Programmer's Guide

Altair HyperMesh 3.1

For technical support, contact us by phone or e-mail:

Country	Phone	E-mail
United States	248.614.2400	hmsupport@altair.com
Germany	49.7031.6208.22	support@altair-gmbh.de
India	91.80.665.8871	support@india.altair.soft.net
Israel	972.3.5473651	support@netvision.net.il
Italy	39.11.900.77.11	support@altairtorino.it
Japan	81.3.5396.1341	aj-support@altair.com
Korea	822.573.4152	support@yewon.co.kr
Scandinavia	46.46.286.2052	support.sweden@altair.com
United Kingdom	44.1327.810700	support@uk.altair.com

Trademark Acknowledgments:

HyperMesh is a registered trademark of Altair Engineering, Inc.
ACIS is a registered trademark of SPATIAL TECHNOLOGY, INC.
ACIS Geometric Modeler is a registered trademark of SPATIAL TECHNOLOGY, INC.
ACIS Kernel is the registered trademark of SPATIAL TECHNOLOGY, INC.
ACIS Parametric Surfaces is the registered trademark of SPATIAL TECHNOLOGY, INC.
MS-DOS is a registered trademark of Microsoft Corporation.
UNIX is a registered trademark of AT&T.
MSC/NASTRAN is a registered trademark of MSC.
ABAQUS is a registered trademark of Hibbitt, Karlsson, & Sorensen, Inc.
ANSYS is a registered trademark of Ansys, Inc.
PATRAN is a registered trademark of MSC.
LS-DYNA is a registered trademark of LSTC.
MARC is a registered trademark of MARC Analysis Research Corporation.
PAMCRASH is a registered trademark of Engineering Systems International.
FLUENT is a registered trademark of Fluent, Incorporated.
I-DEAS is a registered trademark of Structural Dynamics Corporation.
Spaceball is a registered trademark of Spacetec IMC Corporation.

Using the HyperMesh C Libraries

This Guide assumes familiarity with the C programming language as well as basic programming concepts. Select a category listed below for suggestions on how to create a C program that uses the HyperMesh libraries.

Compilers

The particular C compiler that you use varies from platform to platform. The compilers for each of the platforms on which HyperMesh runs may vary slightly and some are invoked differently depending on the system you are running. The following is a list of the HyperMesh platforms and the name of the ANSI C compilers used to compile HyperMesh:

DEC/ALPHA	c89
HP	c89
PC	Visual C++ 4.2
SGI	cc
SUN	cc

Modify your `makefile` to call the corresponding compilers.

Header Files

Header files are included in source code files to define structures, variables, and prototypes. In order for a program to be compiled using the HyperMesh libraries, the program must use the header files provided with HyperMesh. The following header files are included with HyperMesh:

hmlib.h	Include this file if any functions prefixed with HM_ are used. In addition, include this file if `hmreslib.h`, `hmmodlib.h`, or `hminlib.h` are included.
hmreslib.h	Include this file if any functions prefixed with `HMRES_` are used. In addition, include this file if `hmmodlib.h` is included.
hmmodlib.h	Include this file if any functions prefixed with `HMMOD_` are used.
hminlib.h	Include this file if any functions prefixed with `HMIN_` are used.

The order of the `include` file statements is very important. If you are writing a results translator that does not use `hmmodlib`, use the following order:

```
#include "hmlib.h"
#include "hmreslib.h"
```

If you are writing a results translator that does use `hmmodlib`, use the following order:

```
#include "hmlib.h"
#include "hmreslib.h"
#include "hmmodlib.h"
```

If you are writing an input translator, use the following order:

```
#include "hmlib.h"
#include "hminlib.h"
```

Libraries

The libraries need to be included as follows and in the order shown if you are writing a results translator:

```
hmmodlib.a hmreslib.a hmlib.a
```

Note that `hmmodlib.a` is optional if you are writing a results translator that does not use `hmmodlib`. If you are writing an input translator, include the libraries as follows:

```
hminlib.a hmlib.a
```

Make

`make` is a powerful utility which allows programs to be broken down into small pieces, and based on the date of modification, the pieces not up-to-date are recompiled and/or linked. A basic `makefile` is shown below.

```
<program> : <objectfiles> <libraries>

<tab>cc -o <program> <objectfiles> <libraries> -lm

.c.o :
<tab>cc -c $*.c
```

where:

`<program>` is the name of the program being compiled.

`<objectfiles>` are the object files needed to create the executable.

`<libraries>` are the HyperMesh libraries needed by the object files.

In this example, the program being compiled is called `mytrans`. The object file needed is `mytrans.o` and the two libraries are `hminlib.a` and `hmlib.a`. The object files are created by compiling the source code files, `mytrans.c`, which are not explicitly listed in the `makefile`. After the substitutions are made, the `makefile` needed to create the program `mytrans` looks like this:

```
mytrans : mytrans.o hminlib.a hmlib.a
<tab>cc -o mytrans mytrans.o hminlib.a hmlib.a -lm
```

```
.c.o :
<tab>cc -c $*.c
```

Note that <tab> is the tab character.

Introduction to hmlib

hmlib is a library which contains a set of routines to aid you in writing a C program. hmlib contains functions that either complement existing programs or make it easier to write programs which work with HyperMesh. test

The functions contained in hmlib are divided into six groups. The functions in these groups:

1. Allow you to read data from ASCII files.
2. Give the capability of working with dynamically allocated arrays without worrying about the low-level details of dynamic memory allocation.
3. Allow for direct memory allocation and also enhance the standard C functions with a level of error checking to prevent common errors from occurring.
4. Handle string operations.
5. Allow for vector operations to be performed.
6. Allow you to terminate programs, send messages, and read and write to binary files.

ASCII File Functions

The ASCII file functions in hmlib enable you to read data from ASCII files.

HM_asciifile_readeol()

HM_asciifile_readfixeddouble()

HM_asciifile_readfixedint()

HM_asciifile_readfixedstring()

HM_asciifile_readeol()

Reads an end of line from an ASCII file.

Syntax void HM_asciifile_readeol(FILE * file);

file The handle to the open ASCII file (must be opened with fopen(“”,”rt”)).

Returns Nothing.

HM_asciifile_readfixeddouble()

Reads a fixed double from an ASCII file.

Syntax double HM_asciifile_readfixeddouble(FILE * file, int width);

file The handle to the open ASCII file (must be opened with fopen(“”,”rt”)).

width The width of the real number that should be read.

Returns The value of the double contained in the field.

HM_asciifile_readfixedint()

Reads a fixed integer from an ASCII file.

Syntax int HM_asciifile_readfixedint(FILE * file, int width);

file The handle to the open ASCII file (must be opened with fopen(“”,”rt”)).

width The width of the integer that should be read.

Returns The value of the integer contained in the field.

HM_asciifile_readfixedstring()

Reads a fixed string from an ASCII file and stores it in the location pointed to by the user-passed character pointer string.

Syntax char * HM_asciifile_readfixedstring(FILE * file, char * string, int width);

file The handle to the open ASCII file (must be opened with fopen(“”,”rt”)).

string The pointer to a string. It is assumed that the memory

allocated for the string is greater than or equal to the width.

width The width of the string that should be read.

Returns The pointer to the value of the string contained in the field.

Dynamic Block Functions

The dynamic block functions provide an easy-to-use interface that allows you to allocate and free dynamically allocated blocks of memory, without worrying about low-level programming issues. In this case, a dynamic block is an entity that, once created, is capable of storing items of a user-defined size. As more items are added to the block, the block automatically acquires more memory to store those elements. After items are stored, the dynamic block interface also allows you to retrieve pointers into the block to access the previously stored items.

Before a dynamic block can be used, it must be created. To create a dynamic block, you must call `HM_dynamicblockallocate()`. This function requires you to pass the type of block being created and the size of the items stored. The function returns a void pointer that points to the dynamic block. You must store the pointer returned from the allocate function, as it is used for all successive calls to the dynamic block routines.

Once the block is created, the next step is to store items in the block. This is accomplished by making a call to the function `HM_dynamicblockadd()`. Call this function for each item to be added and returns a void pointer that points to the memory allocated for that item. This pointer is then used directly, and the appropriate information is placed into the block.

Information stored in a block may be accessed by using the function `HM_dynamicblockgetpointer()` which returns a pointer to any previously stored item. With the use of the pointer returned, information stored in the item can be accessed or modified.

The final important point when using blocks is that once a block is created, it should be destroyed. This is accomplished by calling `HM_dynamicblockfree()`. However, once this function is called, the block is no longer available, and the memory allocated for the block is freed.

`HM_dynamicblockadd()`

`HM_dynamicblockallocate()`

`HM_dynamicblockfind()`

`HM_dynamicblockfree()`

`HM_dynamicblockgetpointer()`

`HM_dynamicblockgetsize()`

`HM_dynamicblockreset()`

`HM_dynamicblocksort()`

Dynamic Block Functions Example Program

HM_dynamicblockadd()

Adds an item to a dynamic block.

Syntax void * HM_dynamicblockadd(void * blockptr);

blockptr A pointer to a dynamic block. This pointer must be allocated with `HM_dynamicblockallocate()`.

Returns A pointer to an item allocated in the dynamic block. The amount of memory allocated for the item is specified in the call `HM_dynamicblockallocate()`.

Comments If unsuccessful, the function calls `HM_terminate()` with the appropriate message, and program execution is terminated.

HM_dynamicblockallocate()

Creates a dynamic block.

Syntax void * HM_dynamicblockallocate(int type, unsigned int size);

Type The type of the dynamic block being created. The type assigned to a dynamic block determines the effectiveness of the block in its ability to store information; all blocks may still store any number of items. Selection of the type is based on the anticipated number of items to be stored in the block. The type is selected from these parameters:

1. HM_DYNAMICBLOCKTYPE_SMALL, when fewer than 100 items are anticipated.
2. HM_DYNAMICBLOCKTYPE_MEDIUM, when fewer than 1000 items are anticipated.
3. HM_DYNAMICBLOCKTYPE_LARGE, when fewer than 10,000 items are anticipated.
4. HM_DYNAMICBLOCKTYPE_HUGE, for all others.

size The size of each item in the dynamic block.

Returns A pointer to a dynamic block.

Comments If unsuccessful, the function calls `HM_terminate` with the appropriate message, and program execution is terminated.

HM_dynamicblockfind()

Finds an item in a dynamic block.

Syntax void * HM_dynamicblockfind(void * key, void * blockptr, int (*compare)(void const *item1, void const *item2));

key The item for which the function is looking.

blockptr A pointer to a dynamic block returned by `HM_dynamicblockallocate()`.

(*compare)(void const *item1, void const *item2)

A function that returns:

-1 If item1 < item2.

1 If item1 > item2.

0 If item1 = item2.

Returns A pointer to the item, if it is found. If the item is not found, the function returns NULL.

Comments **key*, **item1*, and **item2* are all pointers to the same structure. The size of this structure should be passed into `HM_dynamicblockallocate()`. `HM_dynamicblocksort()` must be called with the same compare function before `HM_dynamicblockfind()`.

HM_dynamicblockfree()

Frees a dynamic block.

Syntax void HM_dynamicblockfree(void * blockptr);

blockptr A pointer to a dynamic block allocated by `HM_dynamicblockallocate()`.

Returns Nothing.

Comments After this function is called, the pointer to the dynamic block is no longer valid.

HM_dynamicblockgetpointer()

Gets a pointer to an item in a dynamic block.

Syntax void * HM_dynamicblockgetpointer(void * blockptr, int index);

blockptr A pointer to a dynamic block allocated by `HM_dynamicblockallocate()`.

index The index of the item desired. This may be set to zero through one fewer than the total number of items in the block.

Returns A pointer to the item, if the index is within range. Otherwise, the function returns NULL.

HM_dynamicblockgetsize()

Reports the number of items in a block.

Syntax int HM_dynamicblockgetsize(void * blockptr);

blockptr A pointer to a dynamic block allocated by `HM_dynamicblockallocate()`.

Returns The number of items in a block.

HM_dynamicblockreset()

Resets a dynamic block.

Syntax void HM_dynamicblockreset(void * blockptr);

blockptr A pointer to a dynamic block allocated by `HM_dynamicblockallocate()`.

Returns Nothing.

Comments This function clears the items in a dynamic block.

HM_dynamicblocksort()

Sorts the items in a dynamic block.

Syntax

void HM_dynamicblocksort(void * blockptr, int (*compare)(void const *item1, void const *item2));

blockptr A pointer to a dynamic block allocated by `HM_dynamicblockallocate()`.

(*compare)(void const *item1, void const *item2)

A function that returns:

-1 If item1 < item2.

1 If item1 > item2.

0 If item1 = item2.

Returns

Nothing.

Comments

key, *item1*, and *item2* are all pointers to the same structure. The size of this structure should be passed into `HM_dynamicblockallocate()`.

Dynamic Block Functions Example Program

```
#include <stdio.h>

#include <stdlib.h>

#include “hmlib.h”


 /*

 This program demonstrates how to use the dynamic block functions.  In
 this example, the program stores, sorts, and retrieves a data structure
 containing information about a collector.

 */


 /* The structure for the component that contains the ID, name, and
 color. */

 typedef struct

 {

   HM_entityidtype id;

   char name[9];

   int color;

 } componentrecord, *componentpointer;

 /* The sort function for the block; sorts on the ID of the component */
 int sortfunction(void const *item1, void const *item2)

 {

   if (((componentpointer) item1)->id < ((componentpointer) item2)->id)
        return(-1);

   if (((componentpointer) item1)->id > ((componentpointer) item2)->id)
        return(1);

   return(0);

 }


 void main(void)

 {

   void *componentblock;
```

```
componentrecord componentrec;
componentpointer componentptr;
int i;

/* Create the dynamic block */
componentblock =
HM_dynamicblockallocate(HM_DYNAMICBLOCKTYPE_SMALL,
sizeof(componentrecord));
/* Store items in the block */ componentptr =
   HM_dynamicblockadd(componentblock);
componentptr->id = 3;
HM_stracpy(componentptr->name,”comp3”,9);
componentptr->color = 3;

componentptr = HM_dynamicblockadd(componentblock);
componentptr->id = 1;
HM_stracpy(componentptr->name,”comp1”,9);
componentptr->color = 1;

componentptr = HM_dynamicblockadd(componentblock);
componentptr->id = 2;
HM_stracpy(componentptr->name,”comp2”,9);
componentptr->color = 2;

/* Retrieve and print items */
printf(“Items stored:\n”);
i = 0;
while ((componentptr
HM_dynamicblockgetpointer(componentblock,i++))
  != NULL)
{
  printf(“component id = %d, name = ‘%s’, color =
    %d\n”,componentptr->id,componentptr->name,componentptr->color);
```

```
}
printf("\n");

/* Sort the items in the block */
HM_dynamicblocksort(componentblock,sortfunction);

/* Retrieve and print items.  Note the order changes. */
printf("Sorted Items:\n");
i = 0;
while ((componentptr = HM_dynamicblockgetpointer(componentblock,i++))
  != NULL)
{
  printf("component id = %d, name = '%s', color =
    %d\n",componentptr->id,componentptr->name,componentptr->color);
}
printf("\n");

/* Find item 2 in the block */
componentrec.id = 2;
componentptr =
HM_dynamicblockfind(&componentrec,componentblock,sortfunction);
if (componentptr) printf("found item 2 whose name is
  '%s'\n",componentptr->name);

/* Print the number of items in the block */
printf("The number of items stored is:
  %d\n",HM_dynamicblockgetsize(componentblock));

/* Free the dynamic block */
HM_dynamicblockfree(componentblock);
```

Memory Allocation Functions

The memory utilities contained within `hmlib` are designed to make it easier for you to allocate and free blocks of memory. While memory allocation and freeing can be performed in standard C without using the routines in `hmlib`, using the memory routines eliminates the need for you to perform error checking, and also allows you to access improved functionality in the future. In addition to using the memory routines inside of `hmlib`, you may prefer to use the dynamic blocks that are also found in `hmlib`.

`HM_calloc()`

`HM_free()`

`HM_malloc()`

`HM_realloc()`

Memory Allocation Functions Example Program

HM_calloc()

Allocates a block of memory and clears it to zero.

Syntax void * HM_calloc(int n, int size);

n The number of items to be stored in the block.

size The size of the items.

Returns A pointer to a block of memory.

Comments If successful, `HM_calloc()` returns a pointer to the block of memory. If unsuccessful, `HM_calloc()` calls `HM_terminate()` with an appropriate error message, and program execution is terminated.

HM_free()

Frees a block of memory.

Syntax void HM_free(void * ptr);

ptr A pointer to a block of memory.

Returns Nothing.

Comments If **ptr* is NULL, `HM_free()` does not perform any operation.

HM_malloc()

Allocates a block of memory.

Syntax void * HM_malloc(int size);

size The size of the wanted block in bytes.

Returns If successful, `HM_malloc()`returns a pointer to the block of memory. If unsuccessful, `HM_malloc()` calls `HM_terminate()` with an appropriate error message, and program execution is terminated.

HM_realloc()

Reallocates a block of memory.

Syntax void * HM_realloc(void * ptr, int size);

ptr The block of memory to be resized.

size The desired size of the block.

Returns If successful, `HM_realloc()` returns a pointer to the resized block of memory. If unsuccessful, `HM_realloc()` calls `HM_terminate()` with an appropriate error message, and program execution is terminated.

Comments If **ptr* is NULL, `HM_realloc()`behaves as `HM_malloc()`.

Memory Allocation Functions Example Program

```
#include <stdio.h>
#include <stdlib.h>
#include “hmlib.h”
 /*
   This example allocates and frees memory using the memory
   functions in hmlib.  If any of the memory allocation
   functions fail, the program terminates, which eliminates
   the need to check the validity of a pointer.
 */

 void main(void)
 {
   double *doubleptr;
   int *integerptr;

   /* Allocate memory to store 4 doubles */
   doubleptr = HM_malloc(4*sizeof(double));
   doubleptr[0] = 0.0;
   doubleptr[1] = 1.0;
   doubleptr[2] = 2.0;
   doubleptr[3] = 3.0;
   printf(“After malloc:\n”);
   printf(“%f %f %f %f\n”,
   doubleptr[0],doubleptr[1],doubleptr[2],doubleptr[3]);
   /* Reallocate the previously allocated memory to store
   63 doubles */
   doubleptr = HM_realloc(doubleptr,63*sizeof(double));
   printf(“After realloc:\n”);
   printf(“%f %f %f %f\n”,
   doubleptr[0],doubleptr[1],doubleptr[2],doubleptr[3]);
```

```
  /* Allocate memory for 12 integers */
  integerptr = HM_calloc(12,sizeof(int));

  /* Free all allocated memory */
  HM_free(doubleptr);
  HM_free(integerptr);

  /*
  Allocate a huge block of memory, which should fail unless
  you have a very large machine.
  */
  doubleptr = HM_malloc(0xffffff*sizeof(double));
  doubleptr[0] = 0.0;
  doubleptr[1] = 1.0;
  doubleptr[2] = 2.0;
  doubleptr[3] = 3.0;
  printf("After allocation of a huge block of memory:\n");
  printf("%f %f %f %f\n",
  doubleptr[0],doubleptr[1],doubleptr[2],doubleptr[3]);
}
```

String Functions

The string functions in `hmlib` were created to enhance the capabilities provided in the standard C library.

`HM_stracpy()`

`HM_strchkdouble()`

`HM_strchkint()`

`HM_strcnt()`

`HM_strdel()`

`HM_stricmp()`

`HM_strins()`

`HM_strinsc()`

`HM_strlwr()`

`HM_strnicmp()`

`HM_strrep()`

`HM_strrmeol()`

`HM_strrmsp()`

`HM_strupr()`

`HM_strxint()`

String Functions Example Program

HM_stracpy()

Copies a string of absolute length.

Syntax char * HM_stracpy(char * dest, char * source, int n);

dest	A pointer to the destination string.
source	A pointer to the source string.
n	The number of characters to copy.

Returns A pointer to the destination string.

Comments This function copies **n-1* characters from **dest* to **source*, and then NULL terminates **dest* at the **nth* position.

HM_strchkdouble()

Checks the characters in a string for those considered valid for a double field.

Syntax int HM_strchkdouble(char * string);

string A pointer to a string.

Returns Zero, if all of the characters in the string are ‘ ‘, ‘0’-‘9’, ‘+’, ‘-‘, ‘D’, ‘E’, ‘e’, or ‘.’; otherwise, nonzero.

HM_strchkint()

Checks the characters in a string for those considered valid for an integer field.

Syntax int HM_strchkint(char * string);

string A pointer to a string.

Returns Zero, if all of the characters in the string are ‘ ‘, ‘0’-‘9’, ‘+’, or ‘-‘; otherwise, nonzero.

HM_strcnt()

Counts the number of occurrences of a character in a string.

Syntax int HM_strcnt(char * string, char c);

string A pointer to a string.

c The character being counted.

Returns The number of occurrences of *c in **string*.

HM_strdel()

Deletes a string of characters from a string.

Syntax char * HM_strdel(char * string, int position, int length);

string A pointer to a string that is to have characters deleted.

position The position in the string where deletion should start.

length The number of characters that should be deleted.

Returns A pointer to the string.

HM_stricmp()

Performs a string compare, ignoring upper and lower case.

Syntax int HM_stricmp(const char *string1, const char *string2)

string1	A pointer to the first string you want to compare.
string2	A pointer to the second string you want to compare.

Returns
-1, if string1 < string2
1, if string1 > string2
0, if string1 = string2

HM_strins()

Inserts a string into another string.

Syntax char * HM_strins(char * string1, int position, char * string2);

string1	A pointer to the string that is to have string2 inserted.
position	The position in **string1* where **string2* should be inserted.
string2	A pointer to the string that is to be inserted into **string1*.

Returns A pointer to the string.

HM_strinsc()

Inserts a character into a string.

Syntax char * HM_strinsc(char * string, int position, char character);

string	A pointer to a string that is to have a character inserted.
position	The position in the string at which the character should be inserted.
character	The character to be inserted.

Returns A pointer to the string.

HM_strlwr()

Converts the characters in a string to lower case.

Syntax char * HM_strlwr(char * string);

string	A pointer to a string.

Returns A pointer to a string.

HM_strnicmp()

Performs a string compare, ignoring upper and lower case.

Syntax int HM_strnicmp(const char *string1, const char *string2, size_t n)

string1	A pointer to the first string you want to compare.
string2	A pointer to the second string you want to compare.
n	The number of characters in the strings to compare.

Returns
-1, if string1 < string2
1, if string1 > string2
0, if string1 = string2

HM_strrep()

Replaces a string within a string.

Syntax char * HM_strrep(char * string, char * replace, char * with);

string	A pointer to a string that is to have a substring replaced with another string.
replace	A pointer to a string that contains the string to be replaced.
with	A pointer to a string that contains the string that is to replace any occurrences of **replace*.

Returns A pointer to **string*.

HM_strrmeol()

Removes end of line spaces.

Syntax char * HM_strrmeol(char * string);

string A pointer to a string.

Returns A pointer to the string.

HM_strrmsp()

Removes leading and trailing spaces from a string.

Syntax char * HM_strrmsp(char * string);

string A pointer to a string.

Returns A pointer to the string.

HM_strupr()

Converts the characters in a string to upper case.

Syntax char * HM_strupr(char * string);

string A pointer to a string.

Returns A pointer to the string.

HM_strxint()

Extracts the integer value from a string.

Syntax int HM_strxint(char * string);

string A pointer to a string.

Returns The integer value found in the string.

Comments This function looks for the first occurrences of the characters “0” through “9” and then converts that character along with all other valid integer characters which follow it.

String Functions Example Program

```
#include <stdio.h>
#include <stdlib.h>
#include “hmlib.h”

 /*
   This example demonstrates the string functions found within hmlib.
 */

   void main(void)
 {
   char string1[81];
   char string2[13];

   HM_stracpy(string1,”This is a string.”,sizeof(string1));
   HM_stracpy(string2,string1,sizeof(string2));
   printf(“Output from HM_stracpy() example:\n”);
   printf(“string1 = ‘%s’\n”,string1);
   printf(“string2 = ‘%s’\n”,string2);
   printf(“\n”);

   HM_stracpy(string1,”THIS IS A STRING.”,sizeof(string1));
   HM_strlwr(string1);
   printf(“Output from HM_strlwr() example:\n”);
   printf(“string1 = ‘%s’\n”,string1);

   HM_stracpy(string1,”See Spot car run.”,sizeof(string1));
   HM_strdel(string1,9,4);
   printf(“Output from HM_strdel() example:\n”);
   printf(“string1 = ‘%s’\n”,string1);
```

```
HM_stracpy(string1,"This is a string.",sizeof(string1));
HM_strupr(string1);
printf("Output from HM_strupr() example:\n");
printf("string1 = '%s'\n",string1);

HM_stracpy(string1,"See pot run.",sizeof(string1));
HM_strinsc(string1,4,'S');
printf("Output from HM_strinsc() example:\n");
printf("string1 = '%s'\n",string1);

HM_stracpy(string1,"See run.",sizeof(string1));
HM_strins(string1,4,"Spot ");
printf("Output from HM_strins() example:\n");
printf("string1 = '%s'\n",string1);

printf("Output from HM_strchkint('123'): %d\n",HM_strchkint("123"));
printf("Output from HM_strchkint('abc'): %d\n",HM_strchkint("abc"));
printf("Output from HM_strchkdouble('123e-3'):
   %d\n",HM_strchkdouble("123e-3"));
printf("Output from HM_strchkdouble('abcdef'):
   %d\n",HM_strchkdouble("abcdef"));
HM_stracpy(string1,"See Jane Run.  Run Jane Run.",sizeof(string1));
HM_strrep(string1,"Jane","Spot");
printf("Output from HM_strrep() example:\n");
printf("string1 = '%s'\n",string1);

HM_stracpy(string1,"   See Spot Run.   ",sizeof(string1));
HM_strrmsp(string1);
printf("Output from HM_strrmsp() example:\n");
printf("string1 = '%s'\n",string1);

HM_stracpy(string1,"See Spo\nt Run.\n",sizeof(string1));
```

```
   HM_strrmeol(string1);
   printf("Output from HM_strrmeol() example:\n");
   printf("string1 = '%s'\n",string1);

   printf("Output from HM_strxint('a12c3'): %d\n",HM_strxint("a12c3"));
   printf("Output from HM_strcnt('A dog ran down the alley','a'):
   %d\n",HM_strcnt("A dog ran down the alley",'a'));
}
```

Vector Functions

The vector utilities provided by `hmlib` allow you to perform operations on vectors, i.e., cross product and dot product. For all of the vector functions described in the Help sections below, you must use the `vectorrecord` and `vectorpointer` structures.

NOTE These functions use vectors of the data type, `HM_vectorrecord`, not HyperMesh vector entities.

`HM_nodeprojecttoplane()`

`HM_vectoranalysis()`

`HM_vectorangle()`

`HM_vectorcrossproduct()`

`HM_vectordotproduct()`

`HM_vectorfromthreepoints()`

`HM_vectormagnitude()`

`HM_vectornormalcrossproduct()`

`HM_vectornormaldotproduct()`

Vector Functions Example Program

HM_nodeprojecttoplane()

Projects a node to a plane along a vector.

Syntax int HM_nodeprojecttoplane(double node[3], HM_planepointer plane, HM_vectorpointer vector);

node[3]	The nodal position to be projected. This variable is overwritten with the result of the projection.
plane	The plane to which the node is to be projected.
vector	The vector along which the node is to be projected.

Returns If successful, `HM_nodeprojecttoplane()` returns zero; otherwise, nonzero.

HM_vectoranalysis()

Creates a vector from two points.

Syntax int HM_vectoranalysis(HM_vectorpointer vector, double point1[3], double point2[3]);

vector	A pointer to a vector where the result of *point2* - *point1* should be stored.
point1[3]	A point in space.
point2[3]	A point in space.

Returns If successful, the function returns zero; otherwise, nonzero.

HM_vectorangle()

Calculates the angle between two vectors.

Syntax double HM_vectorangle(HM_vectorpointer vector1, HM_vectorpointer vector2);

vector1	A pointer to a vector.
vector2	A pointer to a vector.

Returns The angle in radians.

HM_vectorcrossproduct()

Calculates the cross product of two vectors.

Syntax int HM_vectorcrossproduct(HM_vectorpointer result, HM_vectorpointer vector1, HM_vectorpointer vector2);

result	A pointer to a vector where the result of the cross product should be stored.
vector1	A pointer to a vector.
vector2	A pointer to a vector.

Returns If successful, `HM_vectorcrossproduct()` returns zero; otherwise, nonzero.

HM_vectordotproduct()

Calculates the dot product of two vectors.

Syntax double HM_vectordotproduct(HM_vectorpointer vector1, HM_vectorpointer vector2);

vector1	A pointer to a vector.
vector2	A pointer to a vector.

Returns The dot product of `*vector1` and `*vector2`.

HM_vectorfromthreepoints()

Calculates a vector defined by three points.

Syntax int HM_vectorfromthreepoints(HM_vectorpointer vector, double point1[3], double point2[3], double point3[3]);

vector	A pointer to a vector where the result of the cross product should be stored.
point1[3]	A point in space; base of resultant vector.
point2[3]	A point in space.
point3[3]	A point in space.

Returns If successful, the function returns zero; otherwise, nonzero.

Comments This function calculates the vector formed by crossing the vector formed from **point1* to **point2*, and the vector formed from **point1* to **point3*.

HM_vectormagnitude()

Calculates the magnitude of a vector.

Syntax int HM_vectormagnitude(HM_vectorpointer vector);

vector A pointer to a vector.

Returns If successful, the function returns zero; otherwise, nonzero.

Comments This function calculates the magnitude of a vector using the “dist” portion of the vector structure. After the function is called, the “unit” and “magnitude” portion of the vector structure is set to values that represent the unit vector and magnitude of “dist.”

HM_vectornormalcrossproduct()

Calculates the normalized cross product of two vectors.

Syntax int HM_vectornormalcrossproduct(HM_vectorpointer result, HM_vectorpointer vector1, HM_vectorpointer vector2);

result A pointer to a vector where the result of the cross product should be stored.

vector1 A pointer to a vector.

vector2 A pointer to a vector.

Returns If successful, `HM_vectornormalcrossproduct()` returns zero; otherwise, nonzero.

HM_vectornormaldotproduct()

Calculates the normalized dot product of two vectors.

Syntax double HM_vectornormaldotproduct(HM_vectorpointer vector1, HM_vectorpointer vector2);

vector1 A pointer to a vector.

vector2 A pointer to a vector.

Returns The unit dot product of **vector1* and **vector2*.

Comments `HM_vectormagnitude()` should be called before calling this function to ensure that the “unit” portion of the vector structure contains the proper values.

Vector Functions Example Program

```
#include <stdio.h>
#include <stdlib.h>
#include “hmlib.h”

 void main(void)
 {
   HM_vectorrecord vectorrec1;
   HM_vectorrecord vectorrec2;
   HM_vectorrecord vectorrec3;
   HM_planerecord planerec;
   double point1[3];
   double point2[3];
   double point3[3];

   /* To find the dot product of the x and y axis */
   point1[0] = 0.0;
   point1[1] = 0.0;
   point1[2] = 0.0;
   point2[0] = 1.0;
   point2[1] = 0.0;
   point2[2] = 0.0;
   HM_vectoranalysis(&vectorrec1,point1,point2);
   point1[0] = 0.0;
   point1[1] = 0.0;
   point1[2] = 0.0;
   point2[0] = 0.0;
   point2[1] = 1.0;
   point2[2] = 0.0;
   HM_vectoranalysis(&vectorrec2,point1,point2);
   printf(“Dot product of x and y axis: %f\n”,
```

```
HM_vectordotproduct(&vectorrec1,&vectorrec2));
   /* To find the cross product of the x and y axis */
   HM_vectorcrossproduct(&vectorrec3,&vectorrec1,&vectorrec2);
   printf("Cross product of x and y axis: (%f %f %f)\n",
   vectorrec3.unit[0],vectorrec3.unit[1],vectorrec3.unit[2]);
/* To find the angle between the x and y axis */
printf("Angle between the x and y axis:
%f\n",HM_vectorangle(&vectorrec1,&vectorrec2));
/* To find the vector formed from three points */
point1[0] = 0.0;
point1[1] = 0.0;
point1[2] = 0.0;
point2[0] = 1.0;
point2[1] = 0.0;
point2[2] = 0.0;
point3[0] = 0.0;
point3[1] = 1.0;
point3[2] = 0.0;
HM_vectorfromthreepoints(&vectorrec3,point1,point2,point3);
printf("Vector from three points: (%f %f %f)\n",
vectorrec3.unit[0],vectorrec3.unit[1],vectorrec3.unit[2]);
/* To project a point to a plane */
planerec.normal.dist[0] = 1.0;
planerec.normal.dist[1] = 0.0;
planerec.normal.dist[2] = 0.0;
HM_vectormagnitude(&planerec.normal);
planerec.base[0] = 0.0;
planerec.base[1] = 0.0;
planerec.base[2] = 0.0;
point1[0] = 10.0;
point1[1] = 10.0;
point1[2] = 10.0;
```

```
   HM_nodeprojecttoplane(point1,&planerec,&planerec.normal);
   printf("Point (10.0,10.0,10.0) project to (0.0,0.0,0.0)\n");
   printf("  along a vector (1.0,0.0,0.0): (%f %f %f)\n",
   point1[0],point1[1],point1[2]);
}
```

Miscellaneous Functions

The miscellaneous functions in `hmlib` provide you with functions for terminating programs, sending messages, and reading and writing to binary files.

`HM_elementconfiggetnumberofnodes()`

`HM_entitynamegettype()`

`HM_entitytypegetname()`

`HM_entitystringtypegettype()`

`HM_entitytypegettypestring()`

`HM_fread()`

`HM_fwrite()`

`HM_getmachinesizeofdouble()`

`HM_getmachinesizeoffloat()`

`HM_getmachinesizeofint()`

`HM_getmachinetype()`

`HM_message()`

`HM_setterminatefunction()`

`HM_tableadd()`

`HM_tableclear()`

`HM_tablelookup()`

`HM_terminate()`

Miscellaneous Functions Example Program

HM_elementconfiggetnumberofnodes()

For an element configuration, this function returns the number of nodes that the element uses. For rigidlinks and RBE3s, this function returns 0 because these elements have a variable number of nodes.

Syntax int HM_elementconfiggetnumberofnodes(int config);

config The configuration of the element.

Returns The number of nodes used by the element.

HM_entitynamegettype()

Returns the HyperMesh #defined type number of an entity, given its name.

Syntax HM_entitynamegettype(char *name)

name A string representing the name of the entity (“elems,” “comps,” “nodes,” etc.)

Returns The value assigned to the HyperMesh #defined entity type.

HM_entitytypegetname()

Returns the name of an entity, given the HyperMesh #defined type number.

Syntax char *HM_entitytypegetname(HM_entitytype entities, int style)

entities The HyperMesh #defined type number.

style Controls the format of the returned string.

1 Returns an 8-character name, i.e., “comps.”

2 Returns a 16-character name, i.e., “components.”

3 Returns a 16-character, properly capitalized name, i.e., “Components.”

Returns The name of the HyperMesh #defined entity number (i.e., “comps,” “elems,” etc.).

HM_entitystringtypegettype()

Returns the HyperMesh #defined type number of an entity, given its #defined name.

Syntax HM_entitytype HM_entitystringtypegettype(char *name)

name The HyperMesh #defined entity name (“HM_ENTITYTYPE_ELEMS,” “HM_ENTITYTYPE_COMPS,” etc.).

Returns The value assigned to the HyperMesh #defined entity string type.

HM_entitytypegettypestring()

Returns the string associated with the HyperMesh #defined entity, given the type number.

Syntax char *HM_entitytypegettypestring(HM_entitytype type)

type The HyperMesh #defined entity type number (HM_ENTITYTYPE_ELEMS, HM_ENTITYTYPE_COMPS, etc.).

Returns The string associated with the HyperMesh #defined entity type number.

HM_fread()

Reads information from a binary file.

Syntax void HM_fread(void * ptr, int size, int n, FILE * file);

ptr A pointer to a block of memory where information is stored once it is read.

size The size of each item to be read.

n The number of items to be read.

file A pointer to an open binary file.

Returns Nothing.

Comments **ptr* must point to a block of memory that is large enough to hold the result of the read. If the read is not successful, the function calls HM_terminate() with an appropriate error message.

HM_fwrite()

Writes information to a binary file.

Syntax void HM_fwrite(void * ptr, int size, int n, FILE * file);

ptr	A pointer to a block of memory to be written to the file.
size	The size of each item to be written.
n	The number of items to be written.
file	A pointer to an open binary file.

Returns Nothing.

Comments If the write is not successful, the function calls `HM_terminate()` with an appropriate error message.

HM_getmachinesizeofdouble()

Gets the hardware size of a double.

Syntax int HM_getmachinesizeofdouble(int machine);

machine	The value returned from `HM_getmachinetype()`.

Returns The size of a double.

HM_getmachinesizeoffloat()

Gets the hardware size of a float.

Syntax int HM_getmachinesizeoffloat(int machine);

machine	The value returned from `HM_getmachinetype()`

Returns The size of a float.

Comments Most hardware platforms return 4 bytes, except the Cray, which returns 8 bytes.

HM_getmachinesizeofint()

Gets the hardware size of an integer.

Syntax int HM_getmachinesizeofint(int machine);

machine The value returned from `HM_getmachinetype()`.

Returns The size of an integer.

Comments Most hardware platforms return 4 bytes, except the Cray, which returns 8 bytes.

HM_getmachinetype()

Identifies the type of machine on which the program is currently running.

Syntax int HM_getmachinetype(void);

Returns The type of machine on which the program is running. The valid types are found in the header file `hmlib`.h.

HM_message()

Prints a message on the screen.

Syntax void HM_message(char * format, char * message);

format A pointer to a character string that contains a format statement to be used in the `printf` family of functions.

message A pointer to a character string that contains a message.

Returns Nothing.

Comments If message is set to NULL, it is not used in the function.

HM_setterminatefunction()

Sets a pointer to a function that is called by `HM_terminate()`.

Syntax void HM_setterminatefunction(void (*func)(char *format, char *message, void *data), void *data);

func	Pointer to a function that contains the format and message that were passed to `HM_terminate()`. The last argument is a pointer to any data.
data	Pointer to any user-defined data.

Returns Nothing.

Comments Here is an example of using `HM_setterminatefunction:`

```
void MyFunc(char *format, char *message, void *data)
{
  HMIN_message_close();
  HM_message("%s",data);
  HM_message(format,message);
  HM_message("\nProgram was terminated before completion.",NULL);
}

main()
{
  char *str = "User termination function called";
  HM_setterminatefunction(MyFunc,(void *) str);
  HM_terminate(); /* MyFunc is called */
}
```

HM_tableadd()

Adds an item to the table.

Syntax void HM_tableadd(int key, double value);

key Set to a value that is used to find _value.

value The value that should be stored.

Returns Nothing.

Comments The table is intended to store items for later retrieval.

HM_tableclear()

Clears the table of all stored items.

Syntax void HM_tableclear(void);

Returns Nothing.

Comments See `HM_tableadd()` and `HM_tablelookup()`.

HM_tablelookup()

Retrieves a previously stored value.

Syntax double HM_tablelookup(int key);

key The key associated with a stored value.

Returns The value stored with the key.

Comments See `HM_tableadd()` and `HM_tableclear()`.

HM_terminate()

Terminates a program.

Syntax void HM_terminate(char * format, char * message);

format	A pointer to a character string that contains a format statement to be used in the `printf` family of functions.
message	A pointer to a character string that contains a message.

Returns Nothing.

Comments If a function is defined by `HM_setterminatefunction()`, `HM_terminate` calls this function and then calls exit. No messages are displayed; this is left to `HM_setterminatefunction()`.

Miscellaneous Functions Example Program

```
#include <stdio.h>
#include <stdlib.h>
#include “hmlib.h”

 void main(void)
 {
   int i;
   /* You may send messages with HM_message */
   HM_message(“This is a message %s.”,”with a string inserted”);

   /*
   The table functions allow you to store values
   with associated keys.  The keys can then be used
   to find the previously stored values.  As an example,
   items 1 through 3, 5 through 10 are added to a
   look up table with a value of 1.0.
   */
   for (i=1; i<=10; i++)
   {
   if (i != 4) HM_tableadd(i,1.0);
   }

   /*
   To find items that were not added to the table.
   */
   for (i=1; i<=10; i++)
   {
     if (HM_tablelookup(i) == 0.0) printf(“%d was not added to the
       table.\n”,i);
   }

 /* Program termination can be accomplished with HM_terminate(). */
 HM_terminate(“This is a %s message”,”termination”);
```

Introduction to hminlib

hminlib is a set of routines which allows you to write an external input translator. These external programs transfer data in any user-understood format directly into a running copy of HyperMesh. All of the routines found in the hminlib library are designed for use with the C programming language. This Help section assumes that you have some previous knowledge of C.

Concepts

Each program developed with hminlib uses the routines within the library to set up communication links with HyperMesh and to transfer data to HyperMesh using a specific protocol. When writing a translator, all low-level communication issues are automatically handled by hminlib. Essentially, the structure of each external translator program is designed to allow for efficient communication between the external program and HyperMesh.

In order to write a translator, you must perform certain programming tasks; a specific series of C functions must be defined which follow the structure of hminlib. All programs that rely on hminlib must begin execution by initializing hminlib. Initialization allocates memory required by hminlib, opens required files, and prepares hminlib to be used. The next function performed by the external program transfers control of the program to hminlib. Once hminlib has received control of the program, it begins calling functions that you define. The purpose of these user-defined functions is to read data from a file which is in the appropriate format, translate the data to a format HyperMesh can understand, and finally, communicate the translated information to HyperMesh.

User-Definable hminlib Functions

Some of the functions that you can define are on a global bookkeeping level; other functions deal more specifically with an individual entity. Global functions are called only once and allow the performance of operations not related to a specific entity. Examples of global functions include an open function, a color function, and a close function. Entity functions perform operations relating to a specific entity. For each entity type in HyperMesh, there is a possibility of up to four different user-definable functions: open entity, get entity, dictionary and close entity. The following table shows the different user-definable functions that may be created and the functions they perform. They are listed in the order in which they are called by hminlib.

Open	HMIN_OPENFUNCTION is called once control of the program is transferred to hminlib. It sets global variables, initializes routines, and performs other tasks related to setup.
Open Entity	HMIN_ENTITYOPENFUNCTION is called once before any of the entities of a specified type are read. It allows you to prepare their program to read a group of entities.
Get Entity	HMIN_ENTITYGETFUNCTION is called repeatedly until all of the requested entities are transferred from the external program to hminlib. When this function is called by hminlib, you are expected to pass the information about each entity of the requested type in the input file to hminlib.
Dictionary	HMIN_ENTITYDICTIONARYFUNCTION is called once after each entity which can have a dictionary is successfully passed to hminlib. It passes dictionary information about the entity to hminlib.

Close Entity HMIN_ENTITYCLOSEFUNCTION is called once after all of the entities of a specified type are read. It allows you to clean up after their program has read a group of entities.

Name HMIN_NAMEFUNCTION is called once after all of the entities are processed. It passes the names of collectors, if such information exists in the input file.

Move HMIN_MOVEFUNCTION is called once after all of the entities are processed. It moves elements into different components and is used when elements that do not have property information are read into HyperMesh.

Color HMIN_COLORFUNCTION is called once after all of the entities are processed. It passes color information about the collectors, if such information exists in the input file.

Associate HMIN_ASSOCIATEFUNCTION is called after all of the entities are processed. It associates nodes with surfaces.

Close HMIN_CLOSEFUNCTION is called last. It frees memory, closes routines, and does other cleanup functions.

Entity Calling Sequence

For each entity that can exist in a HyperMesh database, a set of three entity level functions (described in the User-Definable hminlib Functions section of this Help guide) can be defined. Since the user-defined functions for each of these entities may be interdependent, the entity calling sequence plays an important role when writing an input translator. The order in which `hminlib` calls the entity functions is listed below, followed by the designations used for each entity:

Null Entity	HM_ENTITYTYPE_NULL
Cards	HM_ENTITYTYPE_CARDS
System Collectors	HM_ENTITYTYPE_SYSTCOLS
Systems	HM_ENTITYTYPE_SYSTS
Nodes	HM_ENTITYTYPE_NODES
Vector Collectors	HM_ENTITYTYPE_VECTORCOLS
Vectors	HM_ENTITYTYPE_VECTORS
Materials	HM_ENTITYTYPE_MATS
Properties	HM_ENTITYTYPE_PROPS
Components	HM_ENTITYTYPE_COMPS
Groups	HM_ENTITYTYPE_GROUPS
Elements	HM_ENTITYTYPE_ELEMS
Load Collectors	HM_ENTITYTYPE_LOADCOLS
Loads	HM_ENTITYTYPE_LOADS
Equations	HM_ENTITYTYPE_EQUATIONS
Geometry Database	HM_ENTITYTYPE_GEOMETRY
Lines	HM_ENTITYTYPE_LINES
Surfs	HM_ENTITYTYPE_SURFS
Points	HM_ENTITYTYPE_POINTS
Assemblies	HM_ENTITYTYPE_ASSEMS
Curves	HM_ENTITYTYPE_CURVES
Plots	HM_ENTITYTYPE_PLOTS
Blocks	HM_ENTITYTYPE_BLOCKS
Titles	HM_ENTITYTYPE_TITLES
Sets	HM_ENTITYTYPE_SETS
Outputblocks	HM_ENTITYTYPE_OUTPUTBLOCKS
Loadsteps	HM_ENTITYTYPE_LOADSTEPS

Max Entity Count HM_ENTITYTYPE_MAX

Writing an Input Translator

The first step in the process of writing an input translator is to define an input format that needs translation. In this example, the definition of the input deck consists of a finite element model that comprises nodes, tria and quad elements, constraints, and forces. In addition, the elements have a property and a material associated with them, and the property and material information is also in the input deck. Given this information, a sample input deck may look something like this:

```
node, 1,   0.0,   0.0,   0.0
node, 2, 100.0,   0.0,   0.0
node, 3, 100.0, 100.0,   0.0
node, 4,   0.0, 100.0,   0.0
node, 5,   0.0,   0.0, 100.0
node, 6, 100.0,   0.0, 100.0
node, 7, 100.0, 100.0, 100.0
node, 8,   0.0, 100.0, 100.0
quad, 1, 1, 1, 2, 3, 4
tria, 2, 2, 5, 6, 7
tria, 3, 2, 5, 7, 8
property, 1, 1, 0.75
property, 2, 1, 1.00
material, 1, 20000.0, 0.3
hmcolor, component, 1, 9
hmcolor, component, 2, 10
hmname, component, 1, comp1
hmname, component, 2, comp2
hmname, material, 1, material
const, 1, 123456
const, 2, 123456
const, 5, 123456
const, 6, 123456
force, 3, 0.0, 0.0, 10.0
force, 7, 0.0, 0.0, 10.0
```

The task of writing an input translator is now defined. The entities that exist in the input deck must be transferred to HyperMesh; more specifically, the nodes, elements, constraints, forces, property, and material information must be translated into a format understood by HyperMesh. Given this objective, and with the use of the `hminlib` functions, it is now possible to write an input translator.

The source code for the example is listed at the end of the section. As the example progresses, it is recommended that you open your manual and flip to the end of the section in order to follow the source code while reading.

As described above, the structure of an input translator written with the program follows a specific

format. The first step that an input translator must perform is to initialize `hminlib`. This is performed by calling the function `HMIN_init()`. Note that in the example source code below, the function `main()` calls `HMIN_init()`, which takes four parameters. The four parameters are the name of the translator, the version of the translator, and then the argc and argv parameters, which are passed into `main()`. Also note that `HMIN_init()` returns a pointer to the input file to be read. `HMIN_setsolver()` is then called to associate attributes with a given template (see `*codename()` in the template section).

In the next line of the program, control is passed to `hminlib`. This is accomplished with the function `HMIN_readmodel()`, which takes a pointer to a function as its only parameter. The function pointer points to a function from which `hminlib` can retrieve the user-defined functions used to read in an input file. For this example, refer to this function as the inquire function. Note that the inquire function, passed in `HMIN_readmodel()`, takes two arguments. The first argument, function, is the type of function `hminlib` is requesting. As described above, one of the user-definable functions is the open function, and the function returns fetestopen() when the value `HMIN_OPENFUNCTION` is requested. Prepare the inquire function to deal with the following values for function:

HMIN_OPENFUNCTION

HMIN_ENTITYOPENFUNCTION

HMIN_ENTITYGETFUNCTION

HMIN_ENTITYDICTIONARYFUNCTION

HMIN_ENTITYCLOSEFUNCTION

HMIN_COLORFUNCTION

HMIN_NAMEFUNCTION

HMIN_MOVEFUNCTION

HMIN_CLOSEFUNCTION

The second parameter only applies to the entity functions, `HMIN_ENTITYOPENFUNCTION`, `HMIN_ENTITYGETFUNCTION`, `HMIN_ENTITYDICTIONARYFUNCTION`, and `HMIN_ENTITYCLOSEFUNCTION`. The second parameter is the entity type `hminlib` is requesting. Any of the entity types listed above are valid for this parameter, but always set the parameter to zero if the function being requested does not relate to an entity. Note that the inquire function must return `NULL` if the function being requested is not defined.

The last step in the program is to close `hminlib`; this is accomplished by calling the function `HMIN_close()`. `HMIN_close()` frees memory and closes files and other resources used during the translation process. Note that `main()` returns an integer value, and in order for HyperMesh to work properly with the input translator, the function `main()`must return zero.

hminlib Source Code Example

The following is a listing of the source code used in the previous section. The source code is included with HyperMesh and is found in the src directory in the file `fetest.c`.

```
/*

    FETEST (c) copyright Altair/Finite Applications 1994.

    Using hminlib, this program reads a finite element
    model from an ASCII file and passes the model information
    to HyperMesh.  For more information on the usage of
    hminlib please refer to the HyperMesh Programmer's Manual,
    which is available upon request.

    A makefile is included with this example.  To compile the
    program, first make sure that the location of the include and
    libraries are specified.  This is performed by setting
    the environment variable as follows:

       setenv HM_DIRECTORY <HyperMesh Directory>

       where:

         <HyperMesh Directory> is the name of the
                               HyperMesh directory.

    After the HM_DIRECTORY environment variable is
    set, type "make'.  Make produces the executable
    'fetest'.

*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "hmlib.h"
#include "hminlib.h"

FILE *FETestInputFile = NULL;
char FETestInputLine[256];
```

```
int FETestInputLineIndex;

int fetestgetline(void)
{
  /*
      This function reads a line from an ASCII file.
  */

  static int c = 0;
  int longline = 0;

  FETestInputLineIndex = 0;
  memset(FETestInputLine,0,sizeof(FETestInputLine));
  if (c == -1)
  {
    c = 0;
    return(1);
  }
  c = fgetc(FETestInputFile);
  while (c != -1 && c != '\n')
  {
    if (FETestInputLineIndex < 255)
    {
      FETestInputLine[FETestInputLineIndex++] = c;
    }
    else
    {
      longline = 1;
    }
    c = fgetc(FETestInputFile);
  }
  if (longline) HMIN_message_send(0,"Input line too long.","The line was
    truncated.");
  HM_strlwr(FETestInputLine);
  FETestInputLineIndex = 0;
  return(0);
}

char *fetestgetstring(char *string, int maxsize)
{
  /*
      This function reads a string field from the line previously read.
```

```
  */

  int i = 0;

  while (FETestInputLine[FETestInputLineIndex] != ',' &&
    FETestInputLine[FETestInputLineIndex] && i < maxsize)
  {
    string[i++] = FETestInputLine[FETestInputLineIndex++];
  }
  if (FETestInputLine[FETestInputLineIndex] == ',')
FETestInputLineIndex++;
  string[i] = 0;
  return(string);
}

double fetestgetdouble(void)
{
  /*
      This function reads a double field from the previously read line.
  */

  char string[256];

  fetestgetstring(string,256);
  return(atof(string));
}

int fetestgetint(void)
{
  /*
      This function reads an integer field from the previously read line.
  */

  char string[256];

  fetestgetstring(string,256);
  return(atoi(string));
}

void fetestresetfile(void)
{
  /*
```

```
      This function resets the input file.
  */

  rewind(FETestInputFile);
}

int fetestopen(void)
{
  /*
      This user-defined function allows you to perform setup
      operations for the program.  This is the first user
      function called by hminlib.
  */

  return(0);
}

int fetestgetnode(void)
{
  /*
      This user-defined function reads all of the nodes from the
      file and passes them to hminlib.
  */

  char string[256];
  int id;
  double coords[3];

  fetestresetfile();
  while (!fetestgetline())
  {
    fetestgetstring(string,256);
    if (!strcmp(string,"node"))
    {
      id = fetestgetint();
      coords[0] = fetestgetdouble();
      coords[1] = fetestgetdouble();
      coords[2] = fetestgetdouble();
      HMIN_node_write(id,coords,0,0,0,0,0);
    }
  }
  return(0);
```

```
}

int fetestgetelement(void)
{
  /*
      This user-defined function reads all of the elements in the
      file and passes them to hminlib.
  */

  char string[256];
  int id;
  int propertyid;
  HM_entityidtype nodes[20];

  fetestresetfile();
  while (!fetestgetline())
  {
    fetestgetstring(string,256);
    if (!strcmp(string,"quad"))
    {
      id = fetestgetint();
      propertyid = fetestgetint();
      nodes[0] = fetestgetint();
      nodes[1] = fetestgetint();
      nodes[2] = fetestgetint();
      nodes[3] = fetestgetint();
      HMIN_element_writequad4(id,1,nodes,propertyid);
    }
    else if (!strcmp(string,"tria"))
    {
      id = fetestgetint();
      propertyid = fetestgetint();
      nodes[0] = fetestgetint();
      nodes[1] = fetestgetint();
      nodes[2] = fetestgetint();
      HMIN_element_writetria3(id,1,nodes,propertyid);
    }
  }
  return(0);
}

int fetestgetcomponent(void)
```

```
{
  /*
      This user-defined function reads the components in the
      file and passes them to hminlib.
  */

  char string[256];
  HM_entityidtype id;
  HM_entityidtype materialid;
  double thickness;

  fetestresetfile();
  while (!fetestgetline())
  {
    fetestgetstring(string,256);
    if (!strcmp(string,"property"))
    {
      id = fetestgetint();
      materialid = fetestgetint();
      thickness = fetestgetdouble();
      HMIN_component_write(id,"",materialid,HMIN_DEFAULT_COLOR);
      HMIN_dictionary_write(HM_ENTITYTYPE_COMPS,id,"T","real","",
       thickness,1);
    }
  }
  return(0);
}

int fetestgetmaterial(void)
{
  /*
      This user-defined function reads the materials in the file
      and passes them to hminlib.
  */

  char string[256];
  HM_entityidtype id;
  double E;
  double Nu;

  fetestresetfile();
  while (!fetestgetline())
```

```
  {
    fetestgetstring(string,256);
    if (!strcmp(string,"material"))
    {
      id = fetestgetint();
      E = fetestgetdouble();
      Nu = fetestgetdouble();
      HMIN_material_write(id,"");
      HMIN_dictionary_write(HM_ENTITYTYPE_MATS,id,"E","real", "",E,1);
      HMIN_dictionary_write(HM_ENTITYTYPE_MATS,id,"Nu","real", "",Nu,1);
    }
  }
  return(0);
}

int fetestgetloadcollector(void)
{
  /*
      This user-defined function creates a load collector where
      the loads being read are placed.
  */

  HMIN_loadcollector_write(1,"loads",12);
  return(0);
}

int fetestgetload(void)
{
  /*
      This user-defined function reads the loads from the file and
      passes them to hminlib.
  */

  char string[256];
  HM_entityidtype id;
  char flags[256];
  double x;
  double y;
  double z;
  double components[6];

  fetestresetfile();
```

```
  while (!fetestgetline())
  {
    fetestgetstring(string,256);
    if (!strcmp(string,"const"))
    {
      id = fetestgetint();
      fetestgetstring(flags,256);
      components[0] = strchr(flags,'1') ? 0.0 : -999999.0;
      components[1] = strchr(flags,'2') ? 0.0 : -999999.0;
      components[2] = strchr(flags,'3') ? 0.0 : -999999.0;
      components[3] = strchr(flags,'4') ? 0.0 : -999999.0;
      components[4] = strchr(flags,'5') ? 0.0 : -999999.0;
      components[5] = strchr(flags,'6') ? 0.0 : -999999.0;
      HMIN_load_writeconstraint(HMIN_maximumids_getnext
       (HM_ENTITYTYPE_LOADS),id,0,components,1);
    }
    else if (!strcmp(string,"force"))
    {
      id = fetestgetint();
      x = fetestgetdouble();
      y = fetestgetdouble();
      z = fetestgetdouble();
      HMIN_load_writeforce(HMIN_maximumids_getnext
       (HM_ENTITYTYPE_LOADS),id,0,x,y,z,1);
    }
  }
  return(0);
}

int fetestname(void)
{
  /*
      This user-defined function reads the name information
      contained in the file and passes it to hminlib.
  */

  char string[256];
  char typename[256];
  HM_entityidtype id;
  char name[256];

  fetestresetfile();
```

```
  while (!fetestgetline())
  {
    fetestgetstring(string,256);
    if (!strcmp(string,"hmname"))
    {
      fetestgetstring(typename,256);
      id = fetestgetint();
      fetestgetstring(name,256);
      HMIN_name_write(typename,id,name);
    }
  }
  return(0);
}

int fetestcolor(void)
{
  /*
      This user-defined function reads the color information
      contained in the file and passes it to hminlib.
  */

  char string[256];
  char typename[256];
  char name[256];
  int color;

  fetestresetfile();
  while (!fetestgetline())
  {
    fetestgetstring(string,256);
    if (!strcmp(string,"hmcolor"))
    {
      fetestgetstring(typename,256);
      fetestgetstring(name,256);
      color = fetestgetint();
      HMIN_color_write(typename,name,color);
    }
  }
  return(0);
}

entityfunctionptr fetestgetfunction(int function, HM_entitytype entities)
```

```
{
  /*
      This user-defined function is passed into hminlib and is
      used by hminlib to find all of the user-defined functions
      which perform reading and information passing.  Note
      that if a user-defined function is not required, this function
      must return NULL.
  */

  switch (function)
  {
    case HMIN_OPENFUNCTION:
      return(fetestopen);
    case HMIN_ENTITYOPENFUNCTION:
      break;
    case HMIN_ENTITYGETFUNCTION:
      switch (entities)
      {
        case HM_ENTITYTYPE_NODES:
          return(fetestgetnode);
        case HM_ENTITYTYPE_ELEMS:
          return(fetestgetelement);
        case HM_ENTITYTYPE_COMPS:
          return(fetestgetcomponent);
        case HM_ENTITYTYPE_MATS:
          return(fetestgetmaterial);
        case HM_ENTITYTYPE_LOADCOLS:
          return(fetestgetloadcollector);
        case HM_ENTITYTYPE_LOADS:
          return(fetestgetload);
      }
      break;
    case HMIN_ENTITYCLOSEFUNCTION:
      break;
    case HMIN_COLORFUNCTION:
      return(fetestcolor);
      break;
    case HMIN_NAMEFUNCTION:
      return(fetestname);
      break;
    case HMIN_CLOSEFUNCTION:
      break;
```

```
  }
  return(NULL);
}

int main(int argc, char *argv[])
{
  /*  Initialize hminlib  */

  FETestInputFile = HMIN_init("FETEST","1.0",argc,argv);

  /*
    Associate attributes to a user-defined templere used with the card
    previewer.
  /*

  HMIN_Setsolver(100);

  /*  Pass the user-defined function shown above and read the model. */

  HMIN_readmodel(fetestgetfunction);

  /*  Close hminlib  */

  HMIN_close();

  /*  All main functions must return zero if successful. */

  return(0);
}
```

The hminlib Functions

Basic Functions

File Locators

Initializing, Setting, and Retrieving IDs

Moving Entities

Renaming Collectors

Sending a Message to HyperMesh

Transferring Entities to HyperMesh

Basic Functions

`HMIN_close()`

`HMIN_combineloads()`

`HMIN_convertentitynametotype()`

`HMIN_deleteemptycomponents()`

`HMIN_infilename()`

`HMIN_init()`

`HMIN_readmodel()`

`HMIN_setsolver()`

`HMIN_setviewangles()`

HMIN_close()

Closes `hminlib`.

Syntax void HMIN_close(void);

Returns Nothing.

HMIN_combineloads()

Determines if HyperMesh combines loads on the same node.

Syntax void HMIN_combineloads(int flag)

flag If "flag" is zero, HyperMesh does not combine loads with the same config if they are acting on the same node. By default, HyperMesh combines loads (for backwards compatibility).

Returns Nothing.

Comments `*HMIN_combineloads()`must be called after `*HMIN_init()`.

HMIN_convertentitynametotype()

Provides a way to convert the name of an entity to the entity type used in HyperMesh.

Syntax int HMIN_convertentitynametotype(char * string);

string A pointer to the string containing the entity name.

Returns The entity type used in HyperMesh.

HMIN_deleteemptycomponents()

Sets a flag to delete, or not delete, empty components.

Syntax void HMIN_deleteemptycomponents(int flag);

flag If "flag" is 0, HyperMesh does not delete empty components after a model is read.

Returns Nothing.

Comments HyperMesh deletes empty components after a model is read by the ***import*** panel unless `HMIN_deleteemptycomponents(0)` is called. This call must be made before `HMIN_readmodel()`.

HMIN_infilename()

Retrieves the name of the file being read for input.

Syntax char *HMIN_infilename(void);

Returns The name of the input file.

HMIN_init()

Initializes `hminlib`. This function must be called before using any other function in `hminlib`.

Syntax FILE * HMIN_init(char * formatname, char * version, int argc, char * argv[]);

formatname	A character string set to the name of the translator.
version	The version number of the translator that is being written.
argc	Argument passed in from the C function main.
argv[]	Argument passed in from the C function main.

Returns Nothing.

Comments HMIN_init calls `HM_setterminatefunction()` to terminate `hminlib`. If you want to call `HM_setterminatefunction()` and set your own terminate function, you must call it after `HMIN_init`. HMIN_message_close should be called by your new terminate function.

HMIN_readmodel()

Allows `hminlib` to read in all of the entities in the database.

Syntax void HMIN_readmodel(entityfunctionptr (* getfunction) (int function, HM_entitytype entities));

getfunction	A pointer to the function that requests the functions that read each entity type.
function	The function to be performed.
entities	The entity type to be read in.

Returns Nothing.

HMIN_setsolver()

Sets the solver type for feinput.

Syntax void HMIN_setsolver(int code);

code	An integer value between 0 and 127 that corresponds to the value defined in the `*codename()` template function.

Returns Nothing.

Comments As attributes are created, they are assigned a specific solver number, 0 through 127. When you write a deck for a specific solver, the template system only uses attributes that match the solver number specified in the template with the `*codename()`command. Templates distributed with HyperMesh use solver numbers 0 through 63. Altair customers use solver numbers 64 through 127. Call `HMIN_setsolver()` after `HMIN_init()` if attributes are being used.

HMIN_setviewangles()

Sets the view to the desired angle as read in the translator.

Syntax HMIN_setviewangles(THETAX,THETAY,THETAZ)

THETAX	The angle in degrees.
THETAY	The angle in degrees.
THETAZ	The angle in degrees.

Returns Nothing.

File Locators

`HMIN_filelocator_calculaterange()`

`HMIN_filelocator_continuereading()`

`HMIN_filelocator_create()`

`HMIN_filelocator_destroy()`

`HMIN_filelocator_getfound()`

`HMIN_filelocator_positionfile()`

`HMIN_filelocator_set()`

HMIN_filelocator_calculaterange()

Calculates the range between file locators.

Syntax void HMIN_filelocator_calculaterange(void * locatorptr, int startlocator, int endlocator);

locatorptr A pointer to the block of file locators.

startlocator The index into the locator block at the beginning of the range to be calculated.

endlocator The index into the locator block at the ending of the range to be calculated.

Returns Nothing.

HMIN_filelocator_continuereading()

Indicates if the last of a specified entity is read.

Syntax int HMIN_filelocator_continuereading(void * filelocatorptr, int filelocator, FILE * file);

filelocatorptr The pointer to the block of file locators.

filelocator The index into the block of file locators.

file The entity to be read.

Returns 1, if it has not read through to the end of the block.

0, if it has reached the end of the block.

HMIN_filelocator_create()

Creates storage in memory where you can store a block of file locators.

Syntax void * HMIN_filelocator_create(int numberoffilelocators);

numberoffilelocators The number of entities you want to keep track of in the file.

Returns Nothing.

HMIN_filelocator_destroy()

Frees the memory that was used for file locators.

Syntax void HMIN_filelocator_destroy(void * filelocatorptr);

filelocatorptr The pointer to the block of file locators.

Returns Nothing.

HMIN_filelocator_getfound()

Determines if an entity is present within the given file.

Syntax int HMIN_filelocator_getfound(void * filelocatorptr, int filelocator);

filelocatorptr The pointer to the block of file locators.

filelocator The index into the block of file locators.

Returns 1, if the entity is found.

0, if the entity is not found.

HMIN_filelocator_positionfile()

Provides a way to reposition the file.

Syntax void HMIN_filelocator_positionfile(void * filelocatorptr, int filelocator, FILE * file);

filelocatorptr The pointer to the block of file locators.

filelocator The index into the block of file locators.

file The file to be repositioned.

Returns Nothing.

HMIN_filelocator_set()

Assigns a location for the file locator.

Syntax void HMIN_filelocator_set(void * filelocatorptr, int filelocator, fpos_t beginposition, fpos_t endposition);

filelocatorptr	The pointer to the block of file locators.
filelocator	An index into the block of file locators.
beginposition	The beginning position of the specified entity in the file.
endposition	The end position of the specified entity in the file.

Returns Nothing.

Initializing, Setting, and Retrieving IDs

`HMIN_maximumids_get()`

`HMIN_maximumids_getnext()`

`HMIN_maximumids_initialize()`

`HMIN_maximumids_reset()`

`HMIN_maximumids_set()`

HMIN_maximumids_get()

Retrieves the value of the current maximum ID.

Syntax HM_entityidtype HMIN_maximumids_get(HM_entitytype entitytype);

entitytype	The type of entity.

Returns The current maximum ID.

HMIN_maximumids_getnext()

Retrieves the next ID and adds 1 to the maximum ID.

Syntax HM_entityidtype HMIN_maximumids_getnext(HM_entitytype entitytype);

entitytype	The type of entity.

Returns The next value of ID.

HMIN_maximumids_initialize()

Initializes the maximum ID.

Syntax void HMIN_maximumids_initialize(void);

Returns Nothing.

Comments Sets the maximum ID to zero.

HMIN_maximumids_reset()

Resets the maximum ID to zero.

Syntax void HMIN_maximumids_reset(HM_entitytype entitytype);

entitytype The type of entity.

Returns Nothing.

HMIN_maximumids_set()

Assigns a maximum ID to the entities.

Syntax void HMIN_maximumids_set(HM_entitytype entitytype, HM_entityidtype id);

entitytype The type of entity.

id The ID to be assigned to the entity.

Returns Nothing.

Moving Entities

`HMIN_move_write()`

`HMIN_move_writeid()`

HMIN_move_write()

Allows an element to be moved from one collector to another.

Syntax void HMIN_move_write(char * name);

name The name of the collector where the element should be placed.

Returns Nothing.

Comments `HMIN_move_writeid()` must be called next to identify the entity that should be moved. No other `hminlib` functions may be called until *all* moves for the given collector are complete.

HMIN_move_writeid()

Moves the entity identified by ID to the new location.

Syntax void HMIN_move_writeid(HM_entityidtype id);

id The ID of the element.

Returns Nothing.

Comments `HMIN_move_write()` must be called first to indicate the name of the collector where the entity should be moved.

Renaming Collectors

`HMIN_name_write()`

HMIN_name_write()

Renames a collector.

Syntax void HMIN_name_write(char * typename, HM_entityidtype id, char * name);

typename The type of collector.

id The ID of the collector.

name The name to be assigned to the collector.

Returns Nothing.

Sending a Message to HyperMesh

`HMIN_extratext()`

`message_headerbar()`

`message_headerbarerror()`

`HMIN_message_send()`

HMIN_extratext()

Saves supplied text to an `.hmx` file

Syntax void HMIN_extratext(int key, char *string, int linenumber)

key	0	No header information written along with string.
	1	Identifies the string as unsupported data.
	2	Identifies the string as an unsupported keyword.
string	The extra text item to include in the file.	
linenumber	The line number in the orginal file where the text was found (included if value is > 0 for key values 1 and 2).	

Returns Nothing.

Comments A file named `modelname.hmx` is created in the directory in which the translation is performed. If you make a call to this function, the supplied string and appropriate header information are included in this file.

This call is used for feinput translators to write unsupported information to the `.hmx` file. This allows you to cut and paste the information to the output file created by HyperMesh.

message_headerbar()

Writes a status message to the menu header bar

Syntax void message_headerbar(char *message);

message	The message to be sent.

Returns Nothing.

message_headerbarerror()

Writes an error message to the menu header bar (in red).

Syntax void message_headerbarerror(char *message);

message The message to be sent.

Returns Nothing.

HMIN_message_send()

Allows you to send a message to the message file while a database is being read in.

Syntax void HMIN_message_send(int type, char * message, char * resolution);

type The type of message you want to send. Set the type to:

0 To send information.

1 To send a warning.

2 To send an error message.

message A character string that contains the message you want sent out to the file.

resolution A character string that gives the your resolution to a problem. If a resolution should not be written set to NULL.

Returns Nothing.

Transferring Entities to HyperMesh

Assemblies

Attributes

Blocks

Color

Components

Control Cards

Coordinate Systems

Curves

Dictionaries

Elements

Equations

Groups

Lines

Load Collectors

Nodes

Plots

Properties

Rigid Walls

Sets

Surfaces

System Collectors

Systems

Titles

Vector Collectors

Vectors

Assemblies

`HMIN_assembly_write()`

`HMIN_assembly_writecomponent()`

HMIN_assembly_write()

Writes an assembly to HyperMesh.

Syntax void HMIN_assembly_write(HM_entityidtype id, char * name, int color);

id	The ID of the assembly to be written.
name	The name of the assembly.
color	The color of the assembly.

Returns Nothing.

HMIN_assembly_writecomponent()

Transfers the components of an assembly to HyperMesh.

Syntax void HMIN_assembly_writecomponent(HM_entityidtype componentid);

componentid	The ID of the component to be transferred.

Returns Nothing.

Prerequisite An assembly must exist (`HMIN_assembly_write()`).

Attributes

Attributes are used within HyperMesh to describe solver-specific data and may be attached to entities within the HyperMesh database. Attributes are defined in the HyperMesh template system and are linked to a specific solver through the use of the template command *codename() and the `hminlib` function `HMIN_setsolver()`, both of which must be present in a given template/input translator pair for the attribute system to work properly. When you write attributes to HyperMesh, the attribute identifier and type must match the contents of the template.

`HMIN_writeattribute_string()`

`HMIN_writeattribute_string_array()`

`HMIN_writeattribute_int()`

`HMIN_writeattribute_int_array()`

`HMIN_writeattribute_int_array2d()`

`HMIN_writeattribute_double()`

`HMIN_writeattribute_double_array()`

`HMIN_writeattribute_double_array2d()`

`HMIN_writeattribute_entity()`

HMIN_writeattribute_string()

Transfer a string attribute to HyperMesh.

Syntax void HMIN_writeattribute_string(HM_entitytype entitytype, HM_entityidtype id, int identifier, int behavior, int status, char *value);

entitytype	The type of the entity to which to attach the attribute.
id	The ID of the entity to which to attach the attribute.
identifier	The attribute identifier (defined in a template with the `*defineattribute()` command).
behavior	Set to 0, reserved for future use.
status	Set to 0 for off.
	Set to 1 for on.
	Set to 2 for always on.
value	The value to be assigned to the attribute.

Returns Nothing.

HMIN_writeattribute_string_array()

Transfer a string array attribute to HyperMesh.

Syntax void HMIN_writeattribute_string_array(HM_entitytype entitytype, HM_entityidtype id, int identifier, int behavior, int status, char *data[], int length);

entitytype	The type of the entity to which to attach the attribute.
id	The ID of the entity to which to attach the attribute.
identifier	The attribute identifier (defined in a template with the `*defineattribute()` command).
behavior	Set to 0, reserved for future use.
status	Set to 0 for off.
	Set to 1 for on.
	Set to 2 for always on.
data[]	The values to be assigned to the attribute.
length	The length of the array.

Returns Nothing.

HMIN_writeattribute_int()

Transfer an integer attribute to HyperMesh.

Syntax void HMIN_writeattribute_int(HM_entitytype entitytype, HM_entityidtype id, int identifier, int behavior, int status, int value);

entitytype	The type of the entity to which to attach the attribute.
id	The ID of the entity to which to attach the attribute.
identifier	The attribute identifier (defined in a template with the `*defineattribute()` command).
behavior	Set to 0, reserved for future use.
status	Set to 0 for off.
	Set to 1 for on.
	Set to 2 for always on.
value	The value to be assigned to the attribute.

Returns Nothing.

HMIN_writeattribute_int_array()

Transfer an integer array attribute to HyperMesh.

Syntax void HMIN_writeattribute_int_array(HM_entitytype entitytype, HM_entityidtype id, int identifier, int behavior, int status, int value[], int length);

entitytype	The type of the entity to which to attach the attribute.
id	The ID of the entity to which to attach the attribute.
identifier	The attribute identifier (defined in a template with the `*defineattribute()` command).
behavior	Set to 0, reserved for future use.
status	Set to 0 for off.
	Set to 1 for on.
	Set to 2 for always on.
data[]	The values to be assigned to the attribute.
length	The length of the array.

Returns Nothing.

HMIN_writeattribute_int_array2d()

Transfer a 2-D integer array to HyperMesh.

Syntax void HMIN_writeattribute_int_array2d(HM_entitytype entitytype, HM_entityidtype id, int identifier, int behavior, int status, int *value, int rows, int cols);

entitytype	The type of the entity to which to attach the attribute.
id	The ID of the entity to which to attach the attribute.
identifier	The attribute identifier (defined in a template with the `*defineattribute()` command).
behavior	Set to 0, reserved for future use.
status	Set to 0 for off.
	Set to 1 for on.
	Set to 2 for always on.
data[][]	The values to be assigned to the attribute.
rows	The number of rows in the array.
cols	The number of columns in the array.

Returns Nothing.

HMIN_writeattribute_double()

Transfers a double attribute to HyperMesh.

Syntax void HMIN_writeattribute_double(HM_entitytype entitytype, HM_entityidtype id, int identifier, int behavior, int status, double value);

entitytype	The type of the entity to which to attach the attribute.
id	The ID of the entity to which to attach the attribute.
identifier	The attribute identifier (defined in a template with the `*defineattribute()` command).
behavior	Set to 0, reserved for future use.
status	Set to 0 for off.
	Set to 1 for on.
	Set to 2 for always on.
value	The value to be assigned to the attribute.

Returns Nothing.

HMIN_writeattribute_double_array()

Transfers a double array attribute to HyperMesh.

Syntax void HMIN_writeattribute_double_array(HM_entitytype entitytype, HM_entityidtype id, int identifier, int behavior, int status, double value[], int length);

entitytype	The type of the entity to which to attach the attribute.
id	The ID of the entity to which to attach the attribute.
identifier	The attribute identifier (defined in a template with the `*defineattribute()` command).
behavior	Set to 0, reserved for future use.
status	Set to 0 for off.
	Set to 1 for on.
	Set to 2 for always on.
data[]	The values to be assigned to the attribute.
length	The length of the array.

Returns Nothing.

HMIN_writeattribute_double_array2d()

Transfers a 2-D double array to HyperMesh.

Syntax void HMIN_writeattribute_double_array2d(HM_entitytype entitytype, HM_entityidtype id, int identifier, int behavior, int status, double *value, int rows, int cols);

entitytype	The type of the entity to which to attach the attribute.
id	The ID of the entity to which to attach the attribute.
identifier	The attribute identifier (defined in a template with the `*defineattribute()` command).
behavior	Set to 0, reserved for future use.
status	Set to 0 for off.
	Set to 1 for on.
	Set to 2 for always on.
data[][]	The values to be assigned to the attribute.
rows	The number of rows in the array.
cols	The number of columns in the array.

Returns Nothing.

HMIN_writeattribute_entity()

Transfers an entity attribute to HyperMesh.

Syntax void HMIN_writeattribute_entity(HM_entitytype entitytype, HM_entityidtype id, int identifier, int behavior, int status, HM_entitytype attributetype, int attributeid);

entitytype	The type of the entity to which to attach the attribute.
id	The ID of the entity to which to attach the attribute.
identifier	The attribute identifier (defined in a template with the `*defineattribute()` command).
behavior	Set to 0, reserved for future use.
status	Set to 0 for off.
	Set to 1 for on.
	Set to 2 for always on.
attributetype	The type of entity to which the attribute references.
attributeid	The ID of the entity that is referenced.

Returns Nothing.

Blocks

NOTE After a block is created, no other `hminlib` comments can be called prior to the completion of reading *all* block data for a given block. The sequence for creating blocks is as follows:

```
HMIN_block_write();

HMIN_block_writewall();

HMIN_block_writedivision();

HMIN_block_writecell();
```

HMIN_block_write()

Writes a block to HyperMesh.

Syntax void HMIN_block_write(HM_entityidtype id, char * name, int color, double minimum[3], double maximum[3], int divisions[3]);

id	The ID of the block to be written.
name	The name of the block.
color	The color of the block.
minimum[3]	The minimum values of the x, y, and z coordinates in the block.
maximum[3]	The maximum values of the x, y, and z coordinates in the block.
divisions[3]	The number of divisions on the block for x, y, and z.

Returns Nothing.

HMIN_block_writecell()

Assigns a cell to a wall ID in a block.

Syntax void HMIN_block_writecell(int i, int j, int k, int wallid);

i	The i location of the cell.
j	The j location of the cell.
k	The k location of the cell.
wallid	The ID of the wall where the cell is to be assigned.

Returns Nothing.

Prerequisite A wall must exist (`HMIN_block_writewall()`).

HMIN_block_writedivision()

Assigns a value to each division.

Syntax void HMIN_block_writedivision(int divindex, double cord);

divindex The coordinate of the division to be assigned a value. Use:

0	When the value being assigned is in an x division.
1	When the value being assigned is in a y division.
2	When the value being assigned is in a z division.

cord The value to be assigned to the division.

Returns Nothing.

Prerequisite A wall must exist (`HMIN_block_write()`).

HMIN_block_writewall()

Writes the walls of a block to HyperMesh.

Syntax void HMIN_block_writewall(int wallid, char * name, int color);

wallid The ID of the wall.

name The name of the wall.

color The color of the wall.

Returns Nothing.

Color

`HMIN_color_write()`

HMIN_color_write()

Assigns a color to the designated component.

Syntax void HMIN_color_write(char * typename, char * name, int color);

typename The type of collector to be assigned a color (components, loadcols, systemcols).

name The name of the component to be assigned a color.

color The color to be assigned to the component.

Returns Nothing.

Components

`HMIN_component_write()`

HMIN_component_write()

Writes a component to HyperMesh.

Syntax void HMIN_component_write(HM_entityidtype id, char * name, HM_entityidtype materialid, int color);

id	The ID of the component to be written.
name	The name of the component to be written.
materialid	The material that the component references.
color	The color of the component to be written.

Returns Nothing.

Control Cards

`HMIN_card_write()`

HMIN_card_write()

Writes a control card to HyperMesh.

Syntax void HMIN_card_write(HM_entityidtype id, char *name);

id	The ID of the control card.
name	The name of the control card.

Returns Nothing.

Comments A template that supports control cards must be used in conjunction with the input translator using the `HMIN_card_write()` function.

Prerequisite An association between the `feinput` translator and a HyperMesh template file must exist for control cards to be visible in the card previewer (specified with the feinput function `HMIN_setsolver()` and the template function `*codename()`). See the *Templates* section for a description of the `*codename()` function.

Coordinate Systems

`HMIN_transform_nodetoglobal()`

`HMIN_transform_vectoratpointtoglobal()`

`HMIN_transform_vectortoglobal()`

`HMIN_transform_write()`

HMIN_transform_nodetoglobal()

Allows nodes to be transformed to a global coordinate system, based on the local coordinate system.

Syntax void HMIN_transform_nodetoglobal(double input[3], int type, double origin[3], double axis[3][3], double output[3]);

input[3]	The coordinates of the node.
type	The type of coordinate system.
origin[3]	The origin of the coordinate system.
axis[3][3]	The axes of the coordinate system.
output[3]	The output of the coordinate system.

Returns Nothing.

HMIN_transform_vectoratpointtoglobal()

Allows vectors in a Cartesian, cylindrical, or spherical system to be transformed to a global coordinate system, based on the local system.

Syntax void HMIN_transform_vectoratpointtoglobal(HM_vectorpointer inputptr, double cords[3], int type, double origin[3], double axis[3][3], HM_vectorpointer outputptr);

inputptr	The pointer to the input vector.
cords[3]	The point at which the vector resides.
type	The type of coordinate system in which the vector is located.
	0 For a Cartesian system.
	1 For a cylindrical system.
	2 For a spherical system.
origin[3]	The origin of the coordinate system.
axis[3][3]	The axes of the coordinate system.
outputptr	The pointer to the output vector.

Returns Nothing.

HMIN_transform_vectortoglobal()

Allows vectors to be transformed to a global coordinate system, based on a local coordinate system.

Syntax void HMIN_transform_vectortoglobal(HM_vectorpointer inputptr, double axis[3][3], HM_vectorpointer outputptr);

inputptr	The pointer to the input data.
axis[3][3]	The set of axes of the vector.
outputptr	A pointer to the vector output.

Returns Nothing.

HMIN_transform_write()

Allows systems, loads, or nodes to be transformed to the global coordinate system, based on the local coordinate system.

Syntax void HMIN_transform_write(HM_entitytype type, HM_entityidtype id, HM_entityidtype systemid);

type	The type of entity to be transformed, systems, loads, or nodes.
id	The ID of the type.
systemid	The ID of the system into which the entities are transformed.

Returns Nothing.

Comments This function allows entities defined in a local coordinate system to be passed to HyperMesh. HyperMesh can then do transformations on those entities in the global coordinate system.

Curves

NOTE Once a curve is created, no other `hminlib` functions may be used until all curve data is processed with HMIN_curve_writepoint(), HMIN_curve_writesources(), or HMIN_writeattribute calls for a given curve.

`HMIN_curve_write()`

`HMIN_curve_writepoint()`

`HMIN_curve_writesources()`

HMIN_curve_write()

Writes a curve to HyperMesh.

Syntax void HMIN_curve_write(HM_entityidtype id, char * name, char * title, int linetype, int markertype, int color, int width, double scalex, double scaley);

id The ID of the curve to be written.

name The name of the curve to be written.

title The title associated with the curve.

linetype The type of line used for the curve:

1 Solid

2 Dotted

3 Dashes

4 Dash-dot

5 Long dashes

markertype The type of marker to be used. Set to 1 for circles, 2 for triangles, or 3 for boxes.

color The color to be used for the curve.

width The desired width of the curve.

scalex The x scale of the curve.

scaley The y scale of the curve.

Returns Nothing.

Comments After using `HMIN_curve_write()` to supply information regarding the curve, use `HMIN_curve_writepoint()` to supply the points of the curve to be transferred.

HMIN_curve_writepoint()

Writes the points of the curve to HyperMesh.

Syntax void HMIN_curve_writepoint(double point1, double point2);

point1 The x coordinate of the point to be passed.

point2 The y coordinate of the point to be passed.

Returns Nothing.

Comments The function `HMIN_curve_write()`, must be used before `HMIN_curve_writepoint()`. HyperMesh is then ready to make the curve with the points supplied in `HMIN_curve_writepoint()`.

HMIN_curve_writesources()

Writes a curve to HyperMesh.

Syntax void HMIN_curve_writesources(int kind1, char *path1, char *type1, char *req1, char *comp1, int kind2, char *path2, char *type2, char *req2, char *comp2);

kind1 Type of source for x data.

Set to 0, if the source is from a file.
Set to 1, if the source is from a math expression.

path1 File path and name, or math expression for x data.

If *kind1* = 0, this indicates the path of the file.
If *kind1* = 1, this includes the math expression (see the *Math Functions* chapter for math expression syntax).

type1 The type of the x component (may be set to "").

If *kind1* = 1, this must be set to "".

req1 The name of the x request (may be set to "").

If *kind1* = 1, this must be set to "".

comp1 The name of the x request component (may be set to "").

If *kind1* = 1, this must be set to "".

kind2 The type of source for y data.

Set to 0, if the source is from a file.
Set to 1, if the source is from a math expression.

path2 File path and name, or math expression for y data.

If *kind2*= 0, this indicates the path of the file.
If *kind2*= 1, this includes the math expression (see the *Curve Mathematics* book for math expression syntax).

type2 The type of the y component (may be set to "").

If *kind2* = 1, this must be set to "".

req2 The name of the y request (may be set to "").

If *kind2* = 1, must be set to "".

comp2 The name of the y request component (may be set to "").

If *kind2* = 1, this must be set to "".

Returns Nothing.

Comments Curve sources are used to define x-y curves based upon data files that can be read by the HyperMesh curve reader or upon math expressions (see the *Curve Mathematics* book). The type, request, and component refer to the hierarchy used to describe data where type is at the uppermost level and the component is at the lowest level. A file that contains displacements for nodes may be described as:

Type = displacements.

Request = Node ID for the given type (displacements).

Component = x, y, or z displacement value for the request.

Dictionaries

`HMIN_dictionary_write()`

HMIN_dictionary_write()

Transfers a dictionary into a collector.

Syntax void HMIN_dictionary_write(int entities, HM_entityidtype entityid, char * name, char * typename, char * string, double value, int active);

entities The type of entity (HM_ENTITYTYPE_COMPS, HM_ENTITYTYPE_LOADCOLS, HM_ENTITYTYPE_SYSTCOLS, HM_ENTITYTYPE_GROUPS, HM_ENTITYTYPE_MATS).

entityid The entity ID that should be assigned to the dictionary.

name The name of the dictionary item.

typename A name assigned to a character string that is "none," "real," "integer," or "string."

string A character string associated with the dictionary.

value The value associated with the dictionary.

active The activity status associated with the dictionary item. Activity can be set to:

-1 If the dictionary is not selectable.

0 If the dictionary is selectable, but is set to off.

1 If the dictionary is selectable, but is set to on.

See the *Template* Chapter of the *HyperMesh Reference Manual*.

Returns Nothing.

Elements

HMIN_element_getnodesfromconfig()

HMIN_element_writebar_vector()

HMIN_element_writebar3_vector()

HMIN_element_writebar_ynode()

HMIN_element_writebar3_ynode()

HMIN_element_writebar_znode()

HMIN_element_writebar3_znode()

HMIN_element_writegap()

HMIN_element_writegroup_shellid

HMIN_element_writegroup_solidid

HMIN_element_writehex8()

HMIN_element_writehex20()

HMIN_element_writejoint()

HMIN_element_writemass()

HMIN_element_writemaster3()

HMIN_element_writemaster4()

HMIN_element_writepenta6()

HMIN_element_writepenta15()

HMIN_element_writeplot()

HMIN_element_writequad4()

HMIN_element_writequad8()

HMIN_element_writerbe3()

HMIN_element_writerbe3header()

HMIN_element_writerbe3node()

HMIN_element_writerigid()

HMIN_element_writerigidlink()

HMIN_element_writerigidlinkheader()

HMIN_element_writerigidlinknode()

HMIN_element_writerod()

HMIN_element_writeslave1()

HMIN_element_writeslave3()

HMIN_element_writeslave4()

HMIN_element_writeslave_face()

HMIN_element_writespring()

HMIN_element_writetetra4()

HMIN_element_writetetra10()

HMIN_element_writetria3()

HMIN_element_writetria6()

HMIN_element_writeweld()

HMIN_element_getnodesfromconfig()

Determines the number of nodes associated with the configuration of the element.

Syntax int HMIN_element_getnodesfromconfig(int config);

config	The configuration of the element (HM_ELEMENT_CONFIG_TRIA3, HM_ELEMENT_CONFIG_QUAD4). For more element types, refer to the include file hmlib.h.

Returns The number of nodes associated with the element. For rigid links and RBE3s, zero is returned because the number of nodes may vary.

HMIN_element_writebar_vector()

Writes a 2-noded bar to HyperMesh using a vector to define the bar local x axis.

Syntax void HMIN_element_writebar_vector(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, double vector[3], double offseta[3], double offsetb[3], int pinsa, int pinsb, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
propertyid	The ID of the property to which the element should point.
vector[3]	The node at end a or end b of the desired vector.
offseta[3]	The amount to offset the bar at end a of the vector.
offsetb[3]	The amount to offset the bar at end b of the vector.
pinsa	The degrees of freedom assigned to end a of the vector.
pinsb	The degrees of freedom assigned to end b of the vector.
nodes[2]	The nodes associated with the element.
collectorid	The ID of the collector where the vector element should be placed.

Returns Nothing.

HMIN_element_writebar3_vector()

Writes a 3-noded bar to HyperMesh using a vector to define the bar local x axis.

Syntax void HMIN_element_writebar_vector(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, double vector[3], double offseta[3], double offsetb[3], int pinsa, int pinsb, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
propertyid	The ID of the property to which the element should point.
vector[3]	The node at end a or end b of the desired vector.
offseta[3]	The amount to offset the bar at end a of the vector.
offsetb[3]	The amount to offset the bar at end b of the vector.
pinsa	The degrees of freedom assigned to end a of the vector.
pinsb	The degrees of freedom assigned to end b of the vector.
nodes[3]	The nodes associated with the element.
collectorid	The ID of the collector where the vector element should be placed.

Returns Nothing.

HMIN_element_writebar_ynode()

Writes a 2-noded bar element to HyperMesh using a node to define the local bar y axis.

Syntax void HMIN_element_writebar_ynode(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, HM_entityidtype node, double offseta[3], double offsetb[3], int pinsa, int pinsb, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
propertyid	The ID of the property to which the bar element should point.
node	The node that lies on the beam local y axis.
offseta[3]	The amount to offset the bar at end a.
offsetb[3]	The amount to offset the bar at end b.
pinsa	The degrees of freedom assigned to end a.
pinsb	The degrees of freedom assigned to end b.
nodes[2]	The nodes associated with the element.
collectorid	The ID of the collector where the bar element should be placed.

Returns Nothing.

HMIN_element_writebar3_ynode()

Writes a 3-noded bar element to HyperMesh using a node to define the local bar y axis.

Syntax void HMIN_element_writebar_ynode(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, HM_entityidtype node, double offseta[3], double offsetb[3], int pinsa, int pinsb, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
propertyid	The ID of the property to which the bar element should point.
node	The node that lies on the beam local y axis.
offseta[3]	The amount to offset the bar at end a.
offsetb[3]	The amount to offset the bar at end b.
pinsa	The degrees of freedom assigned to end a.
pinsb	The degrees of freedom assigned to end b.
nodes[3]	The nodes associated with the element.
collectorid	The ID of the collector where the bar element should be placed.

Returns Nothing.

HMIN_element_writebar_znode()

Writes a bar element to HyperMesh using a node to define the local bar z axis.

Syntax void HMIN_element_writebar_znode(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, HM_entityidtype node, double offseta[3], double offsetb[3], int pinsa, int pinsb, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
propertyid	The ID of the property to which the bar element should point.
node	The node that lies on the beam local z axis.
offseta[3]	The amount to offset the bar at end a.
offsetb[3]	The amount to offset the bar at end b.
pinsa	The degrees of freedom assigned to end a.
pinsb	The degrees of freedom assigned to end b.
nodes[2]	The nodes associated with the element.
collectorid	The ID of the collector where the bar element should be placed.

Returns Nothing.

HMIN_element_writebar3_znode()

Writes a bar element to HyperMesh using a node to define the local bar z axis.

Syntax void HMIN_element_writebar_znode(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, HM_entityidtype node, double offseta[3], double offsetb[3], int pinsa, int pinsb, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
propertyid	The ID of the property to which the bar element should point.
node	The node that lies on the beam local z axis.
offseta[3]	The amount to offset the bar at end a.
offsetb[3]	The amount to offset the bar at end b.
pinsa	The degrees of freedom assigned to end a.
pinsb	The degrees of freedom assigned to end b.
nodes[3]	The nodes associated with the element.
collectorid	The ID of the collector where the bar element should be placed.

Returns Nothing.

HMIN_element_writegap()

Writes a gap element to HyperMesh.

Syntax void HMIN_element_writegap(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, HM_entityidtype nodes[], int collectorid, HM_entityidtype vectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
propertyid	The ID of the property to which the gap element should point.
nodes[2]	The nodes associated with the element.
collectorid	The ID of the collector where the gap element should be placed.
vectorid	The ID of the vector associated with the element.

Returns Nothing.

HMIN_element_writegroup_shellid

Creates a master element from an existing element in HyperMesh.

Syntax void HMIN_element_writemaster_shellid(HM_entityidtype id, char elementtype, HM_entityidtype elementid, int shellid, HM_entityidtype collectorid);

id The ID of the new element.

elementtype The type of the new element, a user-defined value.

elementid The ID of the existing element.

shellid The shellid number. For solids, the shellid number can be 1 - 6. For 2-D elements, use:

1 If the normals of the master element should be created in the same direction as those in the existing element.

2 If the normals of the master element should be created in the opposite direction of those in the existing element.

collectorid The ID of the collector where the master element should be placed.

Returns Nothing.

HMIN_element_writegroup_solidid

Creates a master element from an existing element in HyperMesh.

Syntax void HMIN_element_writemaster_solidid(HM_entityidtype id, char elementtype, HM_entityidtype elementid, int solidid, HM_entityidtype collectorid);

id The ID of the new element.

elementtype The type of the new element, a user-defined value.

elementid The ID of the existing element.

solidid The solidid number. For solids, the solidid number can be 1 - 6. For 2-D elements, use:

1 If the normals of the master element should be created in the same direction as those in the existing element.

2 If the normals of the master element should be created in the opposite direction of those in the existing element.

collectorid The ID of the collector where the master element should be placed.

Returns Nothing.

HMIN_element_writehex8()

Writes an eight-noded brick solid element to HyperMesh.

Syntax void HMIN_element_writehex8(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[8]	The nodes associated with the element.
collectorid	The ID of the collector where the hex8 element should be placed.

Returns Nothing.

HMIN_element_writehex20()

Writes a twenty-noded brick solid element to HyperMesh.

Syntax void HMIN_element_writehex20(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[20]	The nodes associated with the element.
collectorid	The ID of the collector where the hex20 element should be placed.

Returns Nothing.

HMIN_element_writejoint()

Writes a joint element to HyperMesh

Syntax HMIN_element_writejoint(HM_entityidtype id, char elementtype, HM_entityidtype propertyid,HM_entityidtype nodes[6], int orientation, HM_entityidtype orientationids[2], HM_entityidtype collectorid)

id The ID of the element.

type The type of the element, according to the following:

Type	Type Name	# Nodes	Orientation
1	Spherical	2	none/systems/nodes
2	Revolute	4	none/systems
3	Cylindrical	4	none/systems
4	Planar	4	none/systems
5	Universal	4	none/systems
6	Translational	6	none/systems
7	Locking	6	none/systems

propertyid The ID of the property collector to which the joint is associated.

nodes[6] Six node IDs associated with the element. Six are required, regardless of type. Use 0 (zero) for unused nodes.

orientation Use 0 for none, 1, for systems, and 2, for nodes.

orientationids[2] Node or system IDs used to orient the joint.

collectorid The ID of the vector collector.

Returns Nothing.

HMIN_element_writemass()

Writes a mass element to the HyperMesh database.

Syntax void HMIN_element_writemass(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, double mass, HM_entityidtype systemid, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of mass, a user-defined value.
propertyid	The ID of the property assigned to the mass element.
mass	The mass of the element.
systemid	The system ID of the mass.
nodes[1]	The nodes associated with the element.
collectorid	The ID of the collector where the mass should be placed.

Returns Nothing.

HMIN_element_writemaster3()

Writes a three-noded master interface element to HyperMesh.

Syntax void HMIN_element_writemaster3(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[3]	The nodes associated with the element.
collectorid	The ID of the collector where the master3 element should be placed.

Returns Nothing.

HMIN_element_writemaster4()

Writes a four-noded master interface element to HyperMesh.

Syntax void HMIN_element_writemaster4(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[4]	The nodes associated with the element.
collectorid	The ID of the collector where the master4 element should be placed.

Returns Nothing.

HMIN_elementwritepenta6()

Writes a six-noded wedge solid element to HyperMesh.

Syntax void HMIN_element_writepenta6(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[6]	The nodes associated with the element.
collectorid	The ID of the collector where the penta6 element should be placed.

Returns Nothing.

HMIN_element_writepenta15()

Writes a fifteen-noded wedge solid element to HyperMesh.

Syntax void HMIN_element_writepenta15(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[15]	The nodes associated with the element.
collectorid	The ID of the collector where the penta15 element should be placed.

Returns Nothing.

HMIN_element_writeplot()

Writes a plot element to HyperMesh.

Syntax void HMIN_element_writeplot(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of plot, a user-defined value.
nodes[2]	The nodes associated with the element.
collectorid	The ID of the collector where the plot should be placed.

Returns Nothing.

HMIN_element_writequad4()

Writes a four-noded quadrilateral shell element to HyperMesh.

Syntax void HMIN_element_writequad4(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[4]	The nodes associated with the element.
collectorid	The ID of the collector where the quad4 element should be placed.

Returns Nothing.

HMIN_element_writequad8()

Writes an eight-noded quadrilateral shell element to HyperMesh.

Syntax void HMIN_element_writequad8(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[8]	The nodes associated with the element.
collectorid	The ID of the collector where the quad8 element should be placed.

Returns Nothing.

HMIN_element_writerbe3()

Writes an RBE3 element to HyperMesh.

Syntax void HMIN_element_writerrbe3(HM_entityidtype id, char elementtype, int Inode_len, HM_entityidtype *Inodes, int *Idofs, double *Icoeffs, HM_entityidtype Dnode, int Ddof, double Dcoeff, HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
Inode_len	The number of independent nodes.
Inodes	The IDs of the independent nodes.
Idofs	The degrees of freedom associated with the independent nodes.
Icoeffs	The coefficients associated with the independent nodes.
Dnode	The ID of the dependent node.
Ddof	The degrees of freedom associated with the dependent node (may be set to 0).
Dcoeff	The coefficients associated with the dependent node (may be set to 0).
collectorid	The collector where the RBE3 element should be placed.

Returns Nothing.

HMIN_element_writerbe3header()

Writes an RBE3 element header to HyperMesh.

Syntax void HMIN_element_writerbe3header(HM_entityidtype id, char elementtype, HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the elemlent, a user-defined value.
collectorid	The collector where the rbe3 element should be placed.

Returns Nothing.

Comments `HMIN_element_writerbe3node()` should be called immediately after `HMIN_element_writerbe3header()` to add nodes to the element.

HMIN_element_writerbe3node()

Writes the nodes that define the RBE3 element to HyperMesh.

Syntax void HMIN_element_writerbe3node(HM_entityidtype nodeid, double coefficient, int dofs, int dependent_flag);

nodeid	The ID of the node to add to the RBE3 element.
coefficient	The coefficient to be associated with the node (may be set to 0).
dofs	The degrees of freedom to be associated with the node (may be set to 0).
dependent_flag	Set to 0 to define the independent node, and 1 to define the dependent nodes.

Returns Nothing.

Comments `HMIN_element_writerbe3header()` must be called immediately before `HMIN_element_writerbe3node()`. Only one node may be defined as the dependent node. No other `hminlib` commands may be used until *all* RBE3 data is processed for the given element.

HMIN_element_writerigid()

Writes a rigid element to HyperMesh.

Syntax void HMIN_element_writerigid(HM_entityidtype id, char elementtype, int dofs, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
dofs	The degrees of freedom that apply to the element (123, 456, 123456).
nodes[2]	The nodes associated with the element.
collectorid	The ID of the collector where the rigid elements should be placed.

Returns Nothing.

HMIN_element_writerigidlink()

Writes a rigid link element to HyperMesh.

Syntax void HMIN_element_writerigidlink(HM_entityidtype id, char elementtype, HM_entityidtype Inode, int dofs, int Dnode_len, HM_entityidtype *Dnodes, HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
Inode	The ID of the independent node.
dofs	The degrees of freedom associated with the element.
Dnode_len	The number of dependent nodes.
Dnodes	An array of dependent node IDs.

Returns Nothing.

HMIN_element_writerigidlinkheader()

Writes a rigid link header to HyperMesh.

Syntax void HMIN_element_writerigidlinkheader(HM_entityidtype id, char elementtype, int dofs, HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
dofs	The degrees of freedom associated with the element.
collectorid	The ID of the collector where the rigid link element should be placed.

Returns Nothing.

Comments Call `HMIN_element_writerigidlinknode()` immediately after `HMIN_element_writerigidlinkheader()` to add the independent node and dependent nodes to the element.

HMIN_element_writerigidlinknode()

Writes the nodes that define the rigid link to HyperMesh.

Syntax void HMIN_element_writerigidlinknode(HM_entityidtype nodeid, int dependent_flag);

nodeid The ID of the node to add to the rigid link element.

dependent_flag Set to 0 to define an independent node and 1 to define a dependent node.

Returns Nothing.

Comments The function `HMIN_element_writerigidlinkheader()` must be called immediately before any nodes can be defined with `HMIN_element_writerigidlinknode()`. Only one node may be defined as the independent node. No other `hminlib` comments may be used until *all* rigid link nodes are processed for the given element.

HMIN_element_writerod()

Writes a rod element to HyperMesh.

Syntax void HMIN_element_writerod(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, HM_entityidtype nodes[], HM_entityidtype collectorid);

id The ID of the element.

elementtype The type of the element, a user-defined value.

propertyid The ID of the property to which the rod element should point.

nodes[2] The nodes associated with the element.

collectorid The ID of the collector where the rod element should be placed.

Returns Nothing.

HMIN_element_writeslave1()

Writes a one-noded slave interface element to HyperMesh.

Syntax void HMIN_element_writeslave1(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id The ID of the element.

elementtype The type of the element, a user-defined value.

nodes[1] The nodes associated with the element.

collectorid The ID of the collector where the slave1 element should be placed.

Returns Nothing.

HMIN_element_writeslave3()

Writes a three-noded slave interface element to HyperMesh.

Syntax void HMIN_element_writeslave3(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[3]	The nodes associated with the element.
collectorid	The ID of the collector where the slave3 element should be placed.

Returns Nothing.

HMIN_element_writeslave4()

Writes a four-noded slave interface element to HyperMesh.

Syntax void HMIN_element_writeslave4(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[4]	The nodes associated with the element.
collectorid	The ID of the collector where the slave4 element should be placed.

Returns Nothing.

HMIN_element_writeslave_face()

Creates a slave element from an existing element in HyperMesh.

Syntax void HMIN_element_writeslave_face(HM_entityidtype id, char elementtype, HM_entityidtype elementid, int face, HM_entityidtype collectorid);

id The ID of the new element.

elementtype The type of the new element, a user-defined value.

elementid The ID of the existing element.

face The face number. For solids, the face number can be 1 - 6. For 2-D elements, use:

1 If the normals of the slave element should be created in the same direction as those in the existing element.

2 If the normals of the slave element should be created in the opposite direction of those in the existing element.

collectorid The ID of the collector where the slave element should be placed.

Returns Nothing.

HMIN_element_writespring()

Writes a spring element to HyperMesh.

Syntax void HMIN_element_writespring(HM_entityidtype id, char elementtype, HM_entityidtype propertyid, int dof, HM_entityidtype nodes[], HM_entityidtype collectorid, HM_entityidtype vectorid);

id The ID of the element.

elementtype The type of the element, a user-defined value.

propertyid The ID of the property to which the spring element should point.

dof The degree of freedom associated with the element.

nodes[2] The nodes associated with the element.

collectorid The ID of the collector where the spring element should be placed.

vectorid The ID of the vector associated with the element.

Returns Nothing.

HMIN_element_writetetra4()

Writes a four-noded tetrahedral solid element to HyperMesh.

Syntax void HMIN_element_writetetra4(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[4]	The nodes associated with the element.
collectorid	The ID of the collector where the tetra4 element should be placed.

Returns Nothing.

HMIN_element_writetetra10()

Writes a ten-noded tetrahedral solid element to HyperMesh.

Syntax void HMIN_element_writetetra10(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[10]	The nodes associated with the element.
collectorid	The ID of the collector where the tetra10 element should be placed.

Returns Nothing.

HMIN_element_writetria3()

Writes a three-noded triangular shell element to HyperMesh.

Syntax void HMIN_element_writetria3(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], int collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[3]	The nodes associated with the element.
collectorid	The ID of the collector where the tria3 element should be placed.

Returns Nothing.

HMIN_element_writetria6()

Writes a six-noded triangular shell element to HyperMesh.

Syntax void HMIN_element_writetria6(HM_entityidtype id, char elementtype, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
nodes[6]	The nodes associated with the element.
collectorid	The ID of the collector where the tria6 element should be placed.

Returns Nothing.

HMIN_element_writeweld()

Writes a weld element to HyperMesh.

Syntax void HMIN_element_writeweld(HM_entityidtype id, char elementtype, int dofs, HM_entityidtype nodes[], HM_entityidtype collectorid);

id	The ID of the element.
elementtype	The type of the element, a user-defined value.
dofs	The degrees of freedom that apply to the element.
nodes[2]	The nodes associated with the element.
collectorid	The ID of the collector where the weld should be placed.

Returns Nothing.

Equations

`HMIN_equation_write()`

HMIN_equation_write()

Writes an equation to HyperMesh.

Syntax void HMIN_equation_write(HM_entityidtype id, char type, float* constant, int Inode_len, HM_entityidtype *Inodes, int *Idofs, float *Icoeffs, HM_entityidtype Dnode, int Ddof, float* Dcoeff, HM_entityidtype loadcollectorid);

id	The ID of the equation.
equationtype	The type of the equation, a user-defined value.
constant	The constant of the equation.
Inode_len	The number of independent nodes.
Inodes	The IDs of the independent nodes.
Idofs	The degrees of freedom associated with the independent nodes in the form (123456 or 135).
Icoeffs	The coefficients associated with the independent nodes (6 per node).
Dnode	The ID of the dependent node.
Ddof	The degree of freedom associated with the dependent node (may be set to 0).
Dcoeff	The coefficient associated with the dependent node (may be set to 0).
loadcollectorid	The collector where the equation should be placed.

Returns Nothing.

Groups

`HMIN_group_write()`

`HMIN_group_writemasterbox()`

`HMIN_group_writemastercomponent()`

`HMIN_group_writemasterdefinition()`

`HMIN_group_writemasterset()`

`HMIN_group_writeslavebox()`

`HMIN_group_writeslavecomponent()`

`HMIN_group_writeslavedefinition()`

`HMIN_group_writeslaveset()`

HMIN_group_write()

Writes groups to HyperMesh.

Syntax void HMIN_group_write(HM_entityidtype id, char * name, int config, int type, HM_entityidtype materialid, int color, int masterdefinition, int slavedefinition);

id	The ID of the group.
name	The name of the group.
config	The configuration assigned to the group.
type	The type assigned to the group.
materialid	The ID of the material associated with the group.
color	The color assigned to the group.
masterdefinition	The method being used to define the master surface. Set to 0 for elements, 1 for components, 2 for box, 3 for all, or 4 for entity sets.
slavedefinition	The method being used to define the slave surface. Set to 0 for elements, 1 for components, 2 for box, 3 for all, or 4 for entity sets.

Returns Nothing.

HMIN_group_writemasterbox()

Sends the size of the master box of a group to HyperMesh.

Syntax void HMIN_group_writemasterbox(HM_entityidtype id, double xminimum, double yminimum, double zminimum, double xmaximum, double ymaximum, double zmaximum);

id	The ID of the group.
xminimum	The minimum x value for the box.
yminimum	The minimum y value for the box.
zminimum	The minimum z value for the box.
xmaximum	The maximum x value for the box.
ymaximum	The maximum y value for the box.
zmaximum	The maximum z value for the box.

Returns Nothing.

HMIN_group_writemastercomponent()

Sends a component ID to a group's master component list.

Syntax void HMIN_group_writemastercomponent(HM_entityidtype id, HM_entityidtype componentid);

id	The ID of the group.
componentid	The ID of the component that should be part of the master component list.

Returns Nothing.

HMIN_group_writemasterdefinition()

Writes the master definition of a group to HyperMesh.

Syntax void HMIN_group_writemasterdefinition(HM_entityidtype id, int definition);

id	The ID of the group.
definition	The method being used to define the master surface. Set to 0 for elements, 1 for components, 2 for box, 3 for all, or 4 for entity sets.

Returns Nothing.

HMIN_group_writemasterset()

Sends a set’s ID to a group's master set list.

Syntax void HMIN_group_writemasterset(HM_entityidtype id, HM_entityidtype masterid);

id	The ID of the group.
masterid	The ID of the set that should be part of the master set list.

Returns Nothing.

HMIN_group_writeslavebox()

Sends the size of a slave box of a group to HyperMesh.

Syntax void HMIN_group_writeslavebox(HM_entityidtype id, double xminimum, double yminimum, double zminimum, double xmaximum, double ymaximum, double zmaximum);

id	The ID of the group.
xminimum	The minimum x value for the box.
yminimum	The minimum y value for the box.
zminimum	The minimum z value for the box.
xmaximum	The maximum x value for the box.
ymaximum	The maximum y value for the box.
zmaximum	The maximum z value for the box.

Returns Nothing.

HMIN_group_writeslavecomponent()

Sends a component ID to a group's slave component list.

Syntax void HMIN_group_writeslavecomponent(HM_entityidtype id, HM_entityidtype componentid);

id	The ID of the group.
componentid	The ID of the component that should be part of the slave component list.

Returns Nothing.

HMIN_group_writeslavedefinition()

Writes the slave definition of a group to HyperMesh.

Syntax void HMIN_group_writeslavedefinition(HM_entityidtype id, int definition);

id	The ID of the group.
definition	The method being used to define the slave surface. Set to 0 for elements, 1 for components, 2 for box, 3 for all, or 4 for entity sets.

Returns Nothing.

HMIN_group_writeslaveset()

Sends a set’s ID to a group's slave set list.

Syntax void HMIN_group_writeslaveset(HM_entityidtype id, HM_entityidtype setid);

id	The ID of the group.
settid	The ID of the set that should be part of the slave set list.

Returns Nothing.

Lines

The line transfer commands, except `HMIN_line_write()`, are valid only after a call to `HMIN_startline()` and before the corresponding call to `HMIN_endline()`.

`HMIN_line_write()`

`HMIN_startline()`

`HMIN_endline()`

`HMIN_writecubic()`

`HMIN_writeellipse()`

`HMIN_writelinesat()`

`HMIN_writeNURBS()`

`HMIN_writestraight()`

HMIN_line_write()

Writes a line to HyperMesh.

Syntax void HMIN_line_write(HM_entityidtype id, double geometric[3][4], HM_entityidtype componentid);

id The ID of the line.

geometric[3][4] The geometric coordinates of the line. The number in the first set of brackets indicates which coordinate is given. Assign:

0 For the x coordinate.

1 For the y coordinate.

2 For the z coordinate.

The number in the second set of brackets indicates which point the given coordinate is. Assign:

0 For the starting point.

1 For the ending point.

2 For the start tangent.

3 For the end tangent.

componentid The ID of the component.

Returns Nothing.

HMIN_startline()

Signals the beginning of a line.

Syntax void HMIN_startline(HM_entityidtype id, double tolerance, int closed, HM_entityidtype componentid);

id The ID of the line.

tolerance The tolerance to which you have created the line. Two points less than this distance apart are considered the same point.

closed If true, the line is closed, i.e., the start point is the same as the end point.

componentid The ID of the collector where the line should be placed.

Returns Nothing.

Comments In HyperMesh, lines are always connected. Thus, if the end point of one line is more than tolerance away from the start point of the next, HyperMesh arbitrarily modifies the incoming line to make it connected. Currently, straight segments are added to connect the points. This is subject to change in future versions, and

should not be relied on.

Segments must be at least *tolerance* long. HyperMesh arbitrarily changes segments that are shorter than tolerance.

Self-intersecting lines often confuse HyperMesh; do not use them.

HMIN_endline()

Signals the end of a line.

Syntax void HMIN_endline(void);

Returns Nothing.

HMIN_writecubic()

Specifies the next segment in the current line as a piecewise cubic spline.

Syntax void HMIN_writecubic(int number, double data);

number The number of pieces in the cubic spline.

data An array of line data in the form: x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4,... continue with the x1, y1, z1 through x4, y4, z4 data for each piece in the cubic spline.

Returns Nothing.

Comments t goes from 0 to 1 for each piece:

x(t) = x1 + x2 * t + x3 * t2 + x4 * t3
y(t) = y1 + y2 * t + y3 * t2 + y4 * t3
z(t) = z1 + z2 * t + z3 * t2 + z4 * t3

This command is only valid after a call to `HMIN_startline()` and before the corresponding call to `HMIN_endline()`.

HMIN_writeellipse()

Specifies the next segment in the current line as an elliptical arc.

Syntax void HMIN_writeellipse(double cx, double cy, double cz, double nx, double ny, double nz, double ax, double ay, double az, double ratio, double sx, double sy, double sz, double ex, double ey, double ez);

cx	The x coordinate of the center of the arc.
cy	The y coordinate of the center of the arc.
cz	The z coordinate of the center of the arc.
nx	The x coordinate of the vector that is the normal of the arc.
ny	The y coordinate of the vector that is the normal of the arc.
nz	The z coordinate of the vector that is the normal of the arc.
ax	The x coordinate of the vector that is the major axis of the arc.
ay	The y coordinate of the vector that is the major axis of the arc.
az	The z coordinate of the vector that is the major axis of the arc.
ratio	The ratio of the length of the minor axis over the length of the major axis.
sx	The x coordinate of the start of the arc.
sy	The y coordinate of the start of the arc.
sz	The z coordinate of the start of the arc.
ex	The x coordinate of the end of the arc.
ey	The y coordinate of the end of the arc.
ez	The z coordinate of the end of the arc.

Returns Nothing.

Comments The direction of the arc is specified by the normal and major axes using the right-hand rule (negating the normal while leaving the remainder of the arc unchanged gives you the complement of the original arc.)

This command is only valid after a call to `HMIN_startline()` and before the corresponding call to `HMIN_endline()`.

HMIN_writelinesat()

Writes the location of a line in SAT format within a file.

Syntax void HMIN_writelinesat(char * filename, long pos, int prefix_len, HM_entityidtype id, HM_entityidtype componentid);

filename	The name of the file that contains the SAT line.
pos	The offset into the file.
prefix_len	The number of characters to ignore at the beginning of each line in the file.
id	The ID of the line.
componentid	The component ID into which the line should be placed.

Returns Nothing.

HMIN_writeNURBS()

Specifies the next segment in the current line as a NURBS curve.

Syntax void HMIN_writeNURBS(unsigned int degree, unsigned int number_points, double * points, double * weights, unsigned int number_knots, double * knots, double knot_tolerance, double start_t, double end_t);

degree	The degree of the NURBS.
number_points	The number of control points in the NURBS.
points	Those control points, in the form X1, Y1, Z1, X2, Y2, Z2, and so on.
weights	If the NURBS is rational, this is the weight given to each control point. If the NURBS is polynomial, a null pointer should be passed in.
number_knots	The number of knots in the NURBS's controlling knot sequence.
knots	The knot sequence.
knot_tolerance	The tolerance that determines if two knots are the same.
start_t	The parameter value at the start of the portion of the curve under consideration.
end_t	The parameter value at the end of the portion of the curve under consideration.

Returns Nothing.

Comments Refer to the *HyperMesh User's Manual* for a detailed description of the structure and meaning of NURBS curves.

The NURBS format is very versatile. HyperMesh attempts to understand all NURBS curves, but you can ensure more accurate results and faster processing time by following these guidelines:

1. HyperMesh requires that its lines be C1 arc length continuous. Thus multiple control points or knots of multiplicity of the degree or greater force HyperMesh to modify the line. You can avoid these problems by breaking the single NURBS into two or more NURBS; this, in fact, is how HyperMesh handles it.

2. The above caveat implies that NURBS of degree 1 are always problematical. This is certainly the case; avoid them if possible.

This command is only valid after a call to `HMIN_startline()` and before the corresponding call to `HMIN_endline()`.

HMIN_writestraight()

Specifies the next segment in the current line as a straight line segment.

Syntax void HMIN_writestraight(double x1, double y1, double z1, double x2, double y2, double z2);

x1	The x coordinate of the start of the line.
y1	The y coordinate of the start of the line.
z1	The z coordinate of the start of the line.
x2	The x coordinate of the end of the line.
y2	The y coordinate of the end of the line.
z2	The z coordinate of the end of the line.

Returns Nothing.

Comments This command is only valid after a call to `HMIN_startline()` and before the corresponding call to `HMIN_endline()`.

Load Collectors

HMIN_loadcollector_write()

HMIN_load_writeacceleration()

HMIN_load_writeconstraint()

HMIN_load_writeflux()

HMIN_load_writeforce()

HMIN_load_writemoment()

HMIN_load_writepressure_face()

HMIN_load_writepressure_nodes()

HMIN_load_writetemperature()

HMIN_load_writetraction_face()

HMIN_load_writetraction_nodes()

HMIN_load_writevelocity()

HMIN_material_write()

HMIN_loadstep_write()

HMIN_loadstep_writeid()

HMIN_loadcollector_write()

Writes a load collector to HyperMesh.

Syntax void HMIN_loadcollector_write(HM_entityidtype id, char * name, int color);

	id	The ID of the load collector.
	name	The name of the load collector.
	color	The color assigned to the load collector.

Returns Nothing.

HMIN_load_writeacceleration()

Writes an acceleration to `hmlib`.

Syntax void HMIN_load_writeacceleration(HM_entityidtype id, unsigned char type, HM_entityidtype nodeid, HM_entityidtype systemid, double xcomp, double ycomp, double zcomp, HM_entityidtype loadcollectorid);

id	The ID of the acceleration.
type	The type of the acceleration, a user-defined value.
nodeid	The ID of the node where the acceleration is applied.
systemid	The ID of the system in which the acceleration is applied.
xcomp	The x component of the acceleration.
ycomp	The y component of the acceleration.
zcomp	The z component of the acceleration.
loadcollectorid	The ID of the load collector to which the acceleration belongs.

Returns Nothing.

HMIN_load_writeconstraint()

Writes a constraint to `hmlib`.

Syntax void HMIN_load_writeconstraint(HM_entityidtype id, unsigned char type, HM_entityidtype nodeid, HM_entityidtype systemid, double components[6], HM_entityidtype loadcolid);

id	The ID of the constraint.
type	The type of the constraint, a user-defined value.
nodeid	The ID of the node where the constraint is applied.
systemid	The ID of the system in which the constraint is applied.
components[6]	The components of the constraints. If the node is to have no constraints, the value -999999.0 should be used.
loadcolid	The ID of the load collector.

Returns Nothing.

HMIN_load_writeflux()

Writes a flux to `hmlib`.

Syntax void HMIN_load_writeflux(HM_entityidtype id, unsigned char type, HM_entityidtype nodeid, HM_entityidtype systemid, double flux, HM_entityidtype loadcollectorid);

id	The ID of the flux.
type	The type of the flux, a user-defined value.
nodeid	The ID of the node where the flux is applied.
systemid	The ID of the system in which the flux is applied.
flux	The flux to be applied.
loadcollectorid	The ID of the collector in which the flux is applied.

Returns Nothing.

HMIN_load_writeforce()

Writes a force to `hmlib`.

Syntax void HMIN_load_writeforce(HM_entityidtype id, unsigned char type, HM_entityidtype nodeid, HM_entityidtype systemid, double xcomp, double ycomp, double zcomp, HM_entityidtype loadcollectorid);

id	The ID of the force.
type	The type of the force, a user-defined value.
nodeid	The ID of the node where the force is applied.
systemid	The ID of the system in which the force is applied.
xcomp	The x component of the force.
ycomp	The y component of the force.
zcomp	The z component of the force.
loadcollectorid	The ID of the load collector to which the force belongs.

Returns Nothing.

HMIN_load_writemoment()

Writes a moment to `hmlib`.

Syntax void HMIN_load_writemoment(HM_entityidtype id, unsigned char type, HM_entityidtype nodeid, HM_entityidtype systemid, double xcomp, double ycomp, double zcomp, HM_entityidtype loadcollectorid);

id	The ID of the moment.
type	The type of the moment, a user-defined value.
nodeid	The ID of the node where the moment is applied.
systemid	The ID of the system in which the moment is applied.
xcomp	The x component of the moment.
ycomp	The y component of the moment.
zcomp	The z component of the moment.
loadcollectorid	The ID of the load collector to which the moment belongs.

Returns Nothing.

HMIN_load_writepressure_face()

Writes the pressure applied to an element to `hmlib`, given the element face.

Syntax void HMIN_load_writepressure_face(HM_entityidtype id, unsigned char type, HM_entityidtype eid, HM_entityidtype face, HM_entityidtype systemid, double magnitude, HM_entityidtype loadcolid);

id	The ID of the pressure.
type	The type of the pressure, a user-defined value.
eid	The ID of the element to which the pressure should be applied.
face	The face of the element where the pressure should be applied.
systemid	The ID of the system in which the pressure should be applied.
magnitude	The magnitude of the pressure.
loadcolid	The ID of the load collector where the pressure should be placed.

Returns Nothing.

HMIN_load_writepressure_nodes()

Writes the pressure applied to an element to `hmlib`, given the nodes that define the face where the pressure should be applied.

Syntax void HMIN_load_writepressure_nodes(HM_entityidtype id, unsigned char type, HM_entityidtype eid, HM_entityidtype nodes[4], HM_entityidtype systemid, double magnitude, HM_entityidtype loadcolid);

id	The ID of the pressure.
type	The type of the pressure, a user-defined value.
eid	The ID of the element to which the pressure should be applied.
nodes[4]	The nodes that define the face where the pressure should be applied.
systemid	The ID of the system in which the pressure should be applied.
magnitude	The magnitude of the pressure.
loadcolid	The ID of the load collector where the pressure should be placed.

Returns Nothing.

HMIN_load_writetemperature()

Writes a temperature to `hmlib`.

Syntax void HMIN_load_writetemperature(HM_entityidtype id, unsigned char type, HM_entityidtype nodeid, HM_entityidtype systemid, double temperature, HM_entityidtype loadcollectorid);

id	The ID of the temperature.
type	The type of the temperature, a user-defined value.
nodeid	The ID of the node where the temperature is applied.
systemid	The ID of the system in which the temperature is applied.
temperature	The temperature to be applied.
loadcollectorid	The ID of the collector in which the temperature is applied.

Returns Nothing.

HMIN_load_writetraction_face()

Writes the tractive pressure applied to an element to `hmlib`, given the element face.

Syntax void HMIN_load_writetraction_face(HM_entityidtype id, unsigned char type, HM_entityidtype eid, HM_entityidtype face, HM_entityidtype systemid, double xcomp, double ycomp, double zcomp, double magnitude, HM_entityidtype loadcolid);

id	The ID of the tractive pressure.
type	The type of the tractive pressure, a user-defined value.
eid	The ID of the element in which the tractive pressure should be applied.
face	The face of the element where the tractive pressure should be applied.
systemid	The ID of the system in which the tractive pressure should be applied.
xcomp	The x component of the vector.
ycomp	The y component of the vector.
zcomp	The z component of the vector.
magnitude	The magnitude of the pressure.
loadcolid	The ID of the load collector where the tractive pressure should be placed.

Returns Nothing.

HMIN_load_writetraction_nodes()

Writes the tractive pressure applied to an element to `hmlib`, given the nodes that define the face where the pressure should be applied.

Syntax void HMIN_load_writetraction_nodes(HM_entityidtype id, unsigned char type, HM_entityidtype eid, HM_entityidtype nodes[4], HM_entityidtype systemid, double xcomp, double ycomp, double zcomp, double magnitude, HM_entityidtype loadcolid);

id	The ID of the tractive pressure.
type	The type of the tractive pressure, a user-defined value.
eid	The ID of the element in which the tractive pressure should be applied.
nodes[4]	The nodes that define the face where the tractive pressure should be applied.
systemid	The ID of the system in which the tractive pressure should be applied.
xcomp	The x component of the vector.
ycomp	The y component of the vector.
zcomp	The z component of the vector.
magnitude	The magnitude of the pressure.
loadcolid	The ID of the load collector where the tractive pressure should be placed.

Returns Nothing.

HMIN_load_writevelocity()

Writes a velocity to `hmlib`.

Syntax void HMIN_load_writevelocity(HM_entityidtype id, unsigned char type, HM_entityidtype nodeid, HM_entityidtype systemid, double xcomp, double ycomp, double zcomp, HM_entityidtype loadcollectorid);

id	The ID of the velocity.
type	The type of the velocity, a user-defined value.
nodeid	The ID of the node where the velocity is applied.
systemid	The ID of the system in which the velocity is applied.
xcomp	The x component of the velocity.
ycomp	The y component of the velocity.

zcomp The z component of the velocity.

loadcollectorid The ID of the load collector to which the velocity belongs.

Returns Nothing.

HMIN_material_write()

Writes a material to HyperMesh.

Syntax void HMIN_material_write(HM_entityidtype id, char * name);

id The ID of the material.

name The name of the material.

Returns Nothing.

HMIN_loadstep_write()

Writes a loadstep to HyperMesh.

Syntax void HMIN_loadstep_write(HM_entityidtype id, char * name);

id The ID of the loadstep.

name The name of the loadstep.

Returns Nothing.

Comments `HMIN_loadstep_writeid()` must be called after using `HMIN_loadstep_write()` to put individual load collectors into the loadstep.

HMIN_loadstep_writeid()

Writes the load collector into the specified loadstep.

Syntax void HMIN_loadstep_writeid(HM_entityidtype loadcollector);

loadcollector The ID of the load collector to be included in the loadstep.

Returns Nothing.

Comments `HMIN_loadstep_write()` must be called before using `HMIN_loadstep_writeid()` to put individual load collectors into the loadstep.

Nodes

`HMIN_associatenode()`

`HMIN_node_flush()`

`HMIN_node_free()`

`HMIN_node_getfirstpointer()`

`HMIN_node_getnextpointer()`

`HMIN_node_getpointer()`

`HMIN_node_store()`

`HMIN_node_write()`

`HMIN_node_writepointer()`

`HMIN_outputblock_write()`

`HMIN_outputblock_writeid()`

HMIN_associatenode()

Associates a node to a surface.

Syntax void HMIN_associatenode(HM_entityidtype id, HM_entityidtype surfid, int surfaceindex);

id	The ID of the node to be associated.
surfid	The ID of the surface to which the node is to be associated.
surfaceindex	The index into the surface that identifies the specific geometric entity to which the node is attached.

Returns Nothing.

HMIN_node_flush()

Sends all the nodes stored in the buffer to HyperMesh.

Syntax void HMIN_node_flush(void);

Returns Nothing.

HMIN_node_free()

Frees the memory associated with nodes stored with `HMIN_node_store()`.

Syntax void HMIN_node_free(void)

Returns Nothing.

HMIN_node_getfirstpointer()

Retrieves the first pointer to a node.

Syntax HMIN_nodepointer HMIN_node_getfirstpointer(void);

Returns The first node pointer.

Comments This function is called to find the first pointer; subsequent node pointers are retrieved by calling `HMIN_node_getnextpointer()`.

HMIN_node_getnextpointer()

Retrieves the next pointer to a node.

Syntax HMIN_nodepointer HMIN_node_getnextpointer(void);

Returns The next node pointer.

Comments `HMIN_node_getfirstpointer()` must be called to retrieve the first pointer before using this function.

HMIN_node_getpointer()

Retrieves the pointer to a node.

Syntax HMIN_nodepointer HMIN_node_getpointer(HM_entityidtype id);

id The ID of the node.

Returns The pointer to the node.

HMIN_node_store()

Sets up a buffer to provide a work area for nodes before transferring them to HyperMesh.

Syntax void HMIN_node_store(HM_entityidtype id, int attributes, double cords[3], HM_entityidtype superid, HM_entityidtype systemid, HM_entityidtype outputsystemid, HM_entityidtype surfaceid, int surfaceindex);

id	The ID of the node.
attributes	User-assigned value given the node.
cords[3]	The coordinates of the node.
superid	The ID of the super element of the node.
systemid	The ID of the system in which the node is built.
outputsystemid	The ID of the analysis system of the node.
surfaceid	The ID of the surface.
surfaceindex	The index value that identifies an entity within a surface.

Returns Nothing.

HMIN_node_write()

Writes a node to HyperMesh.

Syntax void HMIN_node_write(HM_entityidtype id, double cords[3], HM_entityidtype superid, HM_entityidtype systemid, HM_entityidtype outputsystemid, HM_entityidtype surfaceid, int surfaceindex);

id	The ID of the node.
cords[3]	The coordinates of the node.
superid	The ID of the super element of the node.
systemid	The ID of the system in which the node is built.
outputsystemid	The ID of the analysis system of the node.
surfaceid	The ID of the surface.
surfaceindex	The index value that identifies an entity within a surface.

Returns Nothing.

HMIN_node_writepointer()

Writes a node, accessed by the pointer to the node, to HyperMesh.

Syntax void HMIN_node_writepointer(HMIN_nodepointer nodeptr);

nodeptr The pointer to the node to be written out.

Returns Nothing.

HMIN_outputblock_write()

Writes an outputblock to HyperMesh.

Syntax void HMIN_ outputblock _write(HM_entityidtype id, char * name, char * entitytypename);

id The ID of the outputblock.

name The name of the outputblock.

entitytypename The entity type that the outputblock collects. This can be set to elements, nodes, components, systems, materials, or groups.

Returns Nothing.

Comments `HMIN_outputblock_writeid()` must be called after using `HMIN_outputblockwrite()` to put individual entities into the outputblock.

HMIN_outputblock_writeid()

Writes the entity into the specified outputblock.

Syntax void HMIN_ outputblock_writeid(HM_entityidtype entityid);

entityid The ID of the entity to be included in the outputblock.

Returns Nothing.

Comments `HMIN_outputblock_write()` must be called before using `HMIN_outputblock_writeid()` to put individual entities into the set.

Plots

HMIN_plot_findcurvelimits()

HMIN_plot_write()

HMIN_plot_writeaxis()

HMIN_plot_writeborder()

HMIN_plot_writebounds()

HMIN_plot_writecurve()

HMIN_plot_writegrid()

HMIN_plot_writelabel()

HMIN_plot_writelabels()

HMIN_plot_writelegend()

HMIN_plot_writesubtitle()

HMIN_plot_writetitle()

HMIN_plot_findcurvelimits()

Modifies the xy plot aixs limits so that the curves on the xy plot are fully visible.

Syntax HMIN_plot_findcurvelimits(char *plotname)

plotname The name of an existing plot.

Returns Nothing.

Comments This command should be called after all curve IDs are passed for a given plot.

HMIN_plot_write()

Writes a plot to HyperMesh.

Syntax void HMIN_plot_write(HM_entityidtype id, char * name, int type);

id The ID of the plot.

name The name of the plot.

type The type of the plot. Set to 1.

Returns Nothing.

HMIN_plot_writeaxis()

Writes the plot axis information to HyperMesh.

Syntax void HMIN_plot_writeaxis(int axistitlecolor, int axistitlefont, char * xaxistitle, char * xaxislabel, char * yaxistitle, char * yaxislabel, int xaxistype, int yaxistype, int xaxisgrids, int yaxisgrids, int xaxistics, int yaxistics, int xaxisdynamicrange, int yaxisdynamicrange, char xaxisformat, char yaxiisformat);

axistitlecolor The color to be used for the axis titles.

axistitlefont The font to be used for the axis titles (1 to 4).

xaxistitle The x axis title.

xaxislabel The x axis label.

yaxistitle The y axis title.

yaxislabel The y axis label.

xaxistype The format of the x axis. Use:

0 Decimal
1 Logarithmic
2 Decibel

yaxistype The format of the y axis. Use:

0 Decimal
1 Logarithmic
2 Decibel

xaxisgrids The number of grid marks per decade (logarithmic and decibel only).

yaxisgrids The number of grid marks per decade (logarithmic and decibel only).

xaxistics The number of tic marks per decade (logarithmic and decibel only).

yaxistics The number of tic marks per decade (logarithmic and decibel only).

xaxisdynamicrange Dynamic range for (x,y) axis offsetting.

yaxisdynamicrange Dynamic range for (x,y) axis offsetting.

xaxisformat Numeric format of axis values. Use:

a Automatic
f Fixed
e Exponential

yaxisformat Numeric format of axis values. Use:

a Automatic
f Fixed
e Exponential

Returns Nothing.

HMIN_plot_writeborder()

Writes the border of a plot to HyperMesh.

Syntax void HMIN_plot_writeborder(int borderon, int bordercolor, int borderwidth, int bordermargin, double borderxmin, double borderxmax, double borderymin, double borderymax);

borderon Indicates whether the border is on or off. Use:

1 If the border is on.

0 If the border is off.

bordercolor The color to be assigned to the border.

borderwidth The width to be assigned the border. Use:

0 For a thick border.

3 For a thin border.

bordermargin The margin of the border in pixels.

borderxmin The x value of the upper left plot window border.

borderxmax The x value of the lower right plot window border.

borderymin The y value of the upper left plot window border.

borderymax The y value of the lower right plot window border.

Returns Nothing.

Comments The values allowed for the x and y coordinates used for border minimums and maximums range from (0.0, 0.0) to (1.0, 1.0).

HMIN_plot_writebounds()

Writes the bounds of the plot to HyperMesh. Allows you to show specific sections of the plot.

Syntax void HMIN_plot_writebounds(double xmin, double xmax, double ymin, double ymax);

xmin The x value of the lower left axis range.

xmax The x value of the upper right axis range.

ymin The y value of the lower left axis range.

ymax The y value of the upper right axis range.

Returns Nothing.

HMIN_plot_writecurve()

Writes a curve that should be assigned to a plot to HyperMesh.

Syntax void HMIN_plot_writecurve(HM_entityidtype curveid);

curveid The ID of the curve.

Returns Nothing.

HMIN_plot_writegrid()

Writes the grid information of a plot to HyperMesh.

Syntax void HMIN_plot_writegrid(int gridlines, double gridxincrement, double gridyincrement, int mindivisions, int maxdivisions, int gridcolor, int gridxlabel, int gridylabel, int gridwidth);

gridlines Determines if the grid lines are shown. Use:

1 If grid lines are on.

0 If grid lines are off.

gridxincrement The increment size of grid lines on the x axis.

gridyincrement The increment size of grid lines on the y axis.

mindivisions The minimum number of grid divisions allowed.

maxdivisions The maximum number of grid divisions allowed.

gridcolor The color of the grid lines.

gridxlabel The x grid label frequency.

gridylabel The y grid label frequency.

gridwidth The width of the grid lines in pixels. Use:

1 For thick grid lines.

3 For thin grid lines.

Returns Nothing.

HMIN_plot_writelabel()

Writes the plot label information to HyperMesh.

Syntax void HMIN_plot_writelabels(char * label, int labelcolor, int labelfont);

label The label of the plot.

labelcolor The color of the label.

labelfont The font to be used to display the label (1 to 4).

Returns Nothing.

HMIN_plot_writelabels()

Writes the axis label information to HyperMesh.

Syntax void HMIN_plot_writelabels(int labelsformat, int labelsplaces, int labelscolor, int labelsfont, int margin);

labelsformat The format of the labels. Use:

0 If the format is integer.

1 If the format is real.

labelsplaces The number of places to be used for the label.

labelscolor The color to be used for the labels.

labelsfont The font to be used for the labels (1 to 4).

margin The margin between the labels and the plot grid lines in pixels.

Returns Nothing.

HMIN_plot_writelegend()

Writes the plot legend information to HyperMesh.

Syntax void HMIN_plot_writelegend(int legendon, double legendxloc, double legendyloc, int legendfont, int legendidson);

legendon Determines if the legend is shown. Use:

1 If the legend is on.

0 If the legend is off.

legendxloc The x location where the legend should be located.

legendyloc The y location where the legend should be located.

legendfont The font size to be used for the legend (1 to 4).

legendidson Determines if the legend IDs are shown. Use:

1 If the legend IDs are on.

0 If the legend IDs are off.

Returns Nothing.

HMIN_plot_writesubtitle()

Writes the plot subtitle information to HyperMesh.

Syntax void HMIN_plot_writesubtitle(char * subtitle, int subtitlecolor, int subtitlefont);

subtitle The subtitle of the plot.

subtitlecolor The color of the subtitle.

subtitlefont The font to be used to display the subtitle (1 to 4).

Returns Nothing.

HMIN_plot_writetitle()

Writes the plot title information to HyperMesh.

Syntax void HMIN_plot_writetitle(char * title, int titlecolor, int titlefont);

title	The title of the plot.
titlecolor	The color of the title.
titlefont	The font to be used to display the title (1 to 4).

Returns Nothing.

Properties

`HMIN_property_write()`

HMIN_property_write()

Writes a property to HyperMesh.

Syntax void HMIN_property_write(HM_entityidtype id, char * name, HM_entityidtype materialid);

id	The ID of the property.
name	The name of the property.
materialid	The material ID of the property.

Returns Nothing.

Rigid Walls

`HMIN_rigidwall_write()`

`HMIN_rigidwall_writegeometry_cylinder()`

`HMIN_rigidwall_writegeometry_madymo()`

`HMIN_rigidwall_writegeometry_plane()`

`HMIN_rigidwall_writegeometry_prism()`

`HMIN_rigidwall_writegeometry_sphere()`

`HMIN_rigidwall_writemotion()`

HMIN_rigidwall_write()

Writes a basic rigid wall to Hypermesh.

Syntax HMIN_rigidwall_write(HM_entityidtype id, double basex, double basey,double basez,double normalx,double normaly, double normalz);

id	The ID of the rigid wall.
basex, *basey*, *basez*	(x, y, z) locations of base node.
normalx, normaly, normalz	(x, y, z) locations used to calculate the vector for the rigid wall.

Returns Nothing.

Comments This command must be preceded by a `HMIN_group_write()` command with the same ID.

`normalx` should be equivalent to `basex` + $\hat{i}$, `normaly` should be equivalent to `basey` + $\hat{j}$, and `normalz` should be equivalent to `basek` + $\hat{k}$. This function was designed to be passed the values from the LS-DYNA3D `*RIGIDWALL` card without the values being modified.

HMIN_rigidwall_writegeometry_cylinder()

Writes a MADYMO coupling definition of a rigid wall.

Syntax HMIN_rigidwall_writegeometry_cylinder(HM_entityidtype id, double radius, double lengthz)

id	The ID of the rigid wall.
radius	The radius of the cylinder.
lengthz	The length of the cylinder.

Returns Nothing.

Comments This command must be preceded by a `HMIN_rigidwall_write()` command. The ID must refer to an existing rigid wall.

HMIN_rigidwall_writegeometry_madymo()

Writes a cylinder definition of a rigid wall.

Syntax HMIN_rigidwall_writegeometry_madymo(HM_entityidtype id, int ellipse, int madymoid)

id The ID of the rigid wall.

ellipse Flag to determine geometry. Use:

0 For plane.
1 For ellipse.

madymoid The ID of the plane or ellipse.

Returns Nothing.

HMIN_rigidwall_writegeometry_plane()

Writes a planar definition of a rigid wall.

Syntax HMIN_rigidwall_writegeometry_plane(HM_entityidtype id, unsigned char finite, double xaxisx, double xaxisy, double xaxisz, double lengthx, double lengthy)

id The ID of the rigid wall.

finite A flag for finite geometry.

xaxisx, *xaxisy*, *xaxisz* x, y, z locations used to calculate the local x axis.

lengthx, *lengthy* local x and y length of the rigid wall.

Returns Nothing.

Comments This command must be preceded by the `HMIN_rigidwall_write()` command. The ID must refer to an existing rigid wall.

`xaxisx`, `xaxisy`, and `xaxisz` relate to the base values of the rigid wall in the same way as `normalx`, `normaly`, and `normalz` relate to the base in `HMIN_rigidwall_write()`.

HMIN_rigidwall_writegeometry_prism()

Writes a prism definition of a rigid wall.

Syntax HMIN_rigidwall_writegeometry_prism(HM_entityidtype id, unsigned char finite, double xaxisx, double xaxisy, double xaxisz, double lengthx, double lengthy, double lengthz)

id	The ID of the rigid wall.
finite	A flag for finite geometry.
xaxisx, *xaxisy*, *xaxisz*	The x, y, z locations used to calculate the local x axis.
lengthx, *lengthy lengthz*	The local x, y and z length of the rigid wall.

Returns Nothing.

Comments This command must be preceded by the `HMIN_rigidwall_write()` command. The ID must refer to an existing rigid wall.

`xaxisx`, `xaxisy`, and `xaxisz` relate to the base values of the rigid wall in the same way as `normalx`, `normaly`, and `normalz` relate to the base in `HMIN_rigidwall_write()`.

HMIN_rigidwall_writegeometry_sphere()

Writes a sphere definition of a rigid wall.

Syntax HMIN_rigidwall_writegeometry_sphere(HM_entityidtype id, double radius);

id	The ID of the rigid wall.
radius	The radius of the sphere.

Returns Nothing.

Comments This command must be preceded by the `HMIN_rigidwall_write()` command. The ID must refer to an existing rigid wall.

HMIN_rigidwall_writemotion()

Writes a motion definition of a rigid wall.

Syntax HMIN_rigidwall_writemotion(HM_entityidtype id, unsigned char motiontype, double xcomp, double ycomp, double zcomp);

id	The ID of the rigid wall.
motiontype	The type of motion. Use: 0 None 1 Velocity 2 Displacement
xcomp, *ycomp*, *zcomp*	The x, y, z components of motion.

Returns Nothing.

Comments This command must be preceded by the `HMIN_rigidwall_write()` command. The ID must refer to an existing rigid wall.

HMIN_rigidwall_writenode()

Writes a basic rigid wall to Hypermesh.

Syntax HMIN_rigidwall_writenode(HM_entityidtype id, HM_entityidtype nodeid, normalx,double normaly, double normalz);

id	The ID of the rigid wall.
nodeid	The ID of the base node.
normalx, normaly, normalz	(x, y, z) locations used to calculate the vector for the rigid wall.

Returns Nothing.

Comments This command must be preceded by a `HMIN_group_write()` command with the same ID.

Sets

`HMIN_set_write()`

`HMIN_set_writeid()`

HMIN_set_write()

Writes a set to HyperMesh.

Syntax void HMIN_set_write(HM_entityidtype id, char * name, int color, char * entitytypename);

id	The ID of the set.
name	The name of the set.
color	The color to be assigned to the set.
entitytypename	The entity type that the set collects. This can be set to elements or nodes.

Returns Nothing.

Comments `HMIN_set_writeid()` must be called after using `HMIN_set_write()` to put individual entities into the set.

HMIN_set_writeid()

Writes the entity into the specified set.

Syntax void HMIN_set_writeid(HM_entityidtype entityid);

entityid	The ID of the entity to be included in the set.

Returns Nothing.

Comments `HMIN_set_write()` must be called before using `HMIN_set_writeid()` to put individual entities into the set.

Surfaces

`HMIN_start_trim_line()`

`HMIN_end_trim_line()`

`HMIN_startsurf()`

`HMIN_endsurf()`

`HMIN_start_object_line()`

`HMIN_start_parameter_line()`

`HMIN_surface_writefixedpoint()`

`HMIN_writealtline()`

`HMIN_writealtpoint()`

`HMIN_writealtsurface()`

HMIN_writegeomdata()

HMIN_writeNURBSsurface()

HMIN_writeplane()

HMIN_writesurfsat()

HMIN_start_trim_line()

Signals the start of a trimming line.

Syntax void HMIN_start_trim_line (int outer, int preference);

Syntax void HMIN_start_trim_line(int outer, int preference);

outer 1, if this is the outer trimming loop of the current surface; otherwise, 0.

preference Determines the type of trimming. Use:

0 No preference.

1 Object space trimming preferred.

2 Parameter space trimming preferred.

This is currently ignored by HyperMesh. It is here so that when object space trimming of NURBS is provided, this useful information is provided.

Returns Nothing.

Comments A trimming line consists of the start signal (this command), followed by a parameter space line, an object space line (or both), and then the end signal.

It is presumed that the parameter space line represents the object space line in the parameter space of the surface.

This command is only valid after a call to `HMIN_startsurf()` and before the corresponding call to `HMIN_endsurf()`.

HMIN_end_trim_line()

Signals the end of a trimming line.

Syntax void HMIN_end_trim_line(void);

Returns Nothing.

Comments This command is only valid after a call to `HMIN_startsurf()`, without an intervening call to `HMIN_endsurf()`.

HMIN_startsurf()

Signals the beginning of a surface.

Syntax void HMIN_startsurf(HM_entityidtype id, double tolerance, double ptolerance, HM_entityidtype componentid);

id	The ID of the surface.
tolerance	The tolerance to which you have created the surface. Two points less than this distance apart are considered the same point.
ptolerance	The tolerance for the parameter space on this surface. Two points less than this distance apart in parameter space are considered the same point.
componentid	The ID of the collector where the surface should be placed.

Returns Nothing.

HMIN_endsurf()

Signals the end of a surface.

Syntax void HMIN_endsurf(void);

Returns Nothing.

HMIN_start_object_line()

Signals the start of an object space line.

Syntax void HMIN_start_object_line(void);

Returns Nothing.

Comments This command operates identically to `HMIN_startline()`, but is used for trimming lines. It should be followed by one or more segments, and `HMIN_endline()`.

This command is only valid after a call to `HMIN_start_trim_line()` and before the corresponding call to `HMIN_end_trim_line()`.

HMIN_start_parameter_line()

Signals the start of a parameter space line.

Syntax void HMIN_start_parameter_line(void);

Returns Nothing.

Comments This command operates identically to `HMIN_startline()`, but is used for trimming lines. It should be followed by one or more segments, and a `HMIN_endline()`.

Parameter space lines should have Z = 0 throughout.

This command is only valid after a call to `HMIN_start_trim_line()` and before the corresponding call to `HMIN_end_trim_line()`.

HMIN_surface_writefixedpoint()

Writes a fixed meshing point for a surface.

Syntax HMIN_surface_writefixedpoint (double x, double y, double z, int suppressed)

x, *y*, *z*	Location of the fixed point.
suppressed	0 means this is where a vertex or meshing fixed point would not automatically appear, but should be added. 1 means this is where a vertex would have been placed, but should be omitted.

Comments The command, `HMIN_writesurfsat()` must be called before using `HMIN_surface_writefixedpoint()` to put fixed mesh points upon a surface.

HMIN_writealtline()

Syntax HMIN_writealtline(HM_entityidtype id, HM_entityidtypecollectorid, HM_entityidtype geomid)

id	The ID of the line.
collectorid	The ID of the collector where the line should be placed.
geomid	The internal database reference to connect up the line with its representation in the Altair geometry database

Returns Nothing.

HMIN_writealtpoint()

Syntax HMIN_writealtpoint(HM_entityidtype id, HM_entityidtypecollectorid, HM_entityidtype geomid)

id	The ID of the point.
collectorid	The ID of the collector where the point should be placed.
geomid	The internal database reference to connect up the point with its representation in the Altair geometry database

Returns Nothing.

HMIN_writealtsurface()

Syntax HMIN_writealtsurface(HM_entityidtype id, HM_entityidtypecollectorid, HM_entityidtype geomid)

id	The ID of the surface.
collectorid	The ID of the collector where the surface should be placed.
geomid	The internal database reference to connect up the surface with its representation in the Altair geometry database

Returns Nothing.

HMIN_writegeomdata()

Writes the location of an Altair geometry database in a file. The database must be in the internal format used by the template output command `*writegeometry()`.

Syntax void HMIN_writegeomdata(char * filename, long pos, int prefix_len);

filename	The name of the file that contains the SAT surface.
pos	The offset into the file.
prefix_len	The number of characters to ignore at the beginning of each line in the file.
id	The ID of the surface.
componentid	The component ID into which the surface should be placed.

Returns Nothing.

HMIN_writeNURBSsurface()

Specifies the information for a NURBS surface.

Syntax void HMIN_writeNURBSsurface(unsigned int degree_u, unsigned int degree_v, unsigned int number_pts_u, unsigned int number_pts_v, double * points, double * weights, unsigned int number_knots_u, double * knots_u, unsigned int number_knots_v, double * knots_v, double start_u, double end_u, double start_v, double end_v);

degree_u	The degree of the NURBS surface in the u direction.
degree_v	The degree of the NURBS surface in the v direction.
number_pts_u	The size of the control point array in the u direction.
number_pts_v	The size of the control point array in the v direction.
points	The control points, in the form X1, Y1, Z1, X2, Y2, Z2, and so on; the list should vary first in the u direction. (That is, the index of the X value of point [u, v] is (v * number_pts_u + u) * 3.) There should be number_pts_u * number_pts_v * 3 doubles here.
weights	If the NURBS is rational, this is the weight given to each control point, in the same order as the control points. If the NURBS is polynomial, a null pointer should be passed in.
number_knots_u	The number of knots in the NURBS's controlling u-knot sequence.
knots_u	The u-knot sequence.
number_knots_v	The number of knots in the NURBS's controlling v-knot sequence.
knots_v	The v-knot sequence.
start_u	The u parameter value at the start of the portion of the surface under consideration.
end_u	The u parameter value at the end of the portion of the surface under consideration.
start_v	The v parameter value at the start of the portion of the surface under consideration.
end_v	The v parameter value at the end of the portion of the surface under consideration.

Returns Nothing.

Comments A detailed description of the structure and meaning of NURBS surfaces is found in the *Altair HyperMesh 3.0 User's Manual*.

The NURBS format is very versatile. HyperMesh attempts to understand all NURBS surfaces, but you can ensure more accurate results and faster processing time by following these guidelines:

HyperMesh requires that its surfaces be C1 continuous. Thus multiple knots of multiplicity of the degree or greater force HyperMesh to modify the surfaces.

In particular, if one of the knot sequences has a knot with multiplicity greater than the degree of the surface in that direction, HyperMesh removes the resulting gap in the surface by averaging the points on each side of the gap.

If one of the knot sequences has a knot with multiplicity equal to the degree of the surface in that direction, HyperMesh attempts to remove a knot and a row or column of control points. If this would result in a change in the surface of greater than tolerance, HyperMesh is forced to break the surface into two or more faces. Such surfaces cannot be trimmed.

Multiple control points with multiplicity of the surface's degree or greater are not detected by HyperMesh at this time. Using them is not advised.

Parameterization on a surface should be relatively even. Arc length parameterization would be ideal, but it is not always possible.

Currently HyperMesh requires parameter space lines to trim NURBS surfaces. However, it is advisable to send object space lines as well; if then HyperMesh is unable to trim the surface, the object space lines are created in HyperMesh, and you may be able to trim the surface by hand with these lines, or re-create the surface from scratch.

This command is only valid after a call to `HMIN_startsurf()` and before the corresponding call to `HMIN_endsurf()`. Any trimming lines must be sent before the surface is sent.

HMIN_writeplane()

Specifies the information for a plane.

Syntax void HMIN_writeplane(double bx, double by, double bz, double nx, double ny, double nz);

bx	The x coordinate of the base of the plane.
by	The y coordinate of the base of the plane.
bz	The z coordinate of the base of the plane.
nx	The x coordinate of the vector indicating the plane's normal.
ny	The y coordinate of the vector indicating the plane's normal.
nz	The z coordinate of the vector indicating the plane's normal.

Returns Nothing.

Comments Unlike a NURBS surface, a plane must be trimmed.

HyperMesh requires object space trimming lines to trim a plane. Parameter space trimming lines are ignored - indeed, no parameter space is defined by the plane.

This command is only valid after a call to `HMIN_startsurf()` and before the corresponding call to `HMIN_endsurf()`. Any trimming lines must be sent before the surface is sent.

HMIN_writesurfsat()

Writes the location of a surface in SAT format within a file.

Syntax void HMIN_writesurfsat(char * filename, long pos, int prefix_len, HM_entityidtype id, HM_entityidtype componentid);

filename	The name of the file that contains the SAT surface.
pos	The offset into the file.
prefix_len	The number of characters to ignore at the beginning of each line in the file.
id	The ID of the surface.
componentid	The component ID into which the surface should be placed.

Returns Nothing.

System Collectors

`HMIN_systemcollector_write()`

HMIN_systemcollector_write()

Writes a system collector to HyperMesh.

Syntax void HMIN_systemcollector_write(HM_entityidtype id, char * name, int color);

id	The ID of the system collector.
name	The name of the system collector.
color	The color to be assigned to the system collector.

Returns Nothing.

Systems

`HMIN_system_flush()`

`HMIN_system_free()`

`HMIN_system_getfirstpointer()`

`HMIN_system_getnextpointer()`

`HMIN_system_getpointer()`

`HMIN_system_getstored()`

`HMIN_system_store()`

`HMIN_system_write()`

`HMIN_system_writeangle()`

`HMIN_system_writeatnode()`

`HMIN_system_writeinput()`

`HMIN_system_writeoutput()`

`HMIN_system_writepointer()`

HMIN_system_flush()

Sends the systems stored in the buffer to HyperMesh.

Syntax void HMIN_system_flush(void);

Returns Nothing.

HMIN_system_free()

Frees the memory associated with systems stored with `HMIN_system_store()`.

Syntax void HMIN_systems_free(void)

Returns Nothing.

HMIN_system_getfirstpointer()

Retrieves the first pointer to a system.

Syntax HMIN_systempointer HMIN_system_getfirstpointer(void);

Returns The pointer to the first system.

Comments This function is called to find the first pointer; subsequent system pointers are retrieved by calling `HMIN_system_getnextpointer()`.

HMIN_system_getnextpointer()

Retrieves the next pointer to a system.

Syntax HMIN_systempointer HMIN_system_getnextpointer(void);

Returns The next pointer to a system.

Comments `HMIN_system_getfirstpointer()` must be called to retrieve the first pointer before using this function.

HMIN_system_getpointer()

Retrieves the pointer to a system.

Syntax HMIN_systempointer HMIN_system_getpointer(HM_entityidtype id);

id The ID of the system.

Returns The pointer to the system.

HMIN_system_getstored()

Retrieves the number of stored systems.

Syntax int HMIN_system_getstored(void);

Returns The number of stored systems.

HMIN_system_store()

Sets up a buffer to provide a work area for systems before transferring them to HyperMesh.

Syntax void HMIN_system_store(HM_entityidtype id, int attributes, int type, HM_entityidtype systemid, double axis[3][3], double origin[3], HM_entityidtype systemcollectorid);

id The ID of the system.

attributes Value defined by you.

type The type of the system.

- 0 For a Cartesian system.
- 1 For a cylindrical system.
- 2 For a spherical system.

systemid The ID of the system where it is defined.

	axis[3][3]	The axes of the coordinate system.
	origin[3]	The origin of the coordinate system.
	Systemcollectorid	The ID of the system collector.

Returns Nothing.

HMIN_system_write()

Writes a system to HyperMesh.

Syntax void HMIN_system_write(HM_entityidtype id, int type, HM_entityidtype systemid, double axis[3][3], double origin[3], HM_entityidtype systemcolid);

id	The ID of the system.	
type	The type of the system.	
	0	For a Cartesian system.
	1	For a cylindrical system.
	2	For a spherical system.
systemid	The ID of the system to be defined.	
axis[3][3]	The axes of the coordinate system.	
origin[3]	The origin of the coordinate system.	
systemcolid	The ID of the system collector.	

Returns Nothing.

HMIN_system_writeangle()

Transforms a system based on angles.

Syntax void HMIN_system_writeangle(HM_entityidtype id, double angles[3]);

id	The ID of the system.
angles[3]	The angles by which the system is going to be transformed.

Returns Nothing.

Comments This function is available to support the ANSYS user-interface. Consult the ANSYS manual for more information on the location of the angles.

HMIN_system_writeatnode()

Creates a system at a node.

Syntax void HMIN_system_writeatnode(HM_entityidtype id, int type, HM_entityidtype systemid, double axis[3][3], HM_entityidtype nodeid, HM_entityidtype systemcollectorid);

id	The ID of the system.
type	The type of the system.
	0 For a Cartesian system.
	1 For a cylindrical system.
	2 For a spherical system.
systemid	The ID of the system in which the system is built.
axis[3][3]	The axes of the coordinate system.
nodeid	The ID of the node where it is defined.
Systemcollectorid	The ID of the system collector.

Returns Nothing.

HMIN_system_writeinput()

Defines an entity in another system after you have read in and transformed the entity in HyperMesh.

Syntax void HMIN_system_writeinput(int type, HM_entityidtype id, HM_entityidtype systemid);

type	The type of entity. Assign:
	HM_ENTITYTYPE_SYSTS for systems.
	HM_ENTITYTYPE_LOADS for loads.
	HM_ENTITYTYPE_NODES for nodes.
id	The ID of the entity.
systemid	The ID assigned to the input system of the entity.

Returns Nothing.

HMIN_system_writeoutput()

Sets the output system ID for an entity after you have read in and transformed the entity in HyperMesh.

Syntax void HMIN_system_writeoutput(int type, HM_entityidtype id, HM_entityidtype systemid);

type	The type of entity. The only valid type is HM_ENTITYTYPE_NODES.
id	The ID of the entity.
systemid	The ID assigned to the output system of the entity.

Returns Nothing.

HMIN_system_writepointer()

Writes a system, accessed by a pointer to the system, to HyperMesh.

Syntax void HMIN_system_writepointer(HMIN_systempointer systemptr);

systemptr	A pointer to the system to be written to HyperMesh.

Returns Nothing.

Titles

`HMIN_title_write()`

`HMIN_title_writeanchor()`

`HMIN_title_writeborder()`

HMIN_title_write()

Writes a title to HyperMesh.

Syntax void HMIN_title_write(HM_entityidtype id, char * name, int color, int font, char * text);

id	The ID of the title.
name	The name of the title.
color	The color to be used for the title.
font	The font to be used for the title. Fonts range from small to large (1 - 4).
text	The text to be included in the title.

Returns Nothing.

HMIN_title_writeanchor()

Writes the anchor for a title to HyperMesh.

Syntax HMIN_title_writeanchor(int anchorpoint, double anchorangle, double distance, char * entitytypename, HM_entityidtype entityid);

anchorpoint Identifies the corner of the title that is used as the anchor point. Use the setting:

- 0 To use the lower left corner as the anchor point.
- 1 To use the upper left corner as the anchor point.
- 2 To use the upper right corner as the anchor point.
- 3 To use the lower right corner as the anchor point.

anchorangle The angle relative to a vertical line running through the entity and the title lead.

distance The distance from the entity to the anchor point.

entitytypename The name of the entity to which the title is attached.

entityid The ID of the entity to which the title is attached.

Returns Nothing.

HMIN_title_writeborder()

Writes the border of a title to HyperMesh.

Syntax void HMIN_title_writeborder(int borderon, int bordercolor, int borderwidth, double borderxmin, double borderxmax, double borderymin, double borderymax);

borderon Indicates whether the border is on or off.

- 0 If the border is off.
- 1 If the border is on.

bordercolor The color to be assigned to the border.

borderwidth The width to be assigned to the border.

- 1 For a thick border.
- 3 For a thin border.

borderxmin The x value of the upper left plot window.

borderxmax The x value of the lower right plot window.

borderymin The y value of the upper left plot window.

borderymax The y value of the lower right plot window.

Returns Nothing.

Comments The values allowed for the x and y coordinates used for border minimums and maximums range from (0,0) to (1,1).

Vector Collectors

`HMIN_vectorcollector_write()`

HMIN_vectorcollector_write()

Writes a vector collector to HyperMesh.

Syntax void HMIN_vectorcollector_write(HM_entityidtype id, char *name, int color);

id The ID of the vector collector.

name The name of the vector collector.

color The color assigned to the load collector.

Returns Nothing.

Vectors

`HMIN_vector_write()`

`HMIN_vector_write_twonodes()`

`HMIN_vector_writecomponents()`

HMIN_vector_write()

Writes a vector entity to HyperMesh.

Syntax void HMIN_vector_write(HM_entityidtype id, HM_entityidtype basenode, double axis[3], double magnitude, HM_entityidtype collectorid)

id The ID of the vector.

basenode The ID of the node to which the vector is attached.

axis[3] The direction components of the vector [i,j,k].

magnitude The magnitude of the vector.

collectorid The ID of the vector collector.

systemid The ID of the system associated with the vector.

Returns Nothing.

HMIN_vector_write_twonodes()

Writes a vector entity created using the two-node method to HyperMesh.

Syntax void HMIN_vector_write_twonodes(HM_entityidtype id, HM_entityidtype basenode, HM_entityidtype farnode, HM_entityidtype collectorid)

id	The ID of the vector.
basenode	The ID of the basenode of the vector.
farnode	The ID of the second node of the vector.
collectorid	The ID of the vector collector.

Returns Nothing.

HMIN_vector_writecomponents()

Writes a vector entity to HyperMesh and creates a new node at the base coordinates.

Syntax void HMIN_vector_write(HM_entityidtype id, double base[3] double axis[3], double magnitude, HM_entityidtype collectorid)

id	The ID of the vector.
base [3]	Origin of the coordinate system.
axis[3]	The direction components of the vector.
magnitude	The magnitude of the vector.
collectorid	The ID of the vector collector.

Returns Nothing.

Introduction to hmreslib

If you are interfacing with a solver not currently supported by HyperMesh, you must write a results translator. To assist you in creating a HyperMesh binary results database, a library of C functions is provided with HyperMesh. This library is called `hmreslib. hmreslib` provides a very simple method for creating a HyperMesh binary results database which can be accessed by the post-processing functions in HyperMesh.

The HyperMesh Results Database

The HyperMesh results database is divided into sections called simulations. Each simulation represents the state of a model under an applied load. For a linear run, a simulation exists for each combination of applied load. For a nonlinear run, a simulation exists at every time step, or incremental amount of load, at which results are reported.

Each simulation is divided into sections called data types. Preferably, at least one data type exists within each simulation found in the database. Every data type stores a table of results in the form of nodal displacements, nodal values, or element values (real or complex). A data type can only hold one form of results, and once assigned, it cannot be changed. If the data type is of the form nodal displacements, then each record stores the ID of the node and the x, y, and z coordinate of displacement. Note that the displacement vector stored is relative to the original position of the node. If the data type is of the form nodal values, then each record stores the ID of the node and a value. Finally, just as the nodal value form stores a value per node, the element value form stores a value per element.

Visually, the relationship between the simulation, data type, and results is as follows:

```
100 N Applied to Beam (Simulation 1)
     Displacements (Data Type 1)
          node id, vector (result record)
          .
          .
          .

Von Mises Stress (Data Type 2)
          element id, von mises stress (result record)
          .
          .
          .
     .
     .
     .

100 N Applied to Cross Beam (Simulation 2)
     Displacements (Data Type 1)
```

```
            node id, vector (result record)
            .
            .
            .

        Von Mises Stress (Data Type 2)
            element id, von mises stress (result record)
            .
            .
            .

        .
        .
        .

    .
    .
    .
```

Each simulation and data type in a results file is assigned a name which is later presented to you. You then select the desired simulation and data type by selecting these names. For this reason, make the names assigned to simulation and data types meaningful to you. In addition, all of the simulation names in a results database must be unique, and all of the data type names in a simulation must also be unique.

The database contains a flag which indicates if the displacements recorded in the database are relative to the global coordinate system or a local nodal coordinate system. By default, all displacements are assumed to be relative to the global coordinate system. If this is not the case, you may set the local displacement flag, which causes HyperMesh to display the displacement vectors relative to the local nodal coordinate systems.

Creating a Database with hmreslib

The steps you need to follow to create a results database with `hmreslib` are listed below. When creating programs using C and `hmreslib`, keep in mind that C is case sensitive.

1. Create a simulation by calling `HMRES_simulationcreate()`.
2. Open the simulation to allow items to be placed inside by using `HMRES_simulationopen()`.
3. Create a data type by calling `HMRES_datatypecreate()`.
4. Open the data type by calling `HMRES_datatypeopen()`.
5. Add result records to the simulation and data type by calling the appropriate add function `HMRES_displacementadd()`or `HMRES_valueadd()`. These functions are called once for each node or element record to be added. For example, if there are results for 1000 nodes, then the ...add() functions must be called 1000 times for each data type. Each call passes one of the

1000 nodes and its respective values.

6. Continue to open as many data types as necessary for this simulation and store the appropriate results by repeating the last three steps.
7. Repeat the above steps until all of the simulations, data types for each simulation, and results for each data type are passed to `hmreslib`.
8. After all of the results are passed to `hmreslib`, call `HMRES_writeresults()` to transfer the memory image of the results to disk.

HyperMesh does not require that a data type contain a results record for every node or element in a model. Therefore, only create result records within a data type for the entities that have actual results.

The following rules apply when building a results database:

1. Only one simulation and data type may be opened at a time.
2. When a data type is created, it is placed in the currently open simulation.
3. When a result record is added to the database, it is placed in the currently open data type.
4. When a simulation or data type is created, it is not opened.
5. When a simulation or data type is opened, the simulation or data type that was previously opened is closed.
6. While simulations should only be created once and data types should only be created once for each simulation, they may be opened as many times as necessary. This feature may come in handy when results for a model are scattered in several different locations.

User Interface Functions

The user interface functions in `hmreslib` facilitate the writing of a results translator by helping to create a user interface for your translator. These functions are easy to use and offer a consistent interface.

Program Header

A header or title can be displayed by calling the function `HMRES_programheader()`. This function prints out a title of the program, version number, and a subtitle. Each of the arguments is definable and is displayed in a fashion consistent with other HyperMesh translators.

Arguments

The user interface functions handle argument processing from the command line. The argument functions allow you to create arguments which are displayed to, and selected by, the user.

An argument is an entity which is created by `hmreslib` upon request. Each argument has the following data associated with it:

datatypename	The name for the data that is associated with this argument.
argumentname	The character string that must be typed to select this argument.
type	The type of the argument.
selected	Determines if the argument is selected.

The first step in the procedure for using `hmreslib` arguments is to create or assign the arguments displayed. Arguments are assigned by using the function `HMRES_arguementassign()`. It is with this function call that the fields in the argument structure are initialized.

After all of the arguments are assigned, they must either be displayed, or used by the argument parser to determine which items are selected. To perform this operation, call the function `HMRES_arguementparse()`. The result of this function call indicates that either: (1) you have requested the program usage to be printed; or (2) the selected field in the arguments you selected is set to one.

After `HMRES_arguementparse()` has returned, you have the opportunity to inspect and modify the selected arguments. In this way, you can check to see if an argument is selected, and if so, you may select several other arguments as well. The functions that modify and return the selected status of an argument are `HMRES_arguementgetselected()` and `HMRES_arguementgetselected()`.

After completing any modifications to the selected arguments, you can be informed of their final selections, and any modifications, by calling the function, `HMRES_arguementprintselected()`.

`HMRES_argumentassign()`

`HMRES_argumentgetselected()`

`HMRES_argumentparse()`

`HMRES_argumentprintselected()`

Reserved Arguments

Several argument names cannot be used by `hmreslib`. The reserved arguments and descriptions of their use follow:

disk	Translation is performed on disk.
size	Maximum number of entities that are going to be stored per data type.
file	The file name for the swap file. This can be used to modify the location of the swap file from the system default.
cray	Used internally for number conversions.
dec	Used internally for number conversions.
decalpha	Used internally for number conversions.
ibm	Used internally for number conversions.
pc	Used internally for number conversions.
sgi	Used internally for number conversions.
sun	Used internally for number conversions.
hp	Used internally for number conversions.

Programming Notes

`hmreslib` allows you to translate a file from `stdin` to `stdout`. It is important to note that since `stdout` is redirected by you to a file, programs that perform translations should not write to `stdout`. If they do, corruption of the results file occurs. Instead, send all output that communicates with you to `stderr`. The following is an example of writing to standard error:

```
fprintf(stderr,"Hello World\n");
```

When opening input files, call `HMRES_openinputfile()`, which detects if `stdin` is being opened and returns the appropriate pointer.

Cross Platform Translation

hmreslib allows for cross platform translation. This means that a binary file generated on a Cray may be translated on any of the supported machines. In a similar way, a binary file created on an SGI can be translated on any of the supported machines. While it may present a challenge, it is possible to write a translator which supports cross platform translation.

To assist in cross platform translations, four functions are provided in hmreslib. The first, HMRES_arguementcrossplatform(), must be called before HMRES_arguementparse(). This function activates the reserved arguments associated with platform specification. In addition, it initializes the number conversion process. The other three functions provided are HMRES_readbinaryint(), HMRES_readbinaryint2, and HMRES_readbinaryfloat(). These subroutines read their respective data types from the file and convert the contents of the file to the correct data type on the machine where the translation is being performed.

It is recommended that only experienced programmers attempt cross platform translation.

hmreslib Functions

The ANSI C functions found in hmreslib can be linked into any user program.

HMRES_argumentassign()

HMRES_argumentcategory()

HMRES_argumentcrossplatform()

HMRES_argumentgetargumentname()

HMRES_argumentgetdatatypename()

HMRES_argumentgetlayerdatatypename()

HMRES_argumentgetselected()

HMRES_argumentgettype()

HMRES_argumentparse()

HMRES_argumentprintselected()

HMRES_argumentsetselected()

HMRES_argumentuserparsefunction()

HMRES_argumentuserprintfunction()

HMRES_calculateprincipals()

HMRES_calculatevonmises()

HMRES_close()

HMRES_complexdisplacementadd()

HMRES_complexnumbersform()

HMRES_complexvalueadd()

HMRES_complexvonmisesadd()

HMRES_convertbuffertodouble()

HMRES_convertbuffertofloat()

HMRES_convertbuffertoint()

HMRES_convertdoublebuffer()

HMRES_convertfloatbuffer()

HMRES_convertintbuffer()

HMRES_datatypeclose()

HMRES_datatypecreate()

HMRES_datatypeexists()

HMRES_datatypeopen()

HMRES_displacementadd()

HMRES_getcomplexnumbersform()

HMRES_getinputfileauthor()

HMRES_getmachineformat()

HMRES_initialize()

HMRES_localdisplacements()

HMRES_magnitudephasetorealimaginary()

HMRES_maxsimulations()

HMRES_openinputfile()

HMRES_programheader()

HMRES_readbinaryfloat()

HMRES_readbinaryint()

HMRES_readbinaryint2()

HMRES_realimaginarytomagnitudephase()

HMRES_simulationclose()

HMRES_simulationcreate()

HMRES_simulationexists()

HMRES_simulationgetnamefromid()

HMRES_simulationopen()

HMRES_terminate()

HMRES_valueadd()

HMRES_writeresults()

HMRES_writesimulation()

HMRES_argumentassign()

Assigns arguments to the hmreslib program.

Syntax void HMRES_argumentassign(int index, char * datatypename, char * argumentname, int type);

Index	The index number of the argument to be assigned. The index can be any integer greater than or equal to zero. Assign indexes in sequence; previous index numbers should already be defined.
datatypename	The name associated with that argument, i.e., displacements, stresses, reaction forces.
argumentname	The name of the argument being assigned.
type	The type of the argument being assigned. Types include:

1. HMRES_ARGUMENT_TYPE_ SELECTABLE, for arguments which you may select.
2. HMRES_ARGUMENT_TYPE_ AUTOSELECTABLE, for arguments which are automatically set if no options are selected.
3. HMRES_ARGUMENT_TYPE_USER, for arguments which must be resolved by you. Refer to the functions: HMRES_arguementuserparsefunction(), HMRES_arguementuserprintfunction(). This type requires that a user-defined function parse the argument list.
4. HMRES_ARGUMENT_TYPE_USER_FILE, for file arguments which must be resolved by you. Refer to the functions: HMRES_arguementuserparsefunction(),HMRES_arguementuserprintfunction(). This type requires that a user-defined function parse the argument list.

Returns Nothing.

HMRES_argumentcategory()

Call this function to place arguments into categories.

Syntax void HMRES_argumentcategory(char *string);

string A static string (it is not copied) that sets the category for all subsequent arguments.

Returns Nothing.

Comments When a results translator is invoked with the -gui option, the output displays categories for sets of options. This output can be parsed to other programs to automate the creation of a graphical user interface (gui).

HMRES_argumentcrossplatform()

Allows translation of binary files across a platform. This function gives a list of arguments that allows you to tell a program which machine wrote the binary file.

Syntax void HMRES_argumentcrossplatform(void);

Returns Nothing.

HMRES_argumentgetargumentname()

Indicates the argument name at the user-specified index.

Syntax char * HMRES_argumentgetargumentname(int index);

index The index number of the argument to be checked.

Returns The name of the argument at the specified index.

HMRES_argumentgetdatatypename()

Indicates the data type name of the argument at the user-specified index.

Syntax char * HMRES_argumentgetdatatypename(int index);

index The index number of the argument to be checked.

Returns The data type name of the argument at the specified index.

HMRES_argumentgetlayerdatatypename()

Indicates the data type name and appends, in parentheses, the value of layer at the user-specified index.

Syntax char * HMRES_argumentgetlayerdatatypename(int index, int layer);

index The index number of the argument to be checked.

layer The layer on which the results are stored.

Returns The data type name and value of layer, if it is nonzero, at the specified index.

HMRES_argumentgetselected()

Indicates if the argument at the user-specified index is selected.

Syntax int HMRES_argumentgetselected(int index);

index The index number of the argument to be checked.

Returns 1, if the argument was selected.

0, if the argument was not selected.

HMRES_argumentgettype()

Indicates the type associated with the argument at the user-specified index.

Syntax int HMRES_argumentgettype(int index);

index The index number of the argument to be checked.

Returns The type of argument at the specified index. Types specified are:

```
HMRES_ARGUMENT_TYPE_SELECTABLE
HMRES_ARGUMENT_TYPE_AUTOSELECTABLE
HMRES_ARGUMENT_TYPE_USER
```

HMRES_argumentparse()

Takes the arguments passed into the program and determines what is selected.

Syntax void HMRES_argumentparse(int argc, char * argv[], char * inputfilename, char * outputfilename, char * modelfilename);

argc	Argument passed in from the C function main.
argv[]	Argument passed in from the C function main.
inputfilename	Memory location where `argumentparse()` can place the user-selected input file name.
outputfilename	Memory location where `argumentparse()` can place the user-selected output file name.
modelfilename	Memory location `where argumentparse()` can place the user-selected model file name. If the translator being used does not deal with model results, pass NULL as the *modelfilename*.

Returns Nothing.

HMRES_argumentprintselected()

Prints the selected arguments.

Syntax void HMRES_argumentprintselected(char * inputfilename, char * outputfilename, char * modelfilename);

inputfilename	The pointer to the user-selected input file name.
outputfilename	The pointer to the user-selected output file name.
modelfilename	The pointer to the user-selected model file name. If the translator being used does not deal with model results, pass NULL as the *modelfilename*.

Returns Nothing.

HMRES_argumentsetselected()

Sets an argument as being selected. This function can also be used to change whether an argument is selected or not.

Syntax void HMRES_argumentsetselected(int index, int selected);

index	The index number of the argument to be selected or unselected.

selected Determines if the argument is selected or unselected. Use:

1 If the argument is to be selected.

0 If the argument is to be unselected.

Returns Nothing.

HMRES_argumentuserparsefunction()

Assigns the function used to parse the arguments.

Syntax void HMRES_argumentuserparsefunction(int * argfunction, int argument, char * argumentplusone);

argfunction The function called by `hmreslib` when an argument is defined as a "User" type argument.

argument The number of the argument.

argumentplusone The number of the argument plus one.

Returns Nothing.

Comments See `HMRES_arguementassign()`.

HMRES_argumentuserprintfunction()

Assigns the function that prints out what the argument is assigned.

Syntax void HMRES_argumentuserprintfunction(void * printfunction, int argument);

printfunction The function that is called by `hmreslib` when an argument is defined as a "User" type argument.

argument The number of the argument.

Returns Nothing.

Comments See `HMRES_arguementassign()`.

HMRES_calculateprincipals()

Calculates principal stresses given triaxial and shear stresses.

Syntax void HMRES_calculateprincipals(double sigx, double sigy, double sigz, double tauxy, double tauyz, double tauxz, double *principal1, double *principal2, double *principal3);

sigx	The normal stress in the x-direction.
sigy	The normal stress in the y-direction.
sigz	The normal stress in the z-direction.
tauxy	The shear stress in the x-y plane.
tauyz	The shear stress in the y-z plane.
tauxz	The shear stress in the x-z plane.
principal1	The principal1 stress result.
principal2	The principal2 stress result.
principal3	The principal3 stress result.

Returns Nothing.

HMRES_calculatevonmises()

Calculates the vonMises stress given triaxial and shear stresses.

Syntax void HMRES_calculatevonmises(double sigx, double sigy, double sigz, double tauxy, double tauyz, double tauxz, double *vonmises);

sigx	The normal stress in the x-direction.
sigy	The normal stress in the y-direction.
sigz	The normal stress in the z-direction.
tauxy	The shear stress in the x-y plane.
tauyz	The shear stress in the y-z plane.
tauxz	The shear stress in the x-z plane.
Vonmises	The vonMises stress result.

Returns Nothing.

HMRES_close()

Closes the `hmreslib` file and frees memory used by `hmreslib` for temporary files. (You must close other open files.)

Syntax void HMRES_close(void);

Returns Nothing.

Comments This function prints the message, "Translation Complete." Call this function after using `hmreslib`.

HMRES_complexdisplacementadd()

Adds a complex number displacement to a data type.

Syntax int HMRES_complexdisplacementadd(HM_entityidtype id, double xmag, doubleymag, double zmag, double xphase, double yphase, double zphase);

id	The ID of the node.
xmag	The magnitude of the x coordinate of displacement.
ymag	The magnitude of the y coordinate of displacement.
zmag	The magnitude of the z coordinate of displacement.
xphase	The phase of the x coordinate of displacement in degrees.
yphase	The phase of the y coordinate of displacement in degrees.
zphase	The phase of the z coordinate of displacement in degrees.

Returns Nothing.

HMRES_complexnumbersform()

Sets the form of the complex number that `hmreslib` is expecting.

Syntax void HMRES_complexnumbersform(int form);

form Determines the form of the complex number.

0	The default value; tells `hmreslib` to expect complex numbers in polar (magnitude/phase) format.
1	Tells `hmreslib` to expect complex numbers in real/imaginary format.

Returns Nothing.

Comments By default, `hmreslib` expects complex numbers to be in polar (magnitude/phase) format; this is the same as calling `HMRES_complexnumbersform()` with a form set to zero. Setting form to one and calling this function makes `hmreslib` accept complex numbers in real/imaginary format. This changes the meaning of the functions that pass complex numbers to `hmreslib` in that you should pass the real and imaginary part of the complex number where the function would by default expect magnitude and phase.

HMRES_complexvalueadd()

Adds a complex value to a data type.

Syntax int HMRES_complexvalueadd(HM_entityidtype id, double magnitude, double phase);

id The ID of the node or element.

magnitude The magnitude component of the complex number.

phase The phase component of the complex number in degrees.

Returns Nothing.

HMRES_complexvonmisesadd()

Use to add complex vonmises results to a data type.

Syntax int HMRES_complexvonmisesadd(HM_entityidtype id, double magnitude, doublephase, double offset);

id	The ID of the node or element.
magnitude	The magnitude component of the complex number.
phase	The phase component of the complex number in degrees.
offset	The offset applied to the sine wave.

Returns Nothing**.**

Comments Use to add complex vonmises results to a data type. `hmreslib` stores complex results in polar format (magnitude/phase). This presents a problem, as vonmises stress cannot be represented as a complex number. Instead, `hmreslib` stores vonmises stress squared. This requires that a complex number and an offset be written to the results file.

To calculate the correct values for the magnitude, phase, and offset, a three point method can be used. First obtain the square of vonmises stress at a base angle of 0, 45, and 180 degrees. After these three values are found, the following equations provide the correct magnitude, phase, and offset:

VM1, VM2, and VM3 are the three points where vonmises squared is evaluated at 0, 45, and 90 degrees respectively.

```
alpha = 2*VM2-VM1-VM3
beta = VM3-VM1
gamma = VM1+VM3
offset = gamma/2.0
magnitude = 0.5*sqrt(alpha*alpha+beta*beta)
phase = 180.0*atan2(beta,alpha)/PI
```

HMRES_convertbuffertodouble()

Converts a buffer containing floating point information from one machine type to another machine type.

Syntax double HMRES_convertbuffertodouble(int fromformat, int toformat, char * buffer);

fromformat	The format of the "from" buffer.
toformat	The format to which the "from" buffer should be converted.
buffer	A pointer to the buffer being converted.

Returns Returns the converted floating point value.

HMRES_convertbuffertofloat()

Converts a buffer containing floating point information from one machine type to another machine type.

Syntax float HMRES_convertbuffertofloat(int fromformat, int toformat, char * buffer);

fromformat	The format of the "from" buffer.
toformat	The format to which the "from" buffer should be converted.
buffer	A pointer to the buffer being converted.

Returns Returns the converted floating point value.

HMRES_convertbuffertoint()

Converts a buffer containing integer information from one machine type to another machine type.

Syntax int HMRES_convertbuffertoint(int fromformat, int toformat, char * buffer);

fromformat	The format of the "from" buffer.
toformat	The format to which the "from" buffer should be converted.
buffer	A pointer to the buffer being converted.

Returns Returns the converted integer value.

HMRES_convertdoublebuffer()

Converts a buffer containing floating point information from one machine type to another machine type.

Syntax void HMRES_convertdoublebuffer(int fromformat, char * frombuffer, int toformat, char * tobuffer);

fromformat	The format of the "from" buffer.
frombuffer	A pointer to the buffer containing a floating point value.
toformat	The format to which the "from" buffer should be converted.
tobuffer	A pointer to the buffer where the converted floating pointer value should be placed.

Returns Nothing.

HMRES_convertfloatbuffer()

Converts a buffer containing floating point information from one machine type to another machine type.

Syntax void HMRES_convertfloatbuffer(int fromformat, char * frombuffer, int toformat, char * tobuffer);

fromformat	The format of the "from" buffer.
frombuffer	A pointer to the buffer containing a floating point value.
toformat	The format to which the "from" buffer should be converted.
tobuffer	A pointer to the buffer where the converted floating pointer value should be placed.

Returns Nothing.

HMRES_convertintbuffer()

Converts a buffer containing integer information from one machine type to another machine type.

Syntax void HMRES_convertintbuffer(int fromformat, char * frombuffer, int toformat, char * tobuffer);

fromformat	The format of the "from" buffer.
frombuffer	A pointer to the buffer containing an integer value.
toformat	The format to which the "from" buffer should be converted.
tobuffer	A pointer to the buffer where the converted integer value should be placed.

Returns Nothing.

HMRES_datatypeclose()

Closes the current data type.

Syntax void HMRES_datatypeclose(void);

Returns Nothing.

HMRES_datatypecreate()

Creates a data type.

Syntax int HMRES_datatypecreate(char * datatypename, int form);

datatypename The name of the data type to be created.

form The form of the data type. This determines the type of data being stored in the data type. Select the data type from the following:

1. HMRES_NODALDISPLACEMENTS, if the data type should store nodal displacements.
2. HMRES_NODALVALUES, if the data type should store nodal values.
3. HMRES_ELEMENTVALUES, if the data type should store element values.
4. HMRES_COMPLEXNODALDISPLACEMENTS, if the data type should store displacement results (a triplet in x, y, and z) as three complex numbers at each node. Use `HMRES)complexdisplacementadd()` to add results to a data type of this form.
5. HMRES_COMPLEXNODALVALUES, if the data type should store results as one complex number at each node. Use `HMRES_complexvalueadd()` to add results to a data type of this form.
6. HMRES_COMPLEXELEMENTVALUES, if the data type should store results as one complex number at each element. Use `HMRES_complexvalueadd()` to add results to a data type of this form.
7. HMRES_COMPLEXNODALVONMISES, if the data type should store the square of VonMises stress as a complex number at each node. Use `HMRES_complexvonmisesadd()` to add results to a data type of this form.
8. HMRES_COMPLEXELEMENTVONMISES, if the data type should store the square of VonMises stress as a complex number at each element. Use `HMRES_complexvonmisesadd()` to add results to a data type of this form.

Returns Zero, if successful; otherwise, nonzero.

Comments The *datatypename* is presented when the data type to be post-processed is selected.

HMRES_datatypeexists()

Determines if the data type already exists.

Syntax int HMRES_datatypeexists(char * datatypename);

datatypename The name of the data type to be checked.

Returns Zero, if the data type does not exist.

One, if the data type already exists.

HMRES_datatypeopen()

Opens a data type.

Syntax int HMRES_datatypeopen(char * datatypename);

datatypename The name of the data type to be opened.

Returns Zero, if successful; otherwise, nonzero.

Comments Before a data type can be opened, it must be created.

HMRES_displacementadd()

Adds a displacement value to the results file of the data type.

Syntax int HMRES_displacementadd(HM_entityidtype id, double x, double y, double z);

id The ID of the node at which the displacement occurs.

x The x coordinate of displacement.

y The y coordinate of displacement.

z The z coordinate of displacement.

Returns Zero, if successful; otherwise, nonzero.

Comments Before a displacement can be stored in the results database, a simulation and data type must be opened.

The data type that is currently open when this function is called must be capable of storing nodal displacements.

HMRES_getcomplexnumbersform()

Inquires about which format is currently being used to interpret complex numbers.

Syntax int HMRES_getcomplexnumbersform(void);

Returns 0, if the current format is magnitude/phase.

1, if the current format is real/imaginary.

HMRES_getinputfileauthor()

Identifies the author of the binary file being translated.

Syntax int HMRES_getinputfileauthor(void);

Returns Returns the author of the binary file.

HMRES_getmachineformat()

Identifies the format of a machine.

Syntax HMRES_getmachineformat(int machine);

machine The machine type.

Returns The format of the machine.

HMRES_initialize()

Initializes `hmreslib`.

Syntax void HMRES_initialize(int numindatatype);

numindatatype The number of entities that are expected to be stored in a data type. Setting this value to zero uses the default value of 500.

Returns Nothing.

Comments This function must be called before any other function in the `hmreslib` package.

HMRES_localdisplacements()

Sets the local displacements flag in the database.

Syntax void HMRES_localdisplacements(int local);

local Flag that indicates where the displacements are defined. If set to:

0 The displacements are assumed to be in the global coordinate system.

1 The displacements are assumed to be in a local nodal coordinate system.

Returns Nothing.

Comments By default, the database is created as if all of the displacements are defined in the global coordinate system. If this is not the case, this function must be called with local set to 1.

If the local displacement flag is set in a results database, HyperMesh transforms the displacements when the results are processed.

HMRES_magnitudephasetorealimaginary()

Use to convert a complex number in magnitude/phase format to real/imaginaryformat. This function modifies the values passed.

Syntax void HMRES_magnitudephasetorealimaginary(double *magnitude, double *phase);

magnitude The magnitude of a complex number in polar form.

phase The phase of a complex number in polar form in degrees.

HMRES_maxsimulations()

Sets the maximum number of simulations.

Syntax void HMRES_maxsimulations(int max)

max The maximum number of simulations.

Returns Nothing.

Comments This function must be called before creating any simulations.

HMRES_openinputfile()

Opens an input file containing results.

Syntax FILE * HMRES_openinputfile(char * filename, char * mode);

filename The name of the results file to be opened.

mode The type of file to be opened. Use:

rb If the file is a binary file.

rt If the file is a text file.

Returns A pointer to the file.

Comments This function can take the arguments passed in by you and open standard input as its input stream, if a file name is not specified.

HMRES_programheader()

Creates a program header.

Syntax void HMRES_programheader(char * title, char * version, char * subtitle, int rightjustify);

title The title of the translator. Appended to the title is the phrase, "to HyperMesh Binary Results."

version The version of the user translator.

subtitle The subtitle of the translator.

rightjustify Variable that determines the title justification. You should assign a value of:

1 If a right-justified title is desired.

0 If a left-justified title is desired.

Returns Nothing.

HMRES_readbinaryfloat()

Reads a floating-point number from a binary file which opened with `HMRES_openinputfile()`. It pays attention to the cross platform flags and does a conversion on the number, making it understandable to the machine the program is running on.

Syntax float HMRES_readbinaryfloat(FILE * file);

file The binary file containing the floating-point number to be read.

Returns The value of the floating-point number.

HMRES_readbinaryint()

Reads a binary integer from a binary file opened with `HMRES_openinputfile()`. It pays attention to the cross platform flags and does a conversion on the integer, making it understandable to the machine the program is running on.

Syntax int HMRES_readbinaryint(FILE * file);

file The binary file containing the integer to be read.

Returns The value of the integer.

HMRES_readbinaryint2()

Reads a 2-byte integer from a binary file opened with `HMRES_openinputfile()`. It pays attention to the cross-platform flags and does a conversion on the number, making it understandable to the machine on which the program is running.

Syntax int HMRES_readbinaryint2(FILE *file);

file The binary file containing the 2-byte integer to be read.

Returns The value of the 2-byte integer.

HMRES_realimaginarytomagnitudephase()

Use to convert a complex number in real/imaginary format to magnitude/phase format. This function modifies the values passed.

Syntax void HMRES_realimaginarytomagnitudephase(double *real, double *imaginary);

Real	The real component of a complex number.
Imaginary	The imaginary component of a complex number.

HMRES_simulationclose()

Closes the current simulation.

Syntax void HMRES_simulationclose(void);

Returns Nothing.

HMRES_simulationcreate()

Creates a simulation in the results database.

Syntax int HMRES_simulationcreate(char * simulationname, int simulationid);

simulationname	The name of the simulation to be created. If left blank, HyperMesh creates the simulation for you from the simulation ID.
simulationid	The ID of the simulation to be created.

Returns Zero, if successful; otherwise, nonzero.

Comments This function must be called before any data can be sent to the database.

The simulation name is presented when a simulation is selected.

The ID is ignored unless the name passed is zero length. In this case, HyperMesh creates a simulation name based on the ID, which must be nonzero. A simulation must be created before it can be opened.

HMRES_simulationexists()

Determines if the simulation already exists.

Syntax int HMRES_simulationexists(char * simulationname, int simulationid);

simulationname The simulation name of the simulation to be checked.

simulationid The simulation ID of the simulation to be checked.

Returns Zero, if the simulation does not exist.

One, if the simulation already exists.

HMRES_simulationgetnamefromid()

Finds a simulation name based on its ID.

Syntax char * HMRES_simulationgetnamefromid(int simulationid);

simulationid The ID of the simulation for which the name should be found.

Returns The name of the simulation.

HMRES_simulationopen()

Opens a simulation.

Syntax int HMRES_simulationopen(char * simulationname, int simulationid);

simulationname The simulation name of the simulation to be opened. If left blank, the ID is used to create the name.

simulationid The simulation ID of the simulation to be opened.

Returns Zero, if successful; otherwise, nonzero.

Comments The ID is ignored unless the name passed is zero length. In this case, HyperMesh creates a simulation name based on the ID, which must be nonzero.

A simulation must be created before it can be opened.

HMRES_terminate()

Allows a user-specified message to be sent before the program is terminated.

Syntax void HMRES_terminate(char * format, char * message);

format A pointer to a string designating the format for a printf statement.

message A pointer to a string containing a message specified by you. This variable can be set to NULL.

Returns Nothing.

Comments This function calls `exit()` after printing the message.

HMRES_valueadd()

Adds a value to the current open data type.

Syntax int HMRES_valueadd(HM_entityidtype id, double value);

id The ID of the node or element that should be assigned the value.

value The value to be assigned to the node or element.

Returns Zero, if successful; otherwise, nonzero.

Comments Before a value can be stored in the results database, a simulation and data type must be opened.

The data type that is currently open when this function is called must be capable of storing nodal or element values.

HMRES_writeresults()

Writes a HyperMesh binary database containing the results previously stored in a file in `hmreslib`.

Syntax int HMRES_writeresults(char * filename);

filename The name of the file that should be created.

Returns Zero, if successful; otherwise, nonzero.

Comments This function overwrites the file if it already exists.

HMRES_writesimulation()

Writes a simulation to a file.

Syntax int HMRES_writesimulation(char * filename);

filename The name of the file to which the results are written.

Returns Zero, if successful; otherwise, nonzero.

Comments Use this function to write a simulation.

An Example Translator

The example presented below is in the file `hmtrans.c` in the directory lib where HyperMesh is installed. Although simple, it illustrates the basic framework required to build a HyperMesh binary results database with `hmreslib`.

```
/*

  This example program shows how to use hmreslib.  hmreslib is
  designed to allow you to create your own translators, which
  translate data from an analysis code to a HyperMesh binary
  results file.  Complete documentation for hmreslib is in the
  HyperMesh Programmer's Manual.

  This program reads an ASCII file 'results.asc' containing
  results from an analysis program and places them in a
  HyperMesh binary results file.  An ANSI C compiler is
  required. To compile, enter the following:

  cc hmtrans.c <HM dir>/lib/hmreslib.a <HM dir>/lib/hmlib.a -o
    hmtrans -lm

  <HM dir> refers to the directory where HyperMesh resides.  Note
  that the name of the compiler (cc) may vary on
  different systems.  It may be necessary for you to copy the
  file 'hmreslib.h' from the HyperMesh library directory to
  your directory.

  After compilation has completed, the executable that generates
  the results file is named 'hmtrans'.  To execute, you must
  have a copy of the file 'results.asc', found in the HyperMesh
```

```
  lib directory.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hmlib.h"
#include "hmreslib.h"

enum
{
  DISPLACEMENTS,
  STRESS,
  DISPLACEMENTANDSTRESS,
  DISPLACEMENTOFFSET
};

static double DisplacementOffset = 0.0;

void userprintfunction(int argument)
{
  /*
     A user-defined function to handle printing of more complex arguments.
  */
  switch (argument)
  {
    case DISPLACEMENTOFFSET:
      fprintf(stderr,"\n");
      fprintf(stderr,"  The displacement offset is set to: %f\n",
        DisplacementOffset);
      break;
  }
}

int userparsefunction(int argument, char *argumentplusone)
{
  /*
     A user-defined function to handle parsing of more complex arguments.
  */
  switch (argument)
  {
    case DISPLACEMENTOFFSET:
```

```
      DisplacementOffset = atof(argumentplusone);
      return(1);
  }
  return(0);
}

void main(int argc, char *argv[])
{
  FILE *inputfile;
  char string[256];
  char subcasename[256];
  char datatypename[256];
  HM_entityidtype id;
  int datatypeopen = 0;
  double x;
  double y;
  double z;
  double value;
  char inputfilename[256];
  char outputfilename[256];

  /*
     Initialize hmreslib.
 */
  HMRES_initialize(0);
  /*
     Print the program title
 */
  HMRES_programheader("Example Translator","1.0",
    "(c) copyright Altair/Finite Applications",1);
  /*
     Assign the desired arguments, parse, and display
 */
  HMRES_argumentassign(DISPLACEMENTS,"Displacements","d",
    HMRES_ARGUMENT_TYPE_AUTOSELECTABLE);
  HMRES_argumentassign(STRESS,"Stresses","s",
    HMRES_ARGUMENT_TYPE_AUTOSELECTABLE);
  HMRES_argumentassign(DISPLACEMENTOFFSET,"Displacement Offset","off",
    HMRES_ARGUMENT_TYPE_USER);
  HMRES_argumentassign(DISPLACEMENTANDSTRESS,"d and s","ds",
    HMRES_ARGUMENT_TYPE_SELECTABLE);
  HMRES_argumentuserparsefunction(userparsefunction);
```

```
 HMRES_argumentuserprintfunction(userprintfunction);
 HMRES_argumentparse(argc,argv,inputfilename,outputfilename,NULL);
 /*
    Check to see if argument DISPLACEMENTANDSTRESS was
    selected and modify the selected arguments.
*/
 if (HMRES_argumentgetselected(DISPLACEMENTANDSTRESS))
 {
   HMRES_argumentsetselected(DISPLACEMENTANDSTRESS,0);
   HMRES_argumentsetselected(DISPLACEMENTS,1);
   HMRES_argumentsetselected(STRESS,1);
 }
 HMRES_argumentprintselected(inputfilename,outputfilename,NULL);
 /*
    Records in the ASCII results file are formatted so that
    the first 8 characters in a line contain a field
    identifying the type of results or information contained
    on the line.  The keys and format of the cards are listed
    below:

      <234567><234567><234567><234567><234567><234567>
      SUBCASE       ID

      RESULTS NAME

      DISPLACE NODE ID  X DISP  Y DISP  Z DISP

      ESTRESS  ELEM ID  STRESS

      NSTRESS  NODE ID  STRESS

    Read the first eight characters of each line and branch
    on key type until the end of the file is reached.

    Open the input file.  Note the usage of
    HMRES_openinputfile(), which loads from stdin if the user
    has selected this option.
*/
 inputfile = HMRES_openinputfile(inputfilename,"rt");
 while (!feof(inputfile))
 {
   HM_asciifile_readfixedstring(inputfile,string,8);
```

```
if (!strcmp(string,"SUBCASE "))
{
  /*
     Read the subcase ID, create a simulation for the subcase,
     and open it for results input.
 */
  id = (HM_entityidtype) HM_asciifile_readfixedint(inputfile,8);
  sprintf(subcasename,"Subcase %d",id);
  HMRES_simulationcreate(subcasename,id);
  HMRES_simulationopen(subcasename,id);
  HM_asciifile_readeol(inputfile);
}
else if (!strcmp(string,"RESULTS "))
{
  /*
     Read the name of the results that follow
     and set the data type open flag to zero.
 */
  HM_asciifile_readfixedstring(inputfile,datatypename,72);
  datatypeopen = 0;
  HM_asciifile_readeol(inputfile);
}
else if (!strcmp(string,"DISPLACE"))
{
  /*
     If DISPLACEMENTS is selected for output, create and
     open the datatype.  The node ID x, y, and z displacements
     are then read and placed into the open datatype.
 */
  if (datatypeopen == 0)
  {
    if (HMRES_argumentgetselected(DISPLACEMENTS))
    {
      HMRES_datatypecreate(datatypename,HMRES_NODALDISPLACEMENTS);
      HMRES_datatypeopen(datatypename);
      datatypeopen = 1;
    }
  }
  id = (HM_entityidtype) HM_asciifile_readfixedint(inputfile,8);
  x = HM_asciifile_readfixeddouble(inputfile,8);
  y = HM_asciifile_readfixeddouble(inputfile,8);
  z = HM_asciifile_readfixeddouble(inputfile,8);
```

```
  if (datatypeopen) HMRES_displacementadd (id,x+DisplacementOffset,
                                           y+DisplacementOffset,
                                           z+DisplacementOffset);
  HM_asciifile_readeol(inputfile);
}
else if (!strcmp(string,"ESTRESS "))
{
  /*
     If STRESS is selected for output, create and open the
     datatype.  The element ID and value are then read
     and placed into the open datatype.
 */
  if (datatypeopen == 0)
  {
    if (HMRES_argumentgetselected(STRESS))
    {
      HMRES_datatypecreate(datatypename,HMRES_ELEMENTVALUES);
      HMRES_datatypeopen(datatypename);
      datatypeopen = 1;
    }
  }
  id = (HM_entityidtype) HM_asciifile_readfixedint(inputfile,8);
  value = HM_asciifile_readfixeddouble(inputfile,8);
  if (datatypeopen) HMRES_valueadd(id,value);
  HM_asciifile_readeol(inputfile);
}
else if (!strcmp(string,"NSTRESS "))
{
  /*
     If STRESS is selected for output, create and open the
     datatype.  The element ID and value are then read
     and placed into the open datatype.
 */
  if (datatypeopen == 0)
  {
    if (HMRES_argumentgetselected(STRESS))
    {
      HMRES_datatypecreate(datatypename,HMRES_NODALVALUES);
      HMRES_datatypeopen(datatypename);
      datatypeopen = 1;
    }
  }
```

```
      id = HM_asciifile_readfixedint(inputfile,8);
      value = HM_asciifile_readfixeddouble(inputfile,8);
      if (datatypeopen) HMRES_valueadd(id,value);
      HM_asciifile_readeol(inputfile);
    }
    else
    {
      fprintf(stderr,"Unknown record in file: '%s'.\n",string);
      HM_asciifile_readeol(inputfile);
    }
  }
  fclose(inputfile);
  HMRES_writeresults(outputfilename);
  HMRES_close();
}
```

Introduction to hmmodlib

hmmodlib is a library of C routines which allows you to perform results translation of more complex data types, such as integration point results or node-on-element results. The basic concept behind hmmodlib is that you create a model first, with the use of hmmodlib routines. This involves sending both node and element information to hmmodlib. After a model is created in hmmodlib, results can be stored on the model, averaged, and then sent to hmreslib. In this way, hmmodlib can be thought of as a pre-processor to hmreslib.

Creating Models

Models are created in hmmodlib by using the functions HMMOD_nodeadd(), HMMOD_elementadd(), and HMMOD_elementaddnode(). HMMOD_nodeadd() is called once for each node in the model, HMMOD_elementadd() is called once for each element, and HMMOD_elementaddnode() is called once for every node on every element.

Each node created in hmmodlib has a data structure associated with it. The following is a description of the node's data structure:

id	ID of the node.
x, y, z	x, y, and z location.
value	The value is used as a temporary storage area for one result. Often, the value is set by an averaging routine, and once set, can be transferred to hmreslib or compared with the maximum result.
maximum	The maximum is used as a temporary storage area for one result. It is used to store the maximum value at a node, and once set, can be transferred to hmreslib.
displacement	Storage for one displacement (x, y, z).
results	An array of user-defined size that stores results at the node. This component of the node structure exists at every layer defined in the model.
counter	A counter used during averaging.
active	An activity flag used to determine if the results at a node should be averaged or transferred to hmreslib. If the active flag is set to a nonzero value, then the averaging routines use the node when averaging is performed, and the results at the node are transferred to hmreslib when requested. If the activity flag is set to zero, the node is ignored in both the aforementioned cases.
uservalue	Storage for a user-defined value.

Each element created in hmmodlib has a data structure associated with it. The following is a description of the element's data structure:

id	ID of the element.
config	The configuration of the element.

type	The type of the element. This is used only if the model is written.
nodes	The nodes associated with an element.
Numberofnodes	The number of nodes associated with an element.
value	The value is used as a temporary storage area for one result. Often, the value is set by an averaging routine, and once set, can be transferred to `hmreslib` or compared with the maximum.
maximum	The maximum is used as a temporary storage area for one result. It is used to store the maximum value on an element, and once set, can be transferred to `hmreslib`.
centroidal	An array of user-defined size that stores results at the centroid of the element. This component of the element structure exists at every layer defined in the model.
integration pt	An array of user-defined size that stores results at the integration points of an element. This component of the element structure exists at every layer defined in the model.
node on elem	An array of user-defined size that stores results at the node of the element. This component of the element structure exists at every layer defined in the model.
# of int pts	The number of integration points for the element.
counter	A counter used during averaging.
active	An activity flag used to determine if the results at an element should be averaged or transferred to `hmreslib`. If the active flag is set to a nonzero value, then the averaging routines use the element when averaging is performed, and the results at the element are transferred to `hmreslib` when requested. If the activity flag is set to zero, the element is ignored in both the aforementioned cases.
uservalue	Storage for a user-defined value.

After the model is created, it can be written to disk in HyperMesh ASCII format by calling the function `HMMOD_writemodel()`.

Storing Results

After the model is created, `hmmodlib` requires that you call `HMMOD_storeresults()` before calling any results storage routines. This function allocates the memory required to store results, based on the calls made during the model creation phase.

You are now ready to store results on the model. For nodes, the following results can be stored:

displacement A displacement (x, y, z) can be stored at each node.

results Values can be stored at the centroid of the node. The number of values that can be stored is a user-defined number available at every layer in the model.

value A value can be stored at the value of the node.

For elements, the following results can be stored:

centroidal Values can be stored at the centroid of the element. The number of values that can be stored is a user-defined number available at every layer in the model.

integration pt Values can be stored at an integration point on the element. The number of values that can be stored is a user-defined number available at every layer in the model.

node on element Values can be stored at a node on the element. The number of values that can be stored is a user-defined number available at every layer in the model.

value A value can be stored at the value of the element.

hmmodlib Functions

The `hmmodlib` functions are used to create models, store results, and transfer results to `hmreslib`.

Access Functions

Averaging Results

Compare Functions

Model Building Functions

Scanning Functions

Storing Results

Transfering Results to hmreslib

Utilities

Access Functions

HMMOD_elementfindifexists()

HMMOD_elementgetactive()

HMMOD_elementgetcentroidalresult()

HMMOD_elementgetconfig()

HMMOD_elementgetcounter()

HMMOD_elementgetid()

HMMOD_elementgetintegrationpointresult()

HMMOD_elementgetnode()

HMMOD_elementgetnodeonelementresult()

HMMOD_elementgetnumberofintegrationpoints()

HMMOD_elementgetnumberofnodes()

HMMOD_elementgettype()

HMMOD_elementgetuservalue()

HMMOD_elementgetvalue()

HMMOD_elementsetactive()

HMMOD_elementsetcounter()

HMMOD_elementsetuservalue()

HMMOD_nodefindifexists()

HMMOD_nodegetactive()

HMMOD_nodegetcords()

HMMOD_nodegetcounter()

HMMOD_nodegetdisplacement()

HMMOD_nodegetid()

HMMOD_nodegetresult()

HMMOD_nodegetuservalue()

HMMOD_nodegetvalue()

HMMOD_nodesetactive()

HMMOD_nodesetcounter()

HMMOD_nodesetuservalue()

HMMOD_elementfindifexists()

Retrieves the pointer to an element if it exists.

Syntax void * HMMOD_elementfindifexists(HM_entityidtype id);

id The ID of the element to be located.

Returns A pointer to the element identified by *id*.

HMMOD_elementgetactive()

Retrieves the activity status of an element.

Syntax int HMMOD_elementgetactive(void * elementptr);

elementptr The pointer to an element.

Returns 0, if the element is inactive.

1, if the element is active.

HMMOD_elementgetcentroidalresult()

Retrieves the centroidal result of an element specified by element pointer, layer, and index.

Syntax float HMMOD_elementgetcentroidalresult(void * elementptr, int layer, int index);

elementptr The pointer to the element.

layer The layer where the centroidal result is stored.

index The index to the array of centroidal results.

Returns The result at the centroid of the specified element.

HMMOD_elementgetconfig()

Retrieves the configuration of an element specified by a pointer to that element.

Syntax int HMMOD_elementgetconfig(void * elementptr);

elementptr The pointer to the element.

Returns The configuration of the specified element.

HMMOD_elementgetcounter()

Retrieves the value stored in counter.

Syntax int HMMOD_elementgetcounter(void * elementptr);

elementptr The pointer to the element.

Returns The value of counter.

HMMOD_elementgetid()

Retrieves the ID of an element specified by the pointer to the element.

Syntax HM_entityidtype HMMOD_elementgetid(void * elementptr);

elementptr The pointer to the element.

Returns The ID of the specified element.

HMMOD_elementgetintegrationpointresult()

Retrieves the integration point result of an element specified by element pointer, integration point, layer and index.

Syntax float HMMOD_elementgetintegrationpointresult(void * elementptr, int layer, int integrationpoint, int index);

elementptr The pointer to the element.

layer The layer where the integration point result is.

integrationpoint The integration point where the result is.

index The index to the array of integration point results.

Returns The integration point result at the specified element.

HMMOD_elementgetnode()

Retrieves a pointer to a node.

Syntax void * HMMOD_elementgetnode(void * elementptr, int nodeindex);

elementptr The pointer to the element.

nodeindex The index to the array of nodes.

Returns The pointer to the specified node.

HMMOD_elementgetnodeonelementresult()

Retrieves the node-on-element result of an element specified by element pointer, layer, node index, and index.

Syntax float HMMOD_elementgetnodeonelementresult(void * elementptr, int layer, int nodeindex, int index);

elementptr The pointer to the element.

layer The layer where the node-on-element point result is stored.

nodeindex The node-on-element point where the result is stored.

index The index to the array of node-on-element point results.

Returns The result at the node-on-element point of the specified element.

HMMOD_elementgetnumberofintegrationpoints()

Retrieves the number of integration points in an element specified by a pointer to that element.

Syntax int HMMOD_elementgetnumberofintegrationpoints(void * elementptr);

elementptr The pointer to the element.

Returns The number of integration points in the specified element.

HMMOD_elementgetnumberofnodes()

Retrieves the number of nodes associated with an element specified by a pointer to that element.

Syntax int HMMOD_elementgetnumberofnodes(void * elementptr);

elementptr The pointer to the element.

Returns The number of nodes in the specified element.

HMMOD_elementgettype()

Retrieves the type of an element specified by a pointer to that element.

Syntax int HMMOD_elementgettype(void * elementptr);

elementptr The pointer to the element.

Returns The type of the element specified.

HMMOD_elementgetuservalue()

Retrieves the user value of an element.

Syntax int HMMOD_elementgetuservalue(void * elementptr);

elementptr The pointer to an element.

Returns The value of the specified element.

HMMOD_elementgetvalue()

Retrieves the value of an element specified by element pointer.

Syntax float HMMOD_elementgetvalue(void * elementptr);

elementptr The pointer to an element.

Returns The value of an element.

HMMOD_elementsetactive()

Sets the status of an element to active or inactive.

Syntax void HMMOD_elementsetactive(void * elementptr, int active);

elementptr The pointer to the element.

active The status of the element. Settings are:

0 Inactive.

1 Active.

Returns Nothing.

Comments This function allows you to determine which elements in the model are used for averaging; all elements with an active status are used. The default setting for elements is *active*.

HMMOD_elementsetcounter()

Assigns a value to counter.

Syntax void HMMOD_elementsetcounter(void * elementptr, int counter);

elementptr The pointer to the element.

counter The value to be assigned to counter.

Returns Nothing.

HMMOD_elementsetuservalue()

Modifies the value of an element.

Syntax void HMMOD_elementsetuservalue(void * elementptr, int uservalue);

elementptr The pointer to an element.

uservalue The user-specified value to be assigned to the element.

Returns Nothing.

HMMOD_nodefindifexists()

Retrieves the pointer to a node if it exists.

Syntax void * HMMOD_nodefindifexists(HM_entityidtype id);

id The ID of the node to be located.

Returns A pointer to the node identified by *id*.

Comments This function does not issue error messages when a node is not found and in this case returns NULL. You must check for a NULL pointer before using the return value.

HMMOD_nodegetactive()

Retrieves the activity status of a node.

Syntax int HMMOD_nodegetactive(void * nodeptr);

nodeptr The pointer to an node.

Returns 0, if the node is inactive.

1, if the node is active.

HMMOD_nodegetcords()

Retrieves the coordinates of a node.

Syntax void HMMOD_nodegetcords(void * nodeptr, float * x, float * y, float * z);

nodeptr	The pointer to the node.
x	The x coordinate of the node.
y	The y coordinate of the node.
z	The z coordinate of the node.

Returns Nothing.

Comments The x, y, and z variables are set to the coordinates of the specified node.

HMMOD_nodegetcounter()

Retrieves the value stored in counter.

Syntax int HMMOD_nodegetcounter(void * nodeptr);

nodeptr	The pointer to the node.

Returns The value of counter.

Comments This function allows you to use the value stored in the counter to average element values back to the node.

HMMOD_nodegetdisplacement()

Retrieves the displacement values of the node specified by the node pointer.

Syntax void HMMOD_nodegetdisplacement(void * nodeptr, float * x, float * y, float * z);

nodeptr	The pointer to the node.
x	The displacement value at the x coordinate.
y	The displacement value at the y coordinate.
z	The displacement value at the z coordinate.

Returns Nothing.

Comments The values of the x, y, and z variables are set to the displacement values of the specified node.

HMMOD_nodegetid()

Retrieves a node ID.

Syntax HM_entityidtype HMMOD_nodegetid(void * nodeptr);

nodeptr The pointer to the node.

Returns The ID of the specified node.

HMMOD_nodegetresult()

Retrieves the nodal result of the node specified by the node pointer, layer, and index.

Syntax float HMMOD_nodegetresult(void * nodeptr, int layer, int index);

nodeptr The pointer to the node.

layer The layer where the nodal result is stored.

index The index of nodal results.

Returns The nodal result at the specified node.

HMMOD_nodegetuservalue()

Retrieves the user value of a node.

Syntax int HMMOD_nodegetuservalue(void * nodeptr);

nodeptr The pointer to the node.

Returns The value of the specified node.

HMMOD_nodegetvalue()

Retrieves the value associated with a node specified by the pointer to the node.

Syntax float HMMOD_nodegetvalue(void * nodeptr);

nodeptr The pointer to a node.

Returns The value at the specified node.

HMMOD_nodesetactive()

Sets the status of a node to active or inactive.

Syntax void HMMOD_nodesetactive(void * nodeptr, int active);

nodeptr The pointer to the node.

active The status of the node. Settings are:

0 Inactive.

1 Active.

Returns Nothing.

Comments This function allows you to determine which nodes in the element are used for averaging; all nodes with an active status are used. The default setting for nodes is active.

HMMOD_nodesetcounter()

Assigns a value to the counter.

Syntax void HMMOD_nodesetcounter(void * nodeptr, int counter);

nodeptr The pointer to the node.

counter The value to be assigned to the counter.

Returns Nothing.

HMMOD_nodesetuservalue()

Modifies the value of a node.

Syntax void HMMOD_nodesetuservalue(void * nodeptr, int uservalue);

nodeptr The pointer to the node.

uservalue The user-specified value to be assigned to the node.

Returns Nothing.

Averaging Results

`HMMOD_elementintegrationpointsresultsaveragetoelementvalues()`

`HMMOD_elementnodeonelementresultsaveragetonodevalue()`

HMMOD_elementintegrationpointresultsaveragetoelementvalues()

Averages the element integration point results to element values.

Syntax void HMMOD_elementintegrationpointresultsaveragetoelementvalues(int layer, int index);

layer The layer of the element whose integration point results are to be averaged.

index The index into the array of element integration points associated with each element.

Returns Nothing.

Comments This function takes all of the integration points assigned to an element, averages them to the centroid of the element, and stores that result in the element values. Element integration point results must be averaged before they can be transferred to `hmreslib`.

HMMOD_elementnodeonelementresultsaveragetonodevalue()

Averages the element node-on-element results to nodal values.

Syntax void HMMOD_elementnodeonelementresultsaveragetonodevalue(int layer, int index);

layer The layer of the element whose node-on-element results are to be averaged.

index The index into the array of element node-on-element results associated with each element.

Returns Nothing.

Comments This function takes all of the node-on-element results assigned to an element, averages them to the node, and stores that result in the node values. Node-on-element results must be averaged (except when there is only one value associated with the node-on-element results) before they can be transferred to `hmreslib`.

Compare Functions

`HMMOD_elementmaximumscompareelementvalues()`

`HMMOD_nodemaximumscomparenodevalues()`

HMMOD_elementmaximumscompareelementvalues()

Compares the value associated with an element to the maximum of the element. If the value is greater than the maximum, it sets the maximum equal to the value.

Syntax void HMMOD_elementmaximumscompareelementvalues(void);

Returns Nothing.

HMMOD_nodemaximumscomparenodevalues()

Compares the value associated with a node to the maximum of the node. If the value is greater than the maximum, it sets the maximum equal to the value.

Syntax void HMMOD_nodemaximumscomparenodevalues(void);

Returns Nothing.

Model Building Functions

`HMMOD_elementadd()`

`HMMOD_elementaddnode()`

`HMMOD_nodeadd()`

HMMOD_elementadd()

Adds elements to the database.

Syntax void HMMOD_elementadd(HM_entityidtype id, unsigned char config, unsigned char type, int numintegratepts, unsigned char uservalue);

id	The ID of the element.
config	The configuration of the element.
type	The type of element in HyperMesh.

	numintegratepts	The number of integration points for this element.
	uservalue	User-determined value.

Returns Nothing.

HMMOD_elementaddnode()

Adds a node to an element.

Syntax void HMMOD_elementaddnode(int index, HM_entityidtype nodeid);

index	The index into the array of nodes in the element. The index starts at zero and goes to the number of nodes in the element minus one.
nodeid	The entity ID.

Returns Nothing.

HMMOD_nodeadd()

Adds nodes to the database.

Syntax void HMMOD_nodeadd(HM_entityidtype id, float x, float y, float z, unsigned char uservalue);

id	The ID of the node.
x	The x value of its location in space.
y	The y value of its location in space.
z	The z value of its location in space.
uservalue	The value to be assigned to the node.

Returns Nothing.

Scanning Functions

`HMMOD_elementfind()`

`HMMOD_elementgetfirst()`

`HMMOD_elementgetnext()`

`HMMOD_nodefind()`

`HMMOD_nodegetfirst()`

`HMMOD_nodegetnext()`

HMMOD_elementfind()

Retrieves the pointer to an element.

Syntax void * HMMOD_elementfind(HM_entityidtype id);

id The ID of the element to be located.

Returns A pointer to the element identified by *id*.

Comments This function does not issue error messages when an element is not found and in this case returns NULL. You must check for a NULL pointer before using the return value.

HMMOD_elementgetfirst()

Retrieves a pointer to the first element stored in `hmmodlib`.

Syntax void * HMMOD_elementgetfirst(void);

Returns A pointer to the first element.

Comments This function, in combination with `HMMOD_elementgetnext()`, allows the elements in the model to be scanned sequentially.

HMMOD_elementgetnext()

Retrieves a pointer to the next element stored in `hmmodlib`.

Syntax void * HMMOD_elementgetnext(void);

Returns A pointer to the next element.

Comments This function, in combination with `HMMOD_elementgetfirst()`, allows the elements in the model to be scanned sequentially.

HMMOD_nodefind()

Retrieves the pointer to a node.

Syntax void * HMMOD_nodefind(HM_entityidtype id);

id The ID of the node to be located.

Returns A pointer to the node identified by *id*.

HMMOD_nodegetfirst()

Retrieves a pointer to the first node stored in `hmmodlib`.

Syntax void * HMMOD_nodegetfirst(void);

Returns A pointer to the first node.

Comments This function, in combination with `HMMOD_nodegetnext()`, allows the nodes in the model to be scanned sequentially.

HMMOD_nodegetnext()

Retrieves a pointer to the next node stored in `hmmodlib`.

Syntax void * HMMOD_nodegetnext(void);

Returns A pointer to the next node.

Comments This function, in combination with `HMMOD_nodegetfirst()`, allows the elements in the model to be scanned sequentially.

Storing Results

`HMMOD_elementidsetcentroidalresult()`

`HMMOD_elementidsetintegrationpointresult()`

`HMMOD_elementidsetnodeonelementresult()`

`HMMOD_elementidsetvalue()`

`HMMOD_elementsetcentroidalresult()`

`HMMOD_elementsetintegrationpointresult()`

`HMMOD_elementsetnodeonelementresult()`

`HMMOD_elementsetvalue()`

`HMMOD_nodeidsetdisplacement()`

`HMMOD_nodeidsetresult()`

`HMMOD_nodeidsetvalue()`

`HMMOD_nodesetdisplacement()`

`HMMOD_nodesetresult()`

`HMMOD_nodesetvalue()`

`HMMOD_nodevaluestoreresult()`

`HMMOD_storeresults()`

HMMOD_elementidsetcentroidalresult()

Assigns a centroidal result for an element specified by ID, layer, and index.

Syntax void HMMOD_elementidsetcentroidalresult(HM_entityidtype id, int layer, int index, float value);

id	The ID of the element.
layer	The layer where the centroidal result should be stored.
index	The index into the array of centroidal results.
value	The value to be assigned to the centroidal result at the specified *id*.

Returns Nothing.

HMMOD_elementidsetintegrationpointresult()

Assigns a value to the result at an integration point of an element specified by ID, layer, integration point, and index.

Syntax void HMMOD_elementidsetintegrationpointresult(HM_entityidtype id, int layer, int integrationpoint, int index, float value);

id	The ID of the element.
layer	The layer where the integration point result should be stored.
integrationpoint	The integration point where the result should be stored.
index	The index into the array of integration point results.
value	The value to be assigned to the result at the integration point.

Returns Nothing.

HMMOD_elementidsetnodeonelementresult()

Assigns a value to the result at a node-on-element point of an element specified by ID, layer, node index, and index.

Syntax void HMMOD_elementidsetnodeonelementresult(HM_entityidtype id, int layer, int nodeindex, int index, float value);

id	The ID of the element.
layer	The layer where the node-on-element point result should be stored.
nodeindex	The node-on-element point where the result should be stored.
index	The index into the array of node-on-element point results.
value	The value to be assigned to the result at the node-on-element point.

Returns Nothing.

HMMOD_elementidsetvalue()

Assigns a value to an element identified by ID.

Syntax void HMMOD_elementidsetvalue(HM_entityidtype id, float value);

id	The ID of the element.
value	The value to be assigned to the element at the specified *id*.

Returns Nothing.

HMMOD_elementsetcentroidalresult()

Assigns a value to the result at the centroid of an element specified by element pointer, layer, and index.

Syntax void HMMOD_elementsetcentroidalresult(void * elementptr, int layer, int index, float value);

elementptr	The pointer to the element.
layer	The layer where the centroidal result is to be stored.
index	The index to the array of centroidal results.
value	The value to be assigned to the centroidal result of the specified element.

Returns Nothing.

HMMOD_elementsetintegrationpointresult()

Assigns a value to the integration point result of an element specified by element pointer, integration point, layer and index.

Syntax void HMMOD_elementsetintegrationpointresult(void * elementptr, int layer, int integrationpoint, int index, float value);

elementptr	The pointer to the element.
layer	The layer where integration point result should be stored.
integrationpoint	The integration point where the result should be stored.
index	The index to the array of integration point results.
value	The value to be assigned to the integration point of the specified element.

Returns Nothing.

HMMOD_elementsetnodeonelementresult()

Assigns a value to the result at a node-on-element point of an element specified by element pointer, layer, node index, and index.

Syntax void HMMOD_elementsetnodeonelementresult(void * elementptr, int layer, int nodeindex, int index, float value);

elementptr	The pointer to the element.
layer	The layer where the node-on-element point result should be stored.
nodeindex	The node-on-element point where the result should be stored.
index	The index to the array of node-on-element point results.
value	The value to be assigned to the result at the node-on-element point.

Returns Nothing.

HMMOD_elementsetvalue()

Assigns a value to an element specified by element pointer.

Syntax void HMMOD_elementsetvalue(void * elementptr, float value);

elementptr	The pointer to the element.
value	The value to be assigned to the element.

Returns Nothing.

HMMOD_nodeidsetdisplacement()

Assigns displacement values to a node specified by ID.

Syntax void HMMOD_nodeidsetdisplacement(HM_entityidtype id, float x, float y, float z);

id	The ID of the node.
x	The displacement value to be assigned to the x coordinate.
y	The displacement value to be assigned to the y coordinate.
z	The displacement value to be assigned to the z coordinate.

Returns Nothing.

HMMOD_nodeidsetresult()

Assigns a nodal result value to a node specified by ID, layer, and index.

Syntax void HMMOD_nodeidsetresult(HM_entityidtype id, int layer, int index, float value);

id	The ID of the node.
layer	The layer where the nodal result should be stored.
index	The index of nodal results.
value	The value to be assigned to the nodal result at the specified node ID.

Returns Nothing.

HMMOD_nodeidsetvalue()

Assigns a value to a node specified by ID.

Syntax void HMMOD_nodeidsetvalue(HM_entityidtype id, float value);

id	The ID of the node.
value	The value to be assigned to the specified node.

Returns Nothing.

HMMOD_nodesetdisplacement()

Assigns displacement values to a node specified by the node pointer.

Syntax void HMMOD_nodesetdisplacement(void * nodeptr, float x, float y, float z);

nodeptr	The pointer to the node.
x	The displacement value to be assigned to the x coordinate.
y	The displacement value to be assigned to the y coordinate.
z	The displacement value to be assigned to the z coordinate.

Returns Nothing.

HMMOD_nodesetresult()

Assigns a nodal result to a node specified by the node pointer, layer and index.

Syntax void HMMOD_nodesetresult(void * nodeptr, int layer, int index, float value);

nodeptr	The pointer to the node.
layer	The layer where the nodal result should be stored.
index	The index of nodal results.
value	The value to be assigned to the nodal result at the specified node.

Returns Nothing.

HMMOD_nodesetvalue()

Assigns a value to a node specified by the node pointer.

Syntax void HMMOD_nodesetvalue(void * nodeptr, float value);

nodeptr	The pointer to the node.
value	The value to be assigned to the specified node.

Returns Nothing.

HMMOD_nodevaluestoreresult()

Stores the user value as a result for the given layer

Syntax void HMMOD_nodevaluestoreresult(int layer,int index);

layer	The layer where the nodal result should be stored.
index	The index of nodal results.

Returns Nothing.

HMMOD_storeresults()

Indicates to `hmmodlib` that all nodes and elements have been added to the database. This function must be called before storing the results data.

Syntax void HMMOD_storeresults(void);

Returns Nothing.

Transferring Results to hmreslib

`HMMOD_elementcentroidalresultstohmreslib()`

`HMMOD_elementmaximumstohmreslib()`

`HMMOD_elementvaluestohmreslib()`

`HMMOD_nodedisplacementstohmreslib()`

`HMMOD_nodemaximumstohmreslib()`

`HMMOD_noderesultstohmreslib()`

`HMMOD_nodevaluestohmreslib()`

HMMOD_elementcentroidalresultstohmreslib()

Transfers the current values associated with the element centroidal results to `hmreslib`.

Syntax void HMMOD_elementcentroidalresultstohmreslib(int layer, int index);

layer The layer of the element whose centroidal values are to be transferred.

index The index into the array of centroidal results associated with each element.

Returns Nothing.

HMMOD_elementmaximumstohmreslib()

Transfers the current values associated with the element maximums to `hmreslib`.

Syntax void HMMOD_elementmaximumstohmreslib(void);

Returns Nothing.

HMMOD_elementvaluestohmreslib()

Transfers the current values associated with the elements to `hmreslib`.

Syntax void HMMOD_elementvaluestohmreslib(void);

Returns Nothing.

HMMOD_nodedisplacementstohmreslib()

Transfers the current values associated with the node displacements to `hmreslib`.

Syntax void HMMOD_nodedisplacementstohmreslib(void);

Returns Nothing.

HMMOD_nodemaximumstohmreslib()

Transfers the current maximums associated with the nodes to `hmreslib`.

Syntax void HMMOD_nodemaximumstohmreslib(void);

Returns Nothing.

HMMOD_noderesultstohmreslib()

Transfers the current values associated with the node results to `hmreslib`.

Syntax void HMMOD_noderesultstohmreslib(int layer, int index);

layer	The layer of the element whose nodal results are to be transferred.
index	The index into the array of node results associated with each node.

Returns Nothing.

HMMOD_nodevaluestohmreslib()

Transfers the current values associated with the nodes to `hmreslib`.

Syntax void HMMOD_nodevaluestohmreslib(void);

Returns Nothing.

Utilities

`HMMOD_close()`

`HMMOD_elementdividecounterintovalue()`

`HMMOD_elementmaximumsreset()`

`HMMOD_elementresetvalueandcounter()`

`HMMOD_initialize()`

`HMMOD_nodedividecounterintovalue()`

`HMMOD_nodemaximumsreset()`

`HMMOD_noderesetvalueandcounter()`

`HMMOD_writemodel()`

HMMOD_close()

Closes `hmmodlib`.

Syntax void HMMOD_close(void);

Returns Nothing.

Comments This function must be called to free the memory used by `hmmodlib`.

HMMOD_elementdividecounterintovalue()

Divides the value field in the element structure by the counter field.

Syntax void HMMOD_elementdividecounterintovalue(void);

Returns Nothing.

HMMOD_elementmaximumsreset()

Assigns all of the maximums associated with all the elements in the model to a user-specified value.

Syntax void HMMOD_elementmaximumsreset(float value);

value The value at which the element maximum should be reset.

Returns Nothing.

HMMOD_elementresetvalueandcounter()

Resets the value and counter values in the element data structure.

Syntax void HMMOD_elementresetvalueandcounter(void);

Returns Nothing.

HMMOD_initialize()

Initializes `hmmodlib`.

Syntax void HMMOD_initialize(int layers, int node, int ecentroidal, int nodeonelement, int eintegrationpts);

layers The number of layers that `hmmodlib` should allocate for an element in the model.

node The number of nodal result values to be stored, per layer, for each node of the model.

ecentroidal The number of element centroidal points, per layer, per element, which should be stored at each element centroid.

nodeonelement The number of node-on-element results, per layer, which should be stored at each node for each element.

	eintegrationpts	The number of values that should be stored, per layer, per element integration point.

Returns Nothing.

Comments This function must be called before any other function in the `hmmodlib` package.

HMMOD_nodedividecounterintovalue()

Divides the value field in the node structure by the counter field.

Syntax void HMMOD_nodedividecounterintovalue(void);

Returns Nothing.

HMMOD_nodemaximumsreset()

Assigns all of the maximums associated with all the nodes in the model to a user-specified value.

Syntax void HMMOD_nodemaximumsreset(float value);

	value	The value at which the node maximum should be reset.

Returns Nothing.

HMMOD_noderesetvalueandcounter()

Resets the value and counter values in the node data structure.

Syntax void HMMOD_noderesetvalueandcounter(void);

Returns Nothing.

HMMOD_writemodel()

Writes a model that can be read by HyperMesh.

Syntax void HMMOD_writemodel(char * filename);

filename The name of the file where the model should be stored.