc_read_scalar( int fam_id, int name, void* value )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of scalar parameter to read.
`value`	Output	any	A reference to the properly typed variable into which the parameter will be read. In C this would be a void*.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

5.47 **mf_read_string()**

Read the named string parameter from the referenced family.

Declaration:

```
FORTRAN:
mf_read_string( fam_id, name, string, status)
C:
int status
mf_read_string( int fam_id, char* name, char* string )
```

Arguments: :

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of scalar parameter to read.
`string`	Output	CHARACTER*n ($n \leq$ **`m_max_string_len`**)	A character array into which the string parameter will be read.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

5.48 **mf_restart_at_file()**

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

Declaration:

```
FORTRAN:
mf_restart_at_file( fam_id, file_sequence, status )
C:
int status
mf_restart_at_file( int fam_id, int file_sequence )
```

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.

5.49 **mf_restart_at_state()**

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

Declaration:

```
FORTRAN:
mf_restart_at_state( fam_id, file_sequence,
                     file_state_index, status )
C:
int status
mf_restart_at_state( int fam_id, int file_sequence,
                     int file_state_index )
```

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. If set to a -1 then the restart will begin at the global state number,
`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. If file_state_index = -1 then will restart at the global state number.
`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.

5.50 **mf_set_buffer_qty()**

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

Declaration:

```
FORTRAN:
mf_set_buffer_qty( fam_id, mesh_id, class_name,
                   buffer_qty, status )
C:
int status
mf_set_buffer_qty( int fam_id, int mesh_id,
                   char* class_name, int buffer_qty )
```

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.

5.51 **mf_suffix_width()**

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

Declaration:

```
FORTRAN:
mf_suffix_width( fam_id, width, status )
C:
int status
mf_suffix_width( int fam_id, int width )
```

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.

5.52 **mf_wrt_array()**

Write a named array to a Mili family.

Declaration:

```
FORTRAN:
mf_wrt_array( fam_id, type, name, order, dimensions, data,
              status )
C:
int status
mf_wrt_array( int fam_id, int type, char* name, int order,
              int* dimensions, void* data )
```

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.

5.53 **mf_wrt_scalar()**

Write a named scalar value to a Mili family.

Declaration:

```
FORTRAN:
mf_wrt_scalar( fam_id, type, name, value, status )
C:
int status
mf_wrt_scalar( int fam_id, int type,char*  name,
               void* value )
```

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 ($n \leq$ **`m_max_name_len`**)	Name of the scalar.
`value`	Input	(dependent on `type` above)	Value of the scalar.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

5.54 **mf_wrt_stream()**

Write state data as a stream of values.

Declaration:

```
FORTRAN:
mf_wrt_stream( fam_id, type, quantity, data, status )
C:
int status
mc_wrt_stream( int fam_id, int type, int quantity, void* data )
```

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.

5.55 **mf_wrt_string()**

Write a named string to a Mili family.

Declaration:

```
FORTRAN:
mf_wrt_string( fam_id, name, value, status )
C:
int status
mf_wrt_string( int fam_id, char* name, char* value )
```

Arguments:

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

5.56 **mf_wrt_subrec()**

Write part of a subrecords state data.

Declaration:

```
FORTRAN:
mf_wrt_subrec( fam_id, name, start, stop, data, status )
C:
int status
mf_wrt_subrec( int fam_id, char* name, int start, int stop,
               void* data )
```

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()`.

6 MILI FORTRAN and C TI API

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

6.1 mf_ti_check_arch() *Obsolete*

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.

Declaration:

```
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

6.2 mf_ti_def_class()

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

Declaration:

```
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

6.3 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.

Declaration:

```
mf_ti_disable( fam_id )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier--unused.

Notes:

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

Example:

```
mc_ti_disable()
```

See also:

`mf_ti_enable()`

C API: `mc_ti_disable()`

File ti.h

6.4 mf_ti_enable()

Declaration:

```
mf_ti_enable( fam_id )
```

Arguments: :

`fam_id`	Input	INTEGER	Mili family identifier--unused.

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()`

C API: `mc_ti_enable()`

File ti.h

6.5 mf_ti_get_metadata

Reads metadata fields for a ti file family.

Declaration:

```
mf_ti_get_metadata( fam_id, mili_version, host, arch,
                    timestamp, username, xmilics_version,
                    code_name, rval )
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`mili_version`	Ouput	CHARACTER*n ($n \leq$ **m_max_string_len**)	Returns the version of Mili which wrote the database.
`host`	Output	CHARACTER*n ($n \leq$ **m_max_string_len**)	Returns the name of the host from which database was written.
`arch`	Output	CHARACTER*n ($n \leq$ **m_max_string_len**)	Returns the architecture of the host from which database was wriiten.
`timestamp`	Output	CHARACTER*n ($n \leq$ **m_max_string_len**)	Returns the creation time and date of the database.
`username`	Output	CHARACTER*n ($n \leq$ **m_max_string_len**)	Returns the name of the user who generated the database.
`xmilics_version`	Output	CHARACTER*n ($n \leq$ **m_max_string_len**)	Returns the version of xmilics generating the database (if any).

`code_name`	Output	CHARACTER*n ($n \leq$ **`m_max_string_len`**)	Returns the name of the code generating the database (if any).
`rval`	Output	INTEGER	Returns 0 on success, else non-zero error code.

See also:

C API: `mc_ti_get_metadata()`

File: ti.h

6.6 mf_ti_make_var_name()

Returns a TI variable name with embedded class info.

Declaration:

```
mf_ti_make_var_name(fam_id, name, class, new_name, status)
```

Arguments:

`fam_id`	Input	INTEGER	Mili family identifier.
`name`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name without embedded class info.
`class`	Input	CHARACTER*n ($n \leq$ **`m_max_name_len`**)	Name of class variable is associated with.
new_name	Output	CHARACTER*n ($n \leq$ **m_max_name_len**)	Returns the TI variable name with embedded class info.
status	Output	INTEGER	Returns 0 on success, else non-zero error code.

See also:

C API: `mc_ti_make_var_name()`

File: ti.h

6.7 mf_ti_read_array()

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

Declaration:

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

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	void **	Pointer to the array read.
`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.
`num_items_read`	Output	INTEGER	Returns the number of array items read.

See also:

`mf_ti_wrt_array()`

`mf_ti_def_class()`

C API: `mc_ti_read_array()`

File: ti.h

6.8 mf_ti_read_scalar()

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

Declaration:

```
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

6.9 mf_ti_read_string()

Reads a string variable from the TI data file.

Declaration:

```
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 ($n \leq$ **`m_max_string_len)`**	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

6.10 mf_ti_set_class()

Defines class information for a new TI class.

Declaration:

```
mf_ti_set_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 ($n \leq$ **`m_max_name_len`**)	Superclass of data item.
`meshvar`	Input	INTEGER	If TRUE then this data items is a mesh variable and of mesh length– either nodal or an element field.
`nodal`	Input	INTEGER	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 ($n \leq$ **`m_max_name_len`**)	Short class name.
`long_name`	Input	CHARACTER*n	Long class name.

($n \leq$ **m_max_name_len**)

`status`	Output	INTEGER	Returns 0 on success, else non-zero error code.

Notes:

This function is also named `mc_ti_def_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_set_class()`

File: ti.h

6.11 mf_ti_undef_class()

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

Declaration:

```
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

6.12 mf_ti_wrt_array()

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

Declaration:

```
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

6.13 mf_ti_wrt_scalar()

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

Declaration:

```
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

6.14 mf_ti_wrt_string()

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

Declaration:

```
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 ($n \leq$ **`m_max_string_len`**)	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: Configuration Options

Usage: configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE. See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
```
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version           display version information and exit
  -q, --quiet, --silent   do not print `checking...' messages
      --cache-file=FILE   cache test results in FILE [disabled]
  -C, --config-cache      alias for `--cache-file=config.cache'
  -n, --no-create         do not create output files
      --srcdir=DIR        find the sources in DIR [configure dir or `..']
```

Installation directories:
```
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [/usr/local]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                          [PREFIX]
```

By default, `make install' will install all the files in
`/usr/local/bin', `/usr/local/lib' etc. You can specify
an installation prefix other than `/usr/local' using `--prefix',
for instance `--prefix=$HOME'.

For better control, use the options below.

Fine tuning of the installation directories:
```
  --bindir=DIR            user executables [EPREFIX/bin]
  --sbindir=DIR           system admin executables [EPREFIX/sbin]
  --libexecdir=DIR        program executables [EPREFIX/libexec]
  --sysconfdir=DIR        read-only single-machine data [PREFIX/etc]
  --sharedstatedir=DIR    modifiable architecture-independent data [PREFIX/com]
  --localstatedir=DIR     modifiable single-machine data [PREFIX/var]
  --libdir=DIR            object code libraries [EPREFIX/lib]
  --includedir=DIR        C header files [PREFIX/include]
  --oldincludedir=DIR     C header files for non-gcc [/usr/include]
  --datarootdir=DIR       read-only arch.-independent data root [PREFIX/share]
  --datadir=DIR           read-only architecture-independent data [DATAROOTDIR]
  --infodir=DIR           info documentation [DATAROOTDIR/info]
  --localedir=DIR         locale-dependent data [DATAROOTDIR/locale]
```

```
  --mandir=DIR           man documentation [DATAROOTDIR/man]
  --docdir=DIR           documentation root [DATAROOTDIR/doc/Mili]
  --htmldir=DIR          html documentation [DOCDIR]
  --dvidir=DIR           dvi documentation [DOCDIR]
  --pdfdir=DIR           pdf documentation [DOCDIR]
  --psdir=DIR            ps documentation [DOCDIR]

System types:
  --build=BUILD     configure for building on BUILD [guessed]
  --host=HOST       cross-compile to build programs to run on HOST [BUILD]

Optional Features:
  --disable-option-checking  ignore unrecognized --enable/--with options
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --enable-sl            Compile and load with Stresslink
  --enable-silo          Compile and load with SILO
  --enable-silo-debug     Compile and load with debug version of SILO
  --enable-siloh5        Compile and load with Silo HDF5
  --enable-hdf           Compile and load with HDF
  --enable-bits64        Compile and load with 64bit option
  --enable-kw            Compile and load with KlocWorks
  --enable-shared        Compile as a shared library

Optional Packages:
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --with-silo=PATH        Use given base PATH for SILO libraries and header
                          files.
  --with-hdf=PATH         Use given base PATH for HDF libraries and header
                          files.
  --with-kw=PATH          Use given base PATH for KW trace file
  --install-path=PATH     Use given base PATH installing Mili
  --with-usrbuild=PATH    Use given PATH for BUILD DIRECTORY

Some influential environment variables:
  F77         Fortran 77 compiler command
  FFLAGS      Fortran 77 compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  LIBS        libraries to pass to the linker, e.g. -l<library>
  CC          C compiler command
  CFLAGS      C compiler flags
  CPPFLAGS    C/C++/Objective C preprocessor flags, e.g. -I<include dir> if
              you have headers in a nonstandard directory <include dir>
  CPP         C preprocessor
```

Use these variables to override the choices made by `configure' or to help
it to find libraries and programs with nonstandard names/locations.

Report bugs to <durrenberger1@llnl.gov>.

Appendix B: Build Options

Out from running gmake with no options:

```
        ***************************
         *                 *
         *    Mili Makefile      *
         *                 *
         ***************************
            Available Targets
            -----------------
     all:  Build the libraries.
     opt:  Build optimized Libraries.
   debug:  Build debug Libraries.

   Utilities:
   utilsdebug: Build debug utilities such as md.
   utilsopt:   Build opt   utilities such as md.

   clean:      Remove object, core and library files for this build.
   distclean:  Restore build environment back to distribution state.
   clobber:    Remove everything in lib_ directories.

   installdirs: Create the public installation directories.
   install:     Put new code and documentation in public.

   install-initversion: Create public directories for new Mili Version.
   install-chmod: Change file permissions in public.
   install-setversion: Define version link (miliversion) in public.

   uninstall:   Remove code from public.

            Available Options
            -----------------
 TASKS=  Number of processes for parallel builds.
 VERBOSE=  Verbose mode for echoing of build steps.
 fortran={true|false}  Choose whether to build the fortran interface
                 or not. The default is true.
```

```
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, truss_long /'stress','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', 'Material',
      +    't', 'Truss' /

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)
      + 5,6,7,8,9,10,11,12,2,2/ ! Hex2 (Part 2, Mat 2)

c.....Trusses 1:4
c..... 2 nodes per truss Part#=1 / Material#=3
c
       integer truss_1_4(10, 4)
       data truss_1_4/
      + 11,10,2,1, ! Truss1 (Part 1, Mat 2)
      + 10,13,2,1, ! Truss2 (Part 1, Mat 2)
      + 13,14,2,1, ! Truss3 (Part 1, Mat 2)
      + 14,15,2,1/ ! Truss4 (Part 1, Mat 2)
c     State data to written at each time step. We of course would
c     build these up with our data prior to writing.

       real state_record1(149)
       data state_record1/
      + 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, ! end of nodes positions
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, !end of node velocities
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
c       end of node accelerations
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0/       ! end of "Stress" Subrecord

       real state_record2(149)
       data state_record2/
      + 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, ! end of nodes positions
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, !end of node velocities
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
c       end of node accelerations
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0/       ! end of "Stress" Subrecord

       real state_record3(149)
       data state_record3/
      + 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, ! end of nodes positions
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, !end of node velocities
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0. 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
c     end of node accelerations
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
      + 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0/       ! end of "Stress" Subrecord

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 )

      if(stat.ne.0)
          call mf_print_error( 'mf_def_vec_svar', stat )
      endif

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

      if(stat.ne.0)
          call mf_print_error( 'mf_def_vec_svar', stat )
      endif

      call mf_def_vec_svar( fid, m_real, 3, nodal_short(3),
     +                     nodal_long(3), nodal_short(10),
     +                     nodal_long(10), stat )

      if(stat.ne.0)
          call mf_print_error( 'mf_def_vec_svar', stat )
      endif

      call mf_def_vec_svar( fid, m_real, 6, hex_short(1),
     +                     hex_long(1), hex_short(3),
     +                     hex_long(3), stat )

      if(stat.ne.0)
          call mf_print_error( 'mf_def_vec_svar', stat )
      endif

      call mf_def_svars( fid, 1, svar_types, hex_short(2),
     +                   hex_long(2), stat )

      if(stat.ne.0)
          call mf_print_error( 'mf_def_svars', stat )
      endif
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 )

      if(stat.ne.0)
          call mf_print_error( 'mf_make_umesh', stat )
      endif

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 )
```

```
      if(stat.ne.0)
          call mf_print_error( 'mf_def_class (global)', stat )
      endif

      call mf_def_class_idents( fid, mesh_id, global_s, 1, 1, stat )

      if(stat.ne.0)
          call mf_print_error( 'mf_def_class_idents (global)', stat )
      endif

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

      if(stat.ne.0)
          call mf_print_error( 'mf_def_class (material)', stat )
      endif

      call mf_def_class_idents( fid, mesh_id, material_s, 1, 3, stat )

      if(stat.ne.0)
          call mf_print_error( 'mf_def_class_idents (material)', stat )
      endif

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 )
      if(stat.ne.0)
          call mf_print_error( 'mf_def_class (nodal)', stat )
      endif

      call mf_def_nodes( fid, mesh_id, nodal_s, 1, 16, node_1_16, stat )

      if(stat.ne.0)
          call mf_print_error( 'mf_def_nodes', stat )
      endif

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 )

       if(stat.ne.0)
           call mf_print_error( 'mf_def_class (hexs)', stat )
       endif

       call mf_def_conn_seq( fid, mesh_id, hexs_s, 1, 2,
      +                      hex_1_2, stat )

       if(stat.ne.0)
           call mf_print_error( 'mf_def_conn_seq', stat )
       endif

c.....Define a truss class
       call mf_def_class( fid, mesh_id, m_truss, truss_s, truss_l, stat )

       if(stat.ne.0)
           call mf_print_error( 'mf_def_class (truss)', stat )
       endif

       call mf_def_conn_seq( fid, mesh_id, truss_s, 1, 4,
      +                      truss_1_4, stat )

       if(stat.ne.0)
           call mf_print_error( 'mf_def_conn_seq', stat )
       endif

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 )

       if(stat.ne.0)
           call mf_print_error( 'mf_open_srec', stat )
       endif

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 )

       if(stat.ne.0)
           call mf_print_error( 'mf_def_subrec', stat )
       endif
```

```
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 )

       if(stat.ne.0)
           call mf_print_error( 'mf_def_subrec', stat )
       endif

       call mf_close_srec( fid, srec_id, stat )

       if(stat.ne.0)
           call mf_print_error( 'mf_close_srec', stat )
       endif

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 )

       if(stat.ne.0)
           call mf_print_error( 'mf_def_subrec', stat )
       endif

       call mf_close_srec( fid, srec_id, stat )

       if(stat.ne.0)
           call mf_print_error( 'mf_close_srec', stat )
       endif

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 )
       if(stat.ne.0)
           call mf_print_error( 'mf_commit', stat )
       endif

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 )
```

```
if(stat.ne.0)
    call mf_print_error( 'mf_new_state', stat )
endif

call mf_wrt_stream( fid, m_real, 149, state_record1, stat )

if(stat.ne.0)
    call mf_print_error( 'mf_wrt_stream', stat )
endif

call mf_end_state(fid, srec_id, stat)

if(stat.ne.0)
   call mf_print_error( 'mf_end_state', stat )
endif

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

if(stat.ne.0)
    call mf_print_error( 'mf_new_state', stat )
endif

call mf_wrt_stream( fid, m_real, 149, state_record2, stat )

if(stat.ne.0)
    call mf_print_error( 'mf_wrt_stream', stat )
endif

call mf_end_state(fid, srec_id, stat)

if(stat.ne.0)
   call mf_print_error( 'mf_end_state', stat )
endif

time = 2.0
call mf_new_state( fid, srec_id, time, suffix, st_idx, stat )

if(stat.ne.0)
    call mf_print_error( 'mf_new_state', stat )
endif

call mf_wrt_stream( fid, m_real, 149, state_record3, stat )

if(stat.ne.0)
    call mf_print_error( 'mf_wrt_stream', stat )
endif

call mf_end_state(fid, srec_id, stat)
```

```
      if(stat.ne.0)
         call mf_print_error( 'mf_end_state', stat )
      endif


c.....MUST close the Mili family when finished
      call mf_close( fid, stat )
      if(stat.ne.0)
         call mf_print_error( 'mf_close', stat )
      endif
```