2021 R1
Ansys
©2021 ANSYS, Inc.
All Rights Reserved.
Unauthorized use, distribution or duplication is prohibited.

ANSYS Parametric Design Language Guide

ANSYS, Inc.
Southpointe
2600 ANSYS Drive
Canonsburg, PA 15317
ansysinfo@ansys.com
http://www.ansys.com
(T) 724-746-3304
(F) 724-514-9494

Release 2021 R1
January 2021

Table of Contents

List of Figures

List of Tables

Chapter 1: Introducing APDL

APDL stands for ANSYS Parametric Design Language, a scripting language that you can use to automate common tasks or even build your model in terms of parameters (variables). While all Mechanical APDL commands can be used as part of the scripting language, the APDL commands discussed here are the true scripting commands and encompass a wide range of other features such as repeating a command, macros, if-then-else branching, do-loops, and scalar, vector and matrix operations.

While APDL is the foundation for sophisticated features, it also offers many conveniences that you can use in your day-to-day analyses. In this guide we'll introduce you to the basic features - parameters; macros; branching, looping, and repeating; and array parameters - and show you some simple examples. As you become more adept at the language, you will begin to recognize applications for APDL in your own environment.

This guide covers the following topics:

- Working with the toolbar (p. 3): You can add frequently used Mechanical APDL functions or macros to the Mechanical APDL toolbar by defining abbreviations, which are aliases (up to eight characters long) for a Mechanical APDL command, GUI function name, or macro name.
- Using parameters (p. 7): Parameters are APDL variables (they are more similar to FORTRAN variables than to FORTRAN parameters). Mechanical APDL uses two types of parameters: scalar and array.
- Using APDL Math for matrix manipulations (p. 49): APDL Math gives you access to matrix manipulation routines, including fast and efficient solvers. APDL Math provides access to matrices and vectors on the `.FULL`, `.EMAT`, `.MODE` and `.SUB` files, as well as other sources, so that you can read them in, manipulate them, and write them back out or solve them directly. Both dense matrices and sparse matrices can be manipulated using APDL Math.
- Understanding APDL as a macro language (p. 61), including creating a macro: You can record a frequently used sequence of Mechanical APDL commands in a macro file (these are sometimes called command files). Creating a macro enables you to, in effect, create your own custom Mechanical APDL command. In addition to executing a series of Mechanical APDL commands, a macro can call GUI functions or pass values into arguments.
- Interfacing with the GUI: (p. 81) Within a Mechanical APDL macro, you have several ways to access components of the Mechanical APDL GUI (toolbar, dialog box, ***ASK** command, etc.).
- Encrypting macros (p. 89): Mechanical APDL provides the ability to encrypt macro files so that the source is not "human-readable." Encrypted macros require an encryption key to run. You can either place the encryption key explicitly (in readable ASCII) in the macro or you can set it in Mechanical APDL as a global encryption key.

See the APDL Commands (p. 93) for a list of all APDL commands discussed in this guide.

Chapter 2: Working with the Toolbar

You can add frequently used Mechanical APDL functions or macros to the Mechanical APDL toolbar by defining abbreviations, which are aliases (up to eight characters long) for a Mechanical APDL command, GUI function name, or macro name. You can also modify the toolbar and nest toolbar abbreviations,

The following toolbar topics are available:

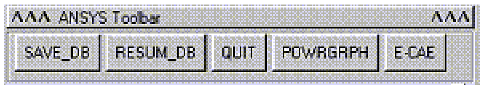
2.1. Adding Commands to the Toolbar

You can add frequently used Mechanical APDL functions or macros to the Mechanical APDL toolbar (creating macros is covered starting in APDL as a Macro Language (p. 61)). You do this by defining *abbreviations*. An abbreviation is an alias (up to eight characters long) for a Mechanical APDL command, GUI function name, or macro name. For example, MATPROP might be an abbreviation for a macro that lists material properties, SAVE_DB is an abbreviation for the **SAVE** command, and **QUIT** is an abbreviation for the **Fnc_/EXIT** function (which launches the Exit from Mechanical APDL dialog box). APDL commands that can be used to define abbreviations are listed in Chapter 2 of the *Command Reference*.

The Mechanical APDL program provides two ways to use abbreviations. You can issue the abbreviation (and execute the macro, command, etc. that it performs) by typing it at the beginning of a command line. If you are using the Mechanical APDL GUI, you can also execute the macro or command by pressing the appropriate button on the Mechanical APDL toolbar.

The toolbar shown in Figure 2.1: Toolbar (p. 3) contains buttons that correspond to existing abbreviations.

Figure 2.1: Toolbar



While some abbreviations, such as SAVE_DB, are predefined, the abbreviations the toolbar contains and the functions they execute are up to you. A single toolbar can hold up to 100 abbreviations (you can "nest" toolbars to extend this number). You can redefine or delete abbreviations at will; however, abbreviations are not automatically saved and must be explicitly saved to a file and reloaded for each Mechanical APDL session.

2.2. Modifying the Toolbar

You can create abbreviations either through the ***ABBR** command or through the **Utility Menu> Macro> Edit Abbreviations** or **Utility Menu> MenuCtrls> Edit Toolbar** menu items. Using one of the menu items is preferable for two reasons:

- Clicking OK automatically updates the toolbar (using the ***ABBR** command requires that you use the **Utility Menu> MenuCtrls> Update Toolbar** menu item to make your new abbreviation appear on the toolbar).
- You can easily edit the abbreviation if required.

The syntax for the ***ABBR** command and related dialogs is

***ABBR**, *`Abbr`*, *`String`*

`Abbr`

The abbreviation name that will appear on the toolbar button. The name can contain up to eight characters.

String

The *String* argument is the name of the macro or command that *Abbr* represents. If *String* is the name of a macro, the macro must be within the macro search path. For more information about using macros, see APDL as a Macro Language (p. 61). If *`String`* references a Mechanical APDL picking menu or dialog box (using UIDL), then specify "Fnc_string." For example, in the abbreviation definitions for "QUIT" and "POWRGRPH" shown above, "Fnc_/QUIT" and "Fnc_/GRAPHICS" are unique UIDL function names which identify the Mechanical APDL picking menu or dialog box associated with the QUIT and POWRGRPH abbreviations respectively. For more information about accessing UIDL functions, see Calling Dialog Boxes From a Macro (p. 88). *`String`* can contain up to 60 characters but cannot include any of the following:

- The character "$"
- The commands **C*****, **/COM**, **/GOPR**, **/NOPR**, **/QUIT**, **/UI**, or ***END**

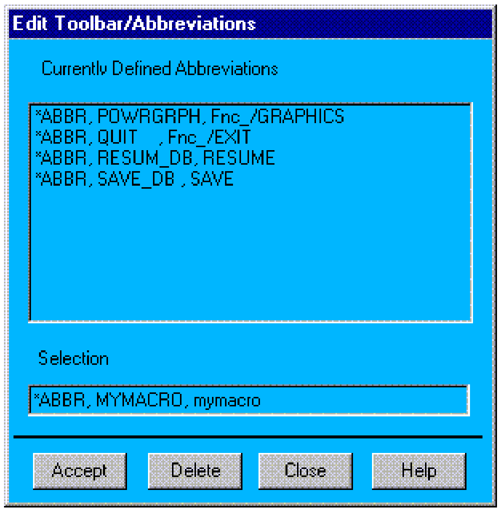
The default Mechanical APDL toolbar has the following abbreviations predefined:

```
*ABBR, SAVE_DB, SAVE
*ABBR, RESUM_DB, RESUME
*ABBR, QUIT, Fnc_/EXIT
*ABBR, POWRGRPH, Fnc_/GRAPHICS
```

2.2.1. Example: Adding a Toolbar Button

For example, to add a button to the toolbar that calls the macro file `mymacro.mac`, you would enter the values shown in the following figure in the **Utility Menu> MenuCtrls> Edit Toolbar** dialog box.

Figure 2.2: Adding a New Abbreviation

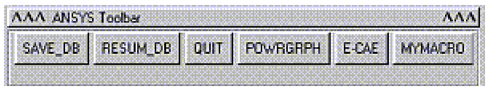


```
*ABBR, POWRGRPH, Fnc_/GRAPHICS
*ABBR, QUIT    , Fnc_/EXIT
*ABBR, RESUM_DB, RESUME
*ABBR, SAVE_DB , SAVE
```

```
*ABBR, MYMACRO, mymacro
```

The new button is appended to the button bar as shown in the following figure.

Figure 2.3: Toolbar with New Button



2.2.2. Saving Toolbar Buttons

Toolbar buttons are not persistent from one Mechanical APDL session to the next; however, they are saved and maintained in the database so that any "resume" of the session will still contain these abbreviations. To save your custom button definitions, you must explicitly save them to a file through the **Utility Menu> MenuCtrls> Save Toolbar** menu item (**ABBSAV** command) and restore them for each session using the **Utility Menu> MenuCtrls> Restore Toolbar** menu item (**ABBRES** command). You can do this programmatically in a macro.

Note:

If any abbreviations already exist in the named file, the **ABBSAV** command overwrites them.

The format of the abbreviations file is the APDL commands that are used to create the abbreviations. Thus, if you wish to edit a large set of buttons or change their order, you may find using a text editor to be the most convenient method. For example, the following is the file that results from saving the default toolbar buttons.

```
/NOPR
*ABB,SAVE_DB ,SAVE
*ABB,RESUM_DB,RESUME
*ABB,QUIT    ,Fnc_/EXIT
*ABB,POWRGRPH,Fnc_/GRAPHICS
/GO
```

The ***ABB** commands (the abbreviated form of ***ABBR**) define the buttons. The **/NOPR** at the top turns off echoing to the log file while the **/GO** at the bottom turns log file echoing on.

2.3. Nesting Toolbar Abbreviations

The save-and-restore features described above allow you to *nest* abbreviations. By nesting abbreviations under one button, you can define specialized toolbars (if you have many abbreviations, having them on a single toolbar can be cluttered, making it difficult to find the proper button). To nest abbreviations, you simply define an abbreviation that restores an abbreviation file. For example, the following command defines PREP_ABR as an abbreviation that restores abbreviations from the file `prep.abbr`.

```
*ABBR,PREP_ABR,ABBRES,,PREP,ABBR
```

PREP_ABR will appear as a button on the toolbar. Clicking it will replace the existing buttons with the set of buttons defined in the `prep.abbr` file.

By defining abbreviations to restore these files and including those abbreviations in the appropriate files, you can have a virtually unlimited number of abbreviations in a given Mechanical APDL session. You can even extend this concept and create your own menu hierarchy by nesting several abbreviation files. If you implement such a hierarchy, it's a good practice to add an abbreviation as a "return" button in each file to navigate back through the menus.

Chapter 3: Using Parameters

Parameters are APDL variables (they are more similar to FORTRAN variables than to FORTRAN parameters). It is not necessary to explicitly declare the parameter type. All numeric values (whether integer or real) are stored as double-precision values. Parameters that are used but not defined are assigned a near-zero, or "tiny," value of approximately 2^{-100}. For example, if parameter A is defined as A=B, and B is not defined, then A is assigned the tiny value.

Mechanical APDL uses two types of parameters: scalar and array. The first part of this chapter discusses information that is applicable to both types. Starting with Array Parameters (p. 19), the information is specific to array type parameters. APDL commands used to define parameters in general and array parameters specifically are listed in Chapter 2 of the *Command Reference*.

Character strings (up to eight characters long) can be assigned to parameters by simply enclosing the string in single quotes. APDL also provides several types of array parameters: numeric, character, string and table (a special numeric type that automatically interpolates values).

You can use a parameter (instead of a literal number or character string) as an argument to any Mechanical APDL command; the parameter is evaluated and its current value is used for that argument. For example, if you assign the value 2.7 to a parameter named AA and then issue the command

```
N,12,AA,4
```

the Mechanical APDL program interprets the command as

```
N,12,2.7,4
```

(which defines node 12 at X=2.7 and Y=4).

Note:

If array, table, or character parameters are used within a macro or input file, those parameters should be dimensioned (if array or table) and defined within that macro or input file. If you fail to follow this practice, Mechanical APDL generates error messages indicating that those parameters are undefined. Mechanical APDL generates the error messages even if the parameters lie within unexecuted ***IF** statements, as parameter substitution occurs before the branching for the ***IF** is checked.

The following parameter topics are available:

3.1. Guidelines for Parameter Names

Parameter names must:

- Begin with a letter
- Contain only letters, numbers, and underscore characters
- Contain no more than 32 characters

Examples of valid and invalid parameter names are

Valid:

ABC
PI
X_OR_Y

Invalid:

MY_PARAMETER_NAME_LONGER_THAN_32_CHARACTERS (more than 32 characters)
2CF3 (begins with a number)
M&E (invalid character "&")

When naming parameters:

- Avoid parameter names that match commonly used Mechanical APDL labels, such as:
  - Degree of freedom (DOF) labels (TEMP, UX, PRES, etc.)
  - Convenience labels (ALL, PICK, STAT, etc.)
  - User-defined labels (such as those defined with the **ETABLE** command)
  - Array type field labels (such as CHAR, ARRAY, TABLE, etc.)
- Parameter names ARG1 through ARG9 and AR10 through AR99 are reserved for local parameters. Generally, local parameters are used in macros (see Local Variables (p. 69)). Use of these names as "regular" parameters is not recommended.
- Parameter names must not match abbreviations defined with the ***ABBR** command. For more information about abbreviations, see Adding Commands to the Toolbar (p. 3).

- Do not begin parameter names with an underscore (_). This convention is reserved for parameters used by the GUI and Mechanical APDL-provided macros.
- APDL programmers supporting an organization should consider naming their parameters with a trailing underscore(_). These can be displayed as a group using the ***STATUS** command and deleted from memory as a group through the ***DEL** command.
- Older Mechanical APDL-provided macro files may use parameter names that do not begin with an underscore. Using these macros embedded in your own macros may cause conflicts if the same parameter names are used.

3.1.1. Hiding Parameters from *STATUS

Listing Parameters (p. 12) discusses listing parameters through the ***STATUS** command. You can use a parameter naming convention to "hide" parameters from the ***STATUS** command. Any parameter whose name ends in an underscore (_) is not listed by ***STATUS**.

This capability was added specifically for those who are developing APDL macros for large audiences. You can use this to build macros that your Mechanical APDL users and other macro programmers cannot list.

3.2. Defining Parameters

Unless otherwise specified, the information in the next several sections applies to both scalar and array type parameters. Beginning with Array Parameters (p. 19), the information is specific to array type parameters.

You can either assign values to parameters or retrieve values provided by Mechanical APDL and store these values in parameters. For retrieving values from Mechanical APDL, you can use either the ***GET** command or the various in-line get functions. The following sections cover these subjects in detail.

3.2.1. Assigning Parameter Values During Execution

You can use the ***SET** command to define parameters. The following examples illustrate a set of example parameters defined using ***SET**:

```
*SET,ABC,-24
*SET,QR,2.07E11
*SET,XORY,ABC
*SET,CPARM,'CASE1'
```

You can use an "=" as a shorthand way of calling the ***SET** command (this is the most convenient method). The format of the shortcut is *Name* = *Value*, where *Name* is the name assigned to the parameter and *Value* is the numeric or character value stored in that parameter. For character parameters, the assigned value must be enclosed in single quotes and cannot exceed eight alphanumeric characters. The following are examples of "=" in use:

```
ABC=-24
QR=2.07E11
XORY=ABC
CPARM='CASE1'
```

In the GUI, you can either type the "=" directly in the Mechanical APDL input window or in the "Selection" field of the Scalar Parameter dialog (**Utility Menu> Parameters> Scalar Parameters**).

3.2.2. Assigning Parameter Values at Startup

You can define parameters as arguments when launching Mechanical APDL from the operating system command line. Simply type parameter definitions after the Mechanical APDL execution command (which is system dependent) using the format *-Name Value*. For example, the following defines two parameters (parm1 and parm2) having the values 89.3 and -0.1:

```
ansys211 -parm1 89.3 -parm2 -0.1
```

It is a good practice to avoid assigning one or two character parameter names at startup to avoid conflicts with Mechanical APDL command line options.

Note:

Linux shells treat single quotes and many other non-alphanumeric characters as special characters. When defining character parameters, set Linux not to interpret the quotes by inserting a backslash (\) before the single quotes. For example, the following defines a character parameter having the value `filename`.

```
ansys211 -cparm1 \'filename\'
```

If you use the Mechanical APDL Launcher to start Mechanical APDL, you can define parameters through the **Customization** tab (using the *-Name Value* format described above).

If you are defining a large number of parameters at startup, you'll find it much more convenient to define these in the `start.ans` file or through a separate file that you can load through the **/INPUT** command instead of the command line.

3.2.3. Assigning Mechanical APDL-Provided Values to Parameters

Mechanical APDL provides two powerful methods for retrieving values:

- The ***GET** command, which retrieves a value from a specified item and stores it in a specified parameter.
- The in-line get functions, which can be used in operations. Each get function returns a specific value from a specific item.

3.2.3.1. Using the *GET Command

The ***GET** command retrieves a Mechanical APDL-provided value for an item (a node, an element, an area, etc.) and stores it as a user-named parameter. Various keyword, label, and number combinations identify the retrieved item. For example, ***GET**,A,ELEM,5,CENT,X returns the centroid x-location of element 5 and stores the result as parameter A.

The format for the ***GET** command is:

```
*GET,Par,Entity,ENTNUM,Item1,IT1NUM,Item2,IT2NUM
```

where

- *Par* is the name of the parameter to store the retrieved item.
- *Entity* is a keyword for the item to be stored. Valid keywords are NODE, ELEM, KP, LINE, AREA, VOLU, etc. For a complete list of valid keywords, see the ***GET** description in the *Command Reference*.
- *ENTNUM* is the number of the entity (or zero for all entities).
- *Item1* is the name of an item for a particular entity. For example, if *Entity* is ELEM, *Item1* is either NUM (the highest or lowest element number in the selected set) or COUNT (the number of elements in the set). (For a complete list of *Item1* values for each entity type, see the ***GET** description in the *Command Reference*.)

You can think of the ***GET** command as a path down a tree structure, from general to specific information.

The following examples show the ***GET** command in use. The first command below *gets* the material attribute (the MAT reference number) of element 97 and assigns it to parameter BCD:

```
*GET,BCD,ELEM,97,ATTR,MAT     ! BCD = Material number of element 97
*GET,V37,ELEM,37,VOLU         ! V37 = volume of element 37
*GET,EL52,ELEM,52,HGEN        ! EL52 = value of heat generation in element 52
*GET,OPER,ELEM,102,HCOE,2     ! OPER = heat coefficient of element 102,face2
*GET,TMP,ELEM,16,TBULK,3      ! TMP = bulk temperature of element 16,face3
*GET,NMAX,NODE,,NUM,MAX       ! NMAX = maximum active node number
*GET,HNOD,NODE,12,HGEN        ! HNOD = value of heat generation at node 12
*GET,COORD,ACTIVE,,CSYS       ! COORD = active coordinate system number
```

3.2.3.2. Using In-line Get Functions

For some items, you can use in-line "get functions" in place of the***GET** command. A get function returns a value for an item and uses it directly in the current operation. This process allows you to bypass the dual steps of storing the value with a parameter name and then entering the parameter name in an operation. For example, suppose that you want to calculate the average x-location of two nodes. You could do the following using the ***GET** function:

1. Issue the following command to assign the x-location of Node 1 to parameter L1.

   ```
   *GET,L1,NODE,1,LOC,X
   ```

2. Issue a second ***GET** command to assign the x-location of Node 2 to parameter L2.
3. Compute the middle location from MID=(L1+L2)/2.

A shorter method is to use the node location "get function" NX(*N*), which returns the x-location of node *N*. You can use it to calculate the MID location without setting intermediate parameters L1 and L2, as is shown in the following example:

```
MID=(NX(1)+NX(2))/2
```

Get function arguments can themselves be parameters or other get functions. For instance, get function NELEM(ENUM,NPOS) returns the node number in position NPOS for element ENUM. Combining functions NX(NELEM(ENUM,NPOS)) returns the x-location of that node.

Get Function Summary (p. 97) summarizes the available get functions.

3.2.4. Listing Parameters

Once you have defined parameters, you can list them using the ***STATUS** command. If the ***STATUS** command is issued without arguments, it provides a list of all of the currently defined parameters. The following example shows the command and a typical listing.

```
*STATUS

PARAMETER STATUS-           (    5 PARAMETERS DEFINED)


NAME         VALUE         TYPE     DIMENSIONS
ABC      -24.0000000      SCALAR
HEIGHT    57.0000000      SCALAR
QR       2.070000000E+11  SCALAR
X_OR_Y   -24.0000000      SCALAR
CPARM      CASE1          CHARACTER
```

You can also access this information via **Utility Menu> List> Other> Parameters** or **Utility Menu> List> Status> Parameters> All Parameters**.

Note:

Any parameters beginning or ending in an underscore (_) are not shown by the ***STATUS** command.

You can check the status of individual parameters by providing these as arguments to the ***STATUS** command. The following example shows the status of the ABC parameter.

```
*STATUS,ABC

PARAMETER STATUS- abc       (    5 PARAMETERS DEFINED)


NAME         VALUE         TYPE     DIMENSIONS
ABC      -24.0000000      SCALAR
```

You can also check the status of specific parameters via **Utility Menu> List> Other> Named Parameter** or **Utility Menu> List> Status> Parameters> Named Parameters**.

3.3. Deleting Parameters

You can delete specific parameters in two ways:

- Issue the "=" command, leaving the right-hand side of the command blank. For example, to delete the QR parameter issue this command:

  ```
  QR=
  ```

- Issue the ***SET** command, but do not specify a value for the parameter. For example, to delete the QR parameter via the ***SET** command, issue the command as follows:

  ```
  *SET,QR,
  ```

Setting a numeric parameter equal to zero does not delete it. Similarly, setting a character parameter equal to empty single quotes (` `) or placing blanks within single quotes does not delete the parameter.

3.4. Using Character Parameters

Typically, character parameters are used to provide file names and extensions. The desired file name can be assigned to a character parameter, and that parameter can be used anywhere a file name is required. Similarly, a file extension can be assigned to a character parameter and used where appropriate (typically the *Ext* command argument). In batch mode, this allows you to easily change file names for multiple runs by simply changing the initial alphanumeric "value" of the character parameter in your input file.

Note:

Remember that character parameters are limited to a total of eight characters.

The following is a list of general uses for character parameters.

- As arguments to any applicable command field (that is, where alphanumeric input is expected).
- As macro name arguments for the ***USE** command.

```
NAME='MACRO'    ! MACRO is the name of a macro file
*USE,NAME       ! Calls MACRO
```

- As arguments to macro calls for ***USE** and for the "unknown command" macro. Any of the following macro calls are allowed:

```
ABC='SX'
*USE,NAME,ABC
```

or

```
*USE,NAME,'SX'

DEF='SY'
NEWMACRO,DEF     ! Calls existing macro file NEWMACRO.MAC
```

or

```
NEWMACRO,'SY'
```

3.5. Substitution of Numeric Parametric Values

Whenever you use a parameter name in a numeric command field, its value is automatically substituted. If no value has been assigned to the parameter (that is, if the parameter has not been defined), a near-zero value (2^{-100}) is substituted, usually without warning.

Note:

Defining the parameter after it is used in a command does not "update" the command in *most* cases. (Exceptions are the commands **/TITLE**, **/STITLE**, ***ABBR**, and **/TLABEL**. See Forced Substitution (p. 14) for more information.) For example:

```
Y=0
X=2.7
N,1,X,Y      ! Node 1 at (2.7,0)
Y=3.5        ! Redefining parameter Y now does not update node 1
```

3.5.1. Preventing Substitution

You can prevent parameter substitution by enclosing the parameter name with single quotes ('), for example, 'XYZ'. The literal string is then used; therefore, this feature is valid only in *non-numerical* fields.

Conversely, you can force parameter substitution in titles, subtitles, and filenames by enclosing the parameter name with percent signs (%). For example,

```
/TITLE, TEMPERATURE CONTOURS AT TIME=%TM%
```

specifies a title in which the numerical value of parameter TM is substituted. Note that the parameter is substituted at the time the title is used.

3.5.2. Substitution of Character Parametric Values

Use of a character parameter in an alphanumeric command field generally results in automatic substitution of its value. Forced substitution and character parameter restrictions are explained below.

3.5.2.1. Forced Substitution

As with numerical parameters, you can force the substitution of a character parameter value in certain cases where substitution would not occur otherwise. This is done by enclosing the character parameter name with percent signs (%). Forced substitution of character parameters is valid for the following commands:

- **/TITLE** command (*`Title`* field). Specifies titles for various printed output.
- **/STITLE** command (*`Title`* field). Specifies subtitles, similar to**/TITLE**. (You cannot access the **/STITLE** command directly in the GUI.)
- **/TLABEL** command (*`Text`* field). Specifies text string for annotation.
- ***ABBR** command (*`Abbr`* field). Defines an abbreviation.

Forced substitution is also valid in the following types of fields:

- Any filename or extension command argument. These arguments apply to commands such as **/FILNAME**, **RESUME**, **/INPUT**, **/OUTPUT**, and **FILE**. (Direct parameter substitution is also valid in these fields.)
- Any 32 character field: A typical example is the name of macros. (Direct substitution is not valid for these fields.)
- As a command name in any command name field. Also as an "unknown command" macro name in field 1. For example:

```
R='RESUME'
%R%,MODEL,DB
```

The following example of the command input method shows forced substitution for a subtitle definition and for a directory name.

```
A='TEST'
B='.RST'
C='/Mechanical APDL'
D='/MODELS/'
/STITLE,,RESULTS FROM FILE %C%%D%%A%%B%


 SUBTITLE  1 =
 RESULTS FROM FILE /Mechanical APDL/MODELS/TEST.RST


/POST1
FILE,%C%%D%%A%,RST     ! Read results from /Mechanical APDL/MODELS/TEST.RST
```

3.5.2.2. Other Places Where Character Parameters Are Valid

In addition to the more general applications already discussed, there are some specific instances where character parameters are allowed for added convenience. The commands which are affected and details of usage are outlined below.

***ASK**

This command may prompt you for an alphanumeric string (up to eight characters enclosed in single quotes) which is assigned to a character scalar parameter. (You cannot access the ***ASK** command directly in the GUI.)

***CFWRITE**

This command writes Mechanical APDL commands to the file opened by ***CFOPEN**. It can be used to write a character parameter assignment to that file. For example, ***CFWRITE**,B = 'FILE' is valid. (You cannot access the ***CFWRITE** and ***CFOPEN** commands directly in the GUI.)

***IF and *ELSEIF**

Character parameters may be used for the *VAL1* and *VAL2* arguments of these commands. For the *Oper* argument, only labels EQ (equal) and NE (not equal) are valid when using character parameters. (You cannot access the ***IF** and ***ELSEIF** commands directly in the GUI.) Example:

```
CPARM='NO'
*IF,CPARM,NE,'YES',THEN
```

***MSG**

Character parameters are allowed as input for the *VAL1* through *VAL8* arguments. The data descriptor %C is used to indicate alphanumeric character data on the format line (which must follow the ***MSG** command). The %C corresponds to the FORTRAN descriptor A8. (You cannot access the ***MSG** command directly in the GUI.)

PARSAV and PARRES

These commands save character parameters to a file (**PARSAV**) and resume character parameters from a file (**PARRES**).

***VREAD**

This command reads alphanumeric character data from a file and generates a character-array parameter. The FORTRAN character descriptor (A) can be used in the format line which must follow the ***VREAD** command.

***VWRITE**

This command writes character parameter data to a file in a formatted sequence. The FORTRAN character descriptor (A) can be used in the format line which must follow the ***VWRITE** command.

3.5.2.3. Character Parameter Restrictions

Although character parameters have much of the same functionality as numerical parameters, there are several instances where character parameters are not valid.

- Character parameter substitution is not allowed for the *Par* argument of the ***SET**, ***GET**, ***DIM**, and ***STATUS** commands.
- Interactive editing of array parameters (***VEDIT** command) is not available for character array parameters.
- Vector operation commands, such as ***VOPER**, ***VSCFUN**, ***VFUN**, ***VFILL**, ***VGET**, and ***VITRP**, do not work with character array parameters.
- When operating on character parameters, the specification commands ***VMASK** and ***VLEN** are applicable only to the ***VWRITE** and ***VREAD** commands.
- Character parameters are not valid in parametric expressions which use addition, subtraction, multiplication, etc.

3.6. Dynamic Substitution of Numeric or Character Parameters

Dynamic substitution of parameters occurs for the following commands: **/TITLE**, **/STITLE**, ***ABBR**, **/AN3D**, and **/TLABEL**. Dynamic substitution allows the revised value of a parameter to be used, even if the command which uses the parameter value has not been reissued.

Example:

```
XYZ='CASE 1'
/TITLE,This is %XYZ%
APLOT
```

The title "This is CASE 1" appears on the area plot.

You can then change the value of XYZ and the new title appears on subsequent plots, even though you did not reissue **/TITLE**.

```
XYZ='CASE 2'
```

The title "This is CASE 2" appears on subsequent plots.

3.7. Parametric Expressions

Parametric expressions involve operations among parameters and numbers such as addition, subtraction, multiplication, and division. For example:

```
X=A+B
P=(R2+R1)/2
D=-B+(E**2)-(4*A*C)      ! Evaluates to D = -B + E2 - 4AC
XYZ=(A<B)+Y**2           ! Evaluates to XYZ = A + Y2 if A is less than B;
                         ! otherwise to XYZ = B + Y2
INC=A1+(31.4/9)
M=((X2-X1)**2-(Y2-Y1)**2)/2
```

The following is a complete list of APDL operators:

Operator	Operation
+	Addition
_	Subtraction
*	Multiplication
/	Division
**	Exponentiation
<	Less-Than Comparison
>	Greater-Than Comparison

You can also use parentheses for clarity and for "nesting" of operations, as shown above. The order in which the Mechanical APDL program evaluates an expression is as follows:

1. Operations in parentheses (innermost first)
2. Exponentiation (in order, from right to left)
3. Multiplication and division (in order, from left to right)
4. Unary association (such as +A or -A)
5. Addition and subtraction (in order, from left to right)
6. Logical evaluation (in order, from left to right)

Thus an expression such as Y2=A+B**C/D*E is evaluated in this order: B**C first, /D second, *E third, and +A last. For clarity, you should use parentheses in expressions such as these. Parentheses can be nested up to four levels deep, and up to nine operations can be performed within each set of parentheses. As a general rule, avoid using blank spaces between operators in expressions. In particular, never include a blank space before the * character because the rest of the input line (beginning with the *) is interpreted as a comment and is therefore ignored. (Do not use this convention as a comment; use an exclamation point (!) for this purpose.)

3.8. Parametric Functions

A parametric function is a programmed sequence of mathematical operations which returns a single value, such as SIN(X), SQRT(B), and LOG(13.2). The following table provides a complete list of functions currently available in Mechanical APDL.

ABS(x)	Absolute value of x.
SIGN(x,y)	Absolute value of x with sign of y. y=0 results in positive sign.
CXABS(x,y)	Absolute value of the complex number x + y*i* ($\sqrt{x^2+y^2}$)
EXP(x)	Exponential of x (e^x).
LOG(x)	Natural log of x (ln (x)).
LOG10(x)	Common log of x ($\log_{10}$(x)).
SQRT(x)	Square root of x.
NINT(x)	Nearest integer to x.
MOD(x,y)	Remainder of x/y, computed as x - (INT(x/y) * y). y=0 returns zero (0).
RAND(x,y)	Random number (uniform distribution) in the range x to y (x = lower bound, y = upper bound).
GDIS(x,y)	Random sample of a Gaussian (normal) distribution with mean x and standard deviation y.
SIN(x), COS(x), TAN(x)	Sine, Cosine, and Tangent of x. x is in radians by default, but can be changed to degrees with ***AFUN**.
SINH(x), COSH(x), TANH(x)	Hyperbolic sine, Hyperbolic cosine, and Hyperbolic tangent of x.
ASIN(x), ACOS(x), ATAN(x)	Arcsine, Arccosine, and Arctangent of x. x must be between -1.0 and +1.0 for ASIN and ACOS. Output is in radians by default, but can be changed to degrees with ***AFUN**. Range of output is -pi/2 to +pi/2 for ASIN and ATAN, and 0 to pi for ACOS.
ATAN2(y,x)	Arctangent of y/x with the sign of each component considered. Output is in radians by default, but can be changed to degrees with ***AFUN**. Range of output is -pi to +pi.
VALCHR(*CPARM*)	Numerical value of *CPARM* (if *CPARM* is non-numeric, returns 0.0).
CHRVAL(*PARM*)	Character value of numerical parameter *PARM*. Number of decimal places depends on magnitude.
UPCASE(*CPARM*)	Upper case equivalent of *CPARM*.
LWCASE(*CPARM*)	Lower case equivalent of *CPARM*.
LARGEINT(x,y)	Forms a 64-bit pointer from low (x) and high (y) 32-bit integers.

The following are examples of parametric functions:

```
PI=ACOS(-1)              ! PI = arc cosine of -1, PI calculated to machine accuracy
Z3=COS(2*THETA)-Z1**2
R2=SQRT(ABS(R1-3))
X=RAND(-24,R2)           ! X = random number between -24 and R2


*AFUN,DEG                ! Units for angular functions are degrees
THETA=ATAN(SQRT(3))      ! THETA evaluates to 60 degrees
PHI=ATAN2(-SQRT(3),-1)   ! PHI evaluates to -120 degrees
*AFUN,RAD                ! Units for angular functions reset to radians


X249=NX(249)             ! X-coordinate of node 249
SLOPE=(KY(2)-KY(1))/(KX(2)-KX(1))
                         ! Slope of line joining keypoints 1 and 2


CHNUM=CHRVAL(X)          ! CHNUM = character value of X
UPPER=UPCASE(LABEL)      ! UPPER = uppercase character value of parameter LABEL
```

3.9. Saving, Resuming, and Writing Parameters

If you must use currently defined parameters in another Mechanical APDL session, you can write them to a file and then read (resume) that file. When you read the file, you can either completely replace currently defined parameters or add to them (replacing those that already exist).

To write parameters to a file, use the **PARSAV** command.

The parameters file is an ASCII file consisting largely of APDL ***SET** commands used to define the various parameters. The following example shows the format of this file.

```
/NOPR
*SET,A       ,  10.00000000000
*SET,B       ,  254.3948750000
*SET,C       ,'string  '
*SET,_RETURN , 0.0000000000000E+00
*SET,_STATUS ,  1.000000000000
*SET,_ZX     ,'        '
/GO
```

To read parameters from a file use the **PARRES** command.

If you wish, you can write up to ten parameters or array parameters using FORTRAN real formats to a file. You can use this feature to write your own output file for use in other programs, reports, etc. To do this, use the ***VWRITE** command. The ***VWRITE** command is discussed in Operations Among Array Parameters (p. 37).

3.10. Array Parameters

In addition to scalar (single valued) parameters, you can define array (multiple valued) parameters. Mechanical APDL arrays can be:

- 1-D (a single column)
- 2-D (rows and columns)
- 3-D (rows, columns, and planes)

- 4-D (rows, columns, planes, and books)
- 5-D (rows, columns, planes, books, and shelves)

Mechanical APDL provides three types of arrays:

ARRAY

This type is similar to FORTRAN arrays and is the default array type when dimensioning arrays. As with FORTRAN arrays, the indices for rows, columns, and planes are sequential integer numbers beginning with one. Array elements can be either integers or real numbers.

CHAR

This is a character array, with each element consisting of an alphanumeric value not exceeding eight characters. The indices for rows, columns, and planes are sequential integer numbers beginning with one.

TABLE

This is a special type of numeric array which allows Mechanical APDL to calculate (through linear interpolation) values between these array elements explicitly defined in the array. Moreover, you can define the array indices for each row, column, and plane and these indices are real (not integer) numbers. Array elements can be either integers or real numbers. As we'll see in the later discussion on TABLE arrays, this capability provides a powerful method for describing mathematical functions.

STRING

You can use the ***DIM**, STRING capability to enter character strings into your arrays. Index numbers for columns and planes are sequential values beginning with 1. Row indices are determined by the character position in the string. See the ***DIM** command for more information.

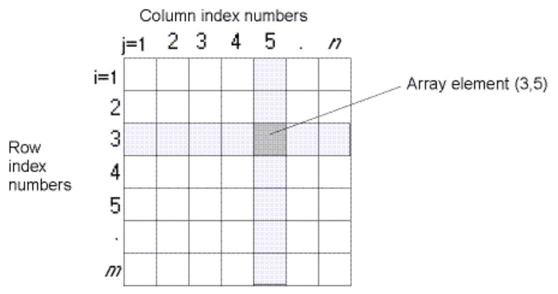
The following array parameter topics are available:

3.10.1. Array Parameter Basics

Consider a 2-D array (either ARRAY or CHAR) as shown below. It is *m* rows long and *n* columns wide; that is, its dimensions are *m* times *n*. Each row is identified by a row index number *i*, which varies from 1 to *m*, and each column is identified by a column index number *j*, which varies from 1 to *n*.

The quantities that make up the array are array elements. Each array element is identified as `(i,j)`, where `i` is its row index number and `j` is its column index number.

Figure 3.1: A Graphical Representation of a 2-D Array



We can extend these definitions to a 3-D array parameter, which may be *m* rows long, *n* columns wide, and *p* planes deep. The plane index number is *k*, which varies from 1 to *p*. Each array element is identified as (`i,j,k`,). The following figure shows a 3-D array.

Figure 3.2: A Graphical Representation of a 3-D Array

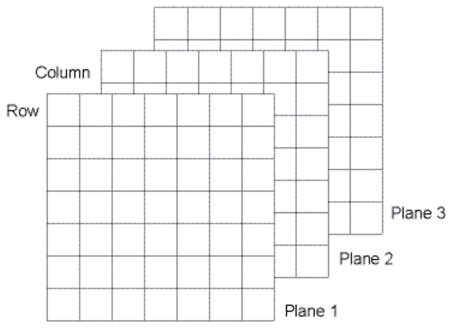
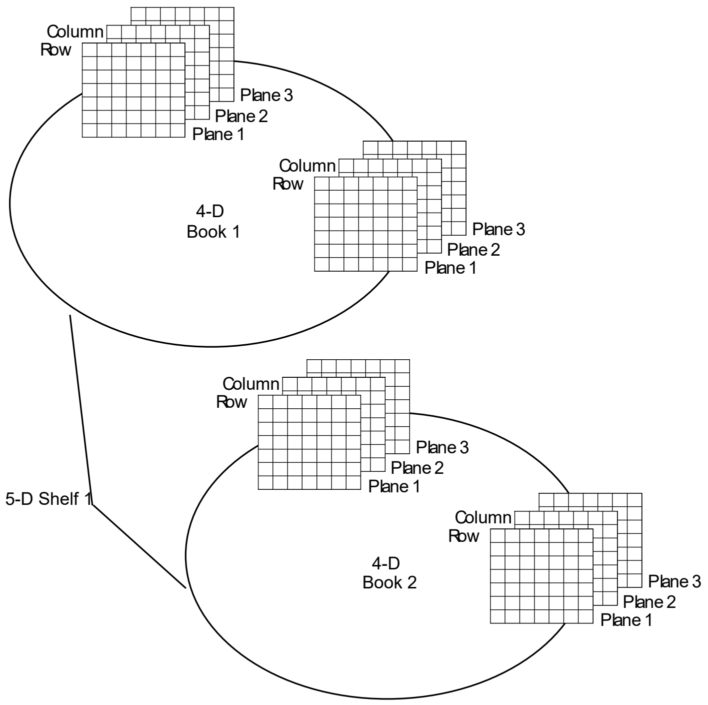


Figure 3.3: A Graphical Representation of a 5-D Array



3.10.2. Array Parameter Examples

Type ARRAY parameters consist of discrete numbers that are simply arranged in a tabular fashion for convenience. Consider the following examples.

$$\text{NTEMP} = \begin{bmatrix} -47.6 \\ -5.2 \\ 25.0 \\ 86.5 \\ 107.9 \\ 168.7 \\ 225.0 \end{bmatrix} \qquad \text{EVOLUM} = \begin{bmatrix} 0.025 \\ 0.01 \\ 0.265 \\ 1.00 \\ 0.832 \\ 0.52 \\ 1.032 \\ 0.002 \\ 0.697 \\ 0.01 \end{bmatrix}$$

$$\text{COMPSTRS} = \begin{bmatrix} 12152 & 814 & -386 & 202 & -82 & -1108 \\ 14848 & 1057 & -704 & 117 & -101 & -555 \\ 15490 & 1033 & -713 & 15 & -76 & 235 \\ 13899 & 786 & -348 & -103 & -45 & 848 \\ 10813 & 420 & -66 & -211 & -17 & 1065 \\ 7151 & 109 & 111 & -272 & 11 & 1052 \end{bmatrix}$$

The parameter NTEMP could be an array of temperatures at selected nodes; NTEMP(1) = -47.6 could be the temperature at node 27, NTEMP(2) = -5.2 could be the temperature at node 43, and so on. Similarly, EVOLUM could be an array of element volumes, and COMPSTRS could be an array of nodal component stresses, with each column representing a particular direction (X, Y, Z, XY, YZ, XZ, for example).

A type CHAR array parameter is structured similarly to an ARRAY parameter, with the tabular values being alphanumeric character strings (up to eight characters). Two examples of character array parameters are:

$$\text{FILNAM} = \begin{bmatrix} \text{JOB1} \\ \text{JOB2} \\ \text{JOB3} \\ \text{JOB4} \\ \text{JOB5} \end{bmatrix} \qquad \text{EXTENS} = \begin{bmatrix} \text{LOG} \\ \text{ERR} \\ \text{DB} \\ \text{LIB} \\ \text{MAC} \end{bmatrix}$$

3.10.3. Tabular Input via Table Array Parameters

A type TABLE array parameter consists of numbers arranged in a tabular fashion, much like the ARRAY type but with three important differences:

- Mechanical APDL can calculate (via linear interpolation) any values that fall between the explicitly declared array element values.
- A table array contains a 0 row and 0 column used for data-access index values and, unlike standard arrays, these index values can be real numbers.

  The only restriction is that the index values must consist of numerically *increasing* (never decreasing) numbers. You must explicitly declare a data access index value for each row and column; otherwise the default value assigned is the *tiny number* (7.888609052E-31). You can more conveniently define the index starting point and index values (***TAXIS**).
- A plane index value resides in the 0,0 location for each plane.

The following figure shows a table array with data-access index values. The indices are specified as the "0" row and column values.

Figure 3.4: A Graphical Representation of a Table Array

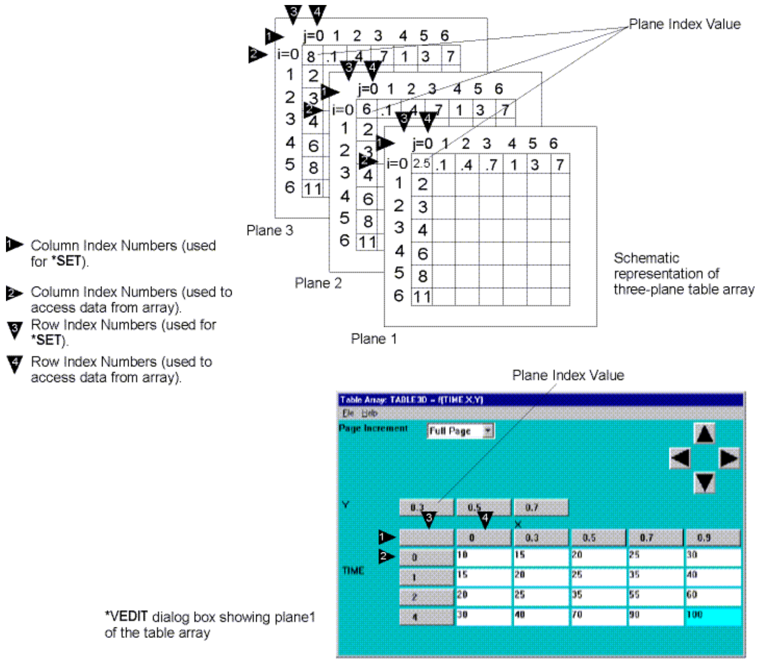

Plane Index Value
j=0 1 2 3 4 5 6
i=0 8 .1 .4 .7 1 3 7
Plane 3
Plane 2
i=0 6
Plane 1
i=0 2.5
Schematic representation of three-plane table array
Column Index Numbers (used for *SET).
Column Index Numbers (used to access data from array).
Row Index Numbers (used for *SET).
Row Index Numbers (used to access data from array).
Plane Index Value
*VEDIT dialog box showing plane1 of the table array
File Help
Page Increment
Full Page
Y
X
TIME

Table Array: TABLE3D = f(TIME,X,Y)

Y	0.3	0.5	0.7		
	0	0.3	0.5	0.7	0.9
0	10	15	20	25	30
1	15	20	25	35	40
2	20	25	35	55	60
4	30	40	70	90	100

As shown in the above example, when configuring a table array you must set

- The plane index value as the 0,0 element value for *each* plane.
- The data-access column index values in the elements in the *0 row in plane 1*. Only the column index values from plane 1 are used when accessing data from the array for all planes. When setting the array element values, you use the traditional row and column index numbers.
- The data-access row index values in the elements in the *0 column in plane 1*. Only the row index values from plane 1 are used when accessing data from the array for all planes. When setting the array element values, you use the traditional row and column index numbers.

For more information, see Defining Linear Material Properties Using Tabular Input in the *Material Reference* and Applying Loads Using Tabular Input in the *Basic Analysis Guide*.

3.10.4. Defining and Listing Array Parameters

To define an array parameter, first declare its type and dimensions (***DIM**).

This following examples illustrate the ***DIM** command used to dimension various types of arrays:

```
*DIM,AA,,4  ! Type ARRAY is default, dimension 4[x1x1]
*DIM,XYZ,ARRAY,12       ! Type ARRAY array, dimension 12[x1x1]
*DIM,FORCE,TABLE,5      ! Type TABLE array, dimension 5[x1x1]
*DIM,T2,,4,3            ! Dimensions are 4x3[x1]
*DIM,CPARR1,CHAR,5      ! Type CHAR array, dimension 5[x1x1]
```

Note:

Array elements for ARRAY and TABLE are initialized to 0 (except for the 0 row and column for TABLE, which is initialized to the tiny value). Array elements for CHAR are initialized to a blank value.

For array parameter operations (for example ***VOPER**, ***VFUN**), the resulting array parameter (ParR) need not be dimensioned beforehand. Also, array parameters defined completely with the implied (colon) loops convention (for example a(1:5)=10,20,30,40,50) need not be dimensioned beforehand.

The next example shows how to fill a 5-D array with data. Use 1-D tables to load a 5-D table. Use the ***TAXIS** to define the table index values. See the full example at Example Analysis Using 5-D Table Array.

```
*dim,xval,array,X1
*dim,yval,array,Y1
yval(1)=0,20
*dim,zval,array,10
zval(1)=10,20,30,40,50,60,70,80,90,100
*dim,tval,array,5
tval(1)=1,.90,.80,.70,.60
*dim,tevl,array,5
tevl(1)=1,1.20,1.30,1.60,1.80

*dim,ccc,tab5,X1,Y1,Z1,D4,D5,X,Y,Z,TIME,TEMP
*taxis,ccc(1,1,1,1,1),1,0,wid                   !!! X-Dim
*taxis,ccc(1,1,1,1,1),2,0,hth                   !!! Y-Dim
*taxis,ccc(1,1,1,1,1),3,1,2,3,4,5,6,7,8,9,10    !!! Z-Dim
*taxis,ccc(1,1,1,1,1),4,0,10,20,30,40            !!! Time
*taxis,ccc(1,1,1,1,1),5,0,50,100,150,200        !!! Temp
*do,ii,1,2
```

```
   *do,jj,1,2
      *do,kk,1,10
         *do,ll,1,5
            *do,mm,1,5
               ccc(ii,jj,kk,ll,mm)=(xval(ii)+yval(jj)+zval(kk))*tval(ll)*tevl(mm)
            *enddo
         *enddo
      *enddo
   *enddo
*enddo
```

3.10.5. Specifying Array Element Values

You can specify array element values by

- Setting individual array element values through the ***SET** command or "=" shortcut.
- Filling individual vectors (columns) in the array with either specified or calculated values (the ***VFILL** command, for example).
- Interactively specifying values for the elements through the ***VEDIT** dialog.
- Reading the values from an ASCII file (***VREAD** or ***TREAD** commands).

Note:

You cannot create or edit 4- or 5-D arrays interactively. ***VEDIT**, ***VREAD**, and ***TREAD** are not applicable to 4- or 5-D arrays.

3.10.5.1. Specifying Individual Array Values

You can use either the ***SET** command or the "=" shortcut. Usage is the same as for scalar parameters, except that you now define a column of data (up to ten array element values per "=" command). For example, to define the parameter XYZ dimensioned above as a 12x1 array you need two "=" commands. In the following example the first command defines the first eight array elements and the second command defines the next four array elements:

```
XYZ(1)=59.5,42.494,-9.01,-8.98,-8.98,9.01,-30.6,51
XYZ(9)=-51.9,14.88,10.8,-10.8
```

$$XYZ = \begin{bmatrix} 59.5 \\ 42.494 \\ -9.01 \\ -8.98 \\ -8.98 \\ 9.01 \\ -30.6 \\ 51 \\ -51.9 \\ 14.88 \\ 10.8 \\ -10.8 \end{bmatrix}$$

Notice that the starting location of the array element is indicated by the row index number of the parameter (1 in the first command, 9 in the second command).

The following example shows how to define the element values for the 4x3 array parameter T2, dimensioned earlier in the ***DIM** examples:

```
T2(1,1)=.6,2,-1.8,4           ! defines (1,1),(2,1),(3,1),(4,1)
T2(1,2)=7,5,9.1,62.5          ! defines (1,2),(2,2),(3,2),(4,2)
T2(1,3)=2E-4,-3.5,22,.01      ! defines (1,3),(2,3),(3,3),(4,3)
```

$$T2 = \begin{bmatrix} 0.6 & 7.0 & 0.0002 \\ 2.0 & 5.0 & -3.5 \\ -1.8 & 9.1 & 22.0 \\ 4.0 & 62.5 & 0.01 \end{bmatrix}$$

The following example defines element values for the TABLE array parameter FORCE discussed earlier.

```
FORCE(1)=0,560,560,238.5,0
FORCE(1,0)=1E-6,.8,7.2,8.5,9.3
```

The first "=" command defines the five array elements of the TABLE array FORCE. The second and third "=" commands redefine the index numbers in the j=0 and i=0 row.

$$FORCE = \begin{array}{c|c} & 0 \\ 1E-6 & 0.0 \\ 0.8 & 560.0 \\ 7.2 & 560.0 \\ 8.5 & 238.5 \\ 9.3 & 0.0 \end{array}$$

Character array parameters can also be defined using the "=" command. Assigned values can be up to eight characters each and must be enclosed in single quotes. For example:

```
*DIM,RESULT,CHAR,3       !Character array parameter with dimensions (3,1,1)
RESULT(1)='SX','SY','SZ'  !Assigns values to parameter RESULT
```

Notice that, as when defining a numerical array parameter, the starting location of the array element must be specified (in this case, the row index number 1 is indicated).

Note:

CHAR cannot be used as a character parameter name because it creates a conflict with the CHAR label on the ***DIM** command. Mechanical APDL substitutes the character string value assigned to parameter CHAR when CHAR is input on the third field of the ***DIM** command (Type field).

3.10.5.2. Filling Array Vectors

You can use the ***VFILL** command to "fill" an ARRAY or TABLE vector (column).

See the ***VFILL** command reference information in the *Command Reference* for more detail about the command syntax. The following example illustrates the capabilities of the ***VFILL** command.

```
*DIM,DTAB,ARRAY,4,3                  ! dimension 4 x 3 numeric array
*VFILL,DTAB(1,1),DATA,-3,8,-12,57    ! four data values loaded into vector 1
*VFILL,DTAB(1,2),RAMP,2.54,2.54      ! fill vector 2 with values starting at
                                     ! 2.54 and incrementing by 2.54
*VFILL,DTAB(1,3),RAND,1.5,10         ! fill vector 3 with random numbers between
```

```
                                            ! 1.5 and 10.  Results vary due to
                                            ! random number generation.
```

$$DTAB = \begin{bmatrix} -3 & 2.54 & 2.799901284 \\ 8 & 5.08 & 6.11292418 \\ -12 & 7.62 & 6.70205516 \\ 57 & 10.16 & 4.11487684 \end{bmatrix}$$

3.10.5.3. Interactively Editing Arrays

The ***VEDIT** command, available only in interactive mode, launches a data entry dialog enabling you to edit an ARRAY or TABLE (not CHAR) array. The dialog provides a number of convenient features:

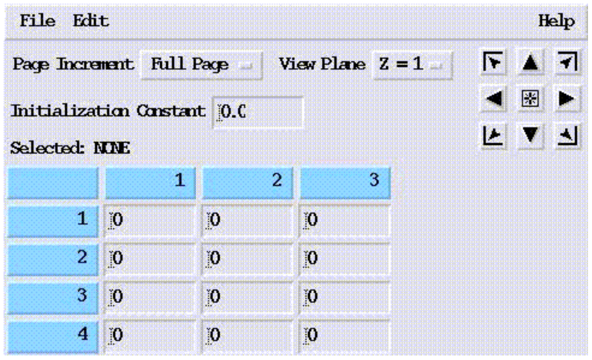
- A spreadsheet-style editor for array element values.
- Navigational controls for scrolling through large arrays.
- An initialize function to set any row or column to a specified value (ARRAY type only).
- Delete, copy, and insert functions for moving rows or columns of data (ARRAY type only).

Complete instructions for using the dialog are available from the box's Help button.

Note:

You cannot edit a 4- or 5-D ARRAY or TABLE interactively.

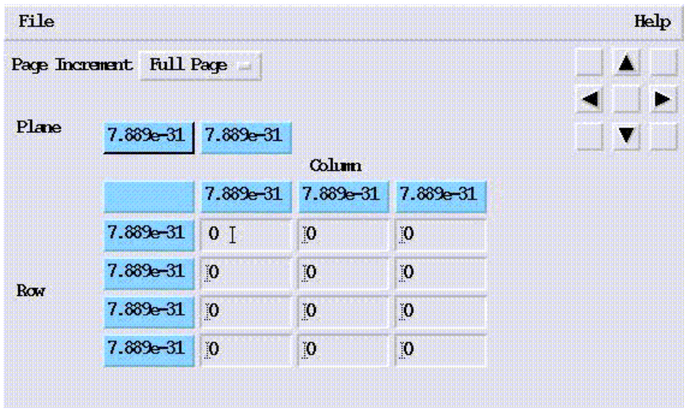
Figure 3.5: An Example *VEDIT Dialog Box for an ARRAY



Page Increment: Full Page
View Plane: Z = 1
Initialization Constant: 0.0
Selected: NONE

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

Figure 3.6: An Example *VEDIT Dialog Box for a TABLE



3.10.5.4. Filling an Array From a Data File Using *VREAD

You can fill an array from a data file via the ***VREAD** command. The command reads information from an ASCII data file and begins writing it into the array, starting with the index location that you specify. You can control the format of the information read from the file through data descriptors. The data descriptors must be enclosed in parenthesis and placed on the line following the ***VREAD** command. See Vector Operations (p. 37) for more information about data descriptors. The data descriptors control the number of fields to be read from each record, the width of the data fields, and the position of the decimal point in the field.

For example, given the following data file named `dataval`:

```
1.5     7.8  12.3
15.6  -45.6  42.5
```

and an array called EXAMPLE that has been dimensioned as 2 x 3, the following commands (provided as either a part or a macro or input listing)

```
*DIM,EXAMPLE,,2,3
*VREAD,EXAMPLE(1,1),dataval,,,JIK,3,2
(3F6.1)
```

result in

EXAMPLE =

```
1.5     7.8  12.3
15.6  -45.6  42.5
```

The ***VREAD** command cannot be issued directly from the command input window. However, the **Utility Menu> Parameters> Array Parameters> Read from File** dialog offers a way to specify the data descriptors and issue the command in interactive mode.

Note:

You cannot fill a 4- or 5-D array via ***VREAD**.

3.10.5.5. Filling a TABLE Array From a Data File Using *TREAD

Once configured, you have two options for specifying values for the TABLE array elements: you can add values as you would for any other type of array, or you can read in a table of data from an external file.

To read in a table of data from an external file, you still define the TABLE array first, specifying the number of rows, columns, and planes, and the labels for each. You can then read an ASCII file containing the table of data via the ***TREAD** command. At this time, you also specify the number of lines to skip (NSKIP) between the top of the file and the first line of the table.

When reading data from an external file, remember:

- The file containing the table of data can be created in a text editor or an external application (such as Microsoft Excel), but it must be in ASCII form, tab-delimited, to be read into Mechanical APDL.
- You must first define the array in Mechanical APDL, remembering to allow for the index values (0,0).
- The values are read straight across the rows until all columns on each row of the array are filled; Mechanical APDL then wraps from one row to the next and begins to fill those columns, and so on. Be sure that the dimensions of the array you defined are correct. If you mistakenly define fewer columns in the Mechanical APDL array than required, Mechanical APDL begins filling in the next row of the array using the values remaining in the first row of the data table being read. Similarly, if you define more columns in the Mechanical APDL array than required, Mechanical APDL fills all columns of the array using values from the next row of the data table being read, and only then wrap and begin filling the next row.

You can create 1-D, 2-D, and 3-D tables by reading data from an external file. Examples of how you create each of these follows.

Note:

You cannot fill a 4- or 5-D TABLE using ***TREAD**.

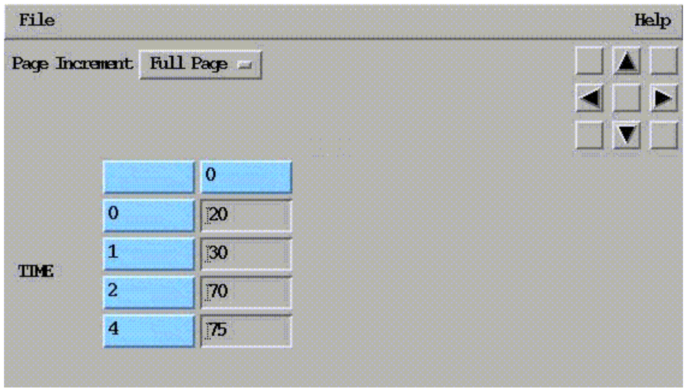
Example 1: 1-D Table

First, create the 1-D table using the application of your choice (such as a spreadsheet application, a text editor, etc.) and then save the file as a text file in tab-delimited format. In this example, the table is named "Tdata" and contains data for a time vs. temperature table . In its ASCII form, the table would look like this:

Time Temperature Table	
Time	Temp
0	20
1	30
2	70
4	75

Define a TABLE parameter "Tt" via the ***DIM** command. Specify four rows and one column, row label of Time, and column label of Temp. The data table you created has four rows and one column of data, plus the row and column index values (where the first column, TIME, is the row index values). Then, read in the file, specifying two skipped lines. The TABLE array in Mechanical APDL looks like this:

Figure 3.7: Example 1-D TABLE Array Dialog



		0
TIME	0	20
	1	30
	2	70
	4	75

The same example using command input looks like this:

```
*DIM,Tt,table,4,1,1,TIME
*TREAD,Tt,tdata,txt,,2
```

Example 2: 2-D Table

For this example, create (in a spreadsheet application, a text editor, etc.) a 2-D table named "T2data" containing temperature data as a function of time and x-coordinate and read it into a TABLE array parameter called "Ttx." The table, in its ASCII form, would look like this:

Temp (time-X-coord) Table					
Time	X-Coordinate				
0	0	.3	.5	.7	.9
0	10	15	20	25	30
1	15	20	25	35	40
2	20	25	35	55	60
4	30	40	70	90	100

In Mechanical APDL, you define a TABLE parameter "Ttx" using the ***DIM** command. Specify four rows, five columns, one plane, row label of TIME, and column label of X. The data table you created has four rows and five columns of data, plus the row and column index values. Then, read in the file, specifying two skipped lines. The TABLE array in Mechanical APDL looks like this:

Figure 3.8: Example 2-D TABLE Array Dialog

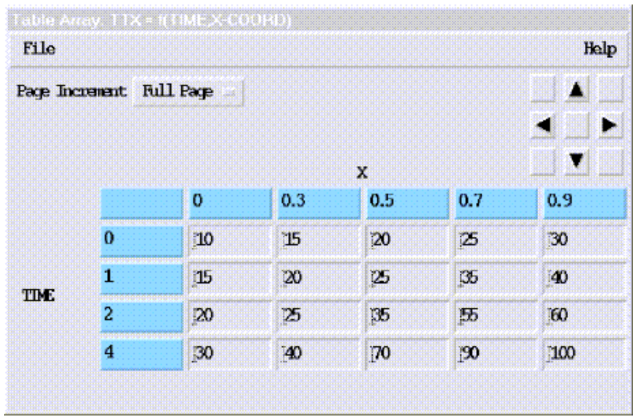


Table Array: TTX = f(TIME,X-COORD)

File | Help

Page Increment: Full Page

X

		0	0.3	0.5	0.7	0.9
	0	10	15	20	25	30
TIME	1	15	20	25	35	40
	2	20	25	35	55	60
	4	30	40	70	90	100

The same example using command input looks like this:

```
*DIM,Ttx,table,4,5,,time,X
*TREAD,Ttx,t2data,txt,,2
```

Example 3: 3-D Table

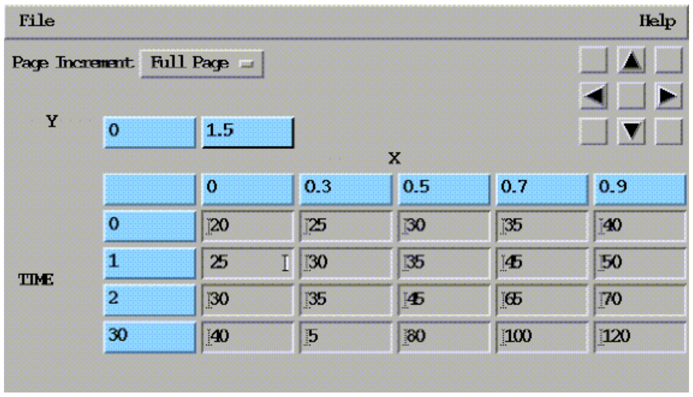
For this example, create a 3-D table named "T3data" containing temperature data as a function of time, x-coordinate, and y-coordinate and read it into a TABLE array parameter called "Ttxy." The table, in its ASCII form, would look like this:

Temp (time-X-coord) Table					
Time	X-Coordinate				
0	0	.3	.5	.7	.9
0	10	15	20	25	30
1	15	20	25	35	40
2	20	25	35	55	60
4	30	40	70	90	100
1.5	0	.3	.5	.7	.9
0	20	25	30	35	40
1	25	30	35	45	50
2	30	35	45	65	70
4	40	50	80	100	120

In the example above, the bold values (in the (0,0,Z) positions) indicate the separate planes. Each plane of data, along with the row and column index values, is repeated for the separate planes. Only the plane index value and the actual data values are different. The shaded area above shows the values that change from plane to plane.

In Mechanical APDL, define a TABLE parameter "Ttxy" via the ***DIM** command. In the case of a 3-D table, the table is dimensioned according to the number of rows, columns, and planes of data. The first column (TIME) is the row index values and the first row is the column index values. Specify four rows, five columns, two planes, row label of TIME, column label of X, and plane label of Y. The data table you created has four rows and five columns of data in two planes, plus the row and column index values. Then, read in the file, specifying two skipped lines. The TABLE array in Mechanical APDL looks like this for the second plane of data (Y=1.5):

Figure 3.9: Example 3-D TABLE Array Dialog



Y: 0, 1.5

X

TIME	0	0.3	0.5	0.7	0.9
0	20	25	30	35	40
1	25	30	35	45	50
2	30	35	45	65	70
30	40	5	80	100	120

The same example using command input looks like this:

```
*DIM,Ttxy,table,4,5,2,TIME,X,Y
*TREAD,Ttxy,t3data,txt,,2
```

3.10.5.6. Interpolating Values

When accessing information from the array, Mechanical APDL interpolates values between those explicitly set.

As examples of how Mechanical APDL interpolates values in TABLE arrays, consider the following:

$$
A = \begin{array}{c|c|} & 1.0 \\ 1.0 & 12.0 \\ 2.0 & 28.0 \\ 3.0 & 146.4 \end{array}
\qquad
PQ = \begin{array}{c|cc|} & 1.0 & 2.0 \\ 1.0 & 2.8 & 4.2 \\ 2.0 & -9.6 & -12.3 \\ 3.0 & 42.0 & 9.7 \\ 4.0 & -4.5 & 2.0 \end{array}
$$

Given that A is a TABLE array parameter, the Mechanical APDL program can calculate any value between A(1) and A(2), for example

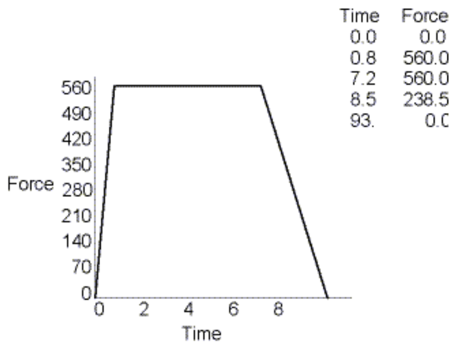
- A(1.5) evaluates to 20.0 (halfway between 12.0 and 28.0)
- A(1.75) evaluates to 24.0
- A(1.9) evaluates to 26.4

Similarly, if PQ is a TABLE array parameter

- PQ(1.5,1) evaluates to -3.4 (halfway between 2.8 and -9.6)
- PQ(1,1.5) evaluates to 3.5 (halfway between 2.8 and 4.2)
- PQ(3.5,1.3) evaluates to 14.88

This feature allows you to describe a *function*, such as y=f(x), using a TABLE array parameter. You would use the *j*=0 column for values of the independent variable x and the "regular" *j*=1 column for values of y. Consider, for example, a time-history forcing function described by five points as shown below.

Figure 3.10: Time-History Forcing Function



Time	Force
0.0	0.0
0.8	560.0
7.2	560.0
8.5	238.5
93.	0.C

You can specify this function as a TABLE array parameter whose array elements are the force values, and whose row index numbers 1 through 5 are time values 0.0 through 9.3. Schematically, the parameter then looks like this:

$$\text{FORCE} = \begin{array}{r|c|} & 0 \\ 1E-6 & 0.0 \\ 0.8 & 560.0 \\ 7.2 & 560.0 \\ 8.5 & 238.5 \\ 9.3 & 0.0 \end{array}$$

Mechanical APDL can calculate (through linear interpolation) force values at times not specified in the FORCE parameter. For the above example, Mechanical APDL calculates a value of 89.4375 for FORCE(9). If a parameter location beyond the dimensions of the array is used, no extrapolation is done and the end value is used. For example, Mechanical APDL provides a value of 560.0 for FORCE(5,2) or 0.0 for FORCE(12)

You can see from these examples that TABLE array parameters can be very powerful tools in your analysis. Typical applications are time-history loading functions, response spectrum curves, stress-strain curves, material-versus- temperature curves, B-H curves for magnetic materials, and so forth. Be aware that TABLE array parameters require more computer time to process than the ARRAY type.

3.10.5.7. Retrieving Values into or Restoring Array Parameter Values

You can issue the ***VGET** command, which is similar to ***GET**, to retrieve Mechanical APDL-provided values and store them in an array.

You must define a starting array location number for the array parameter the ***VGET** command creates. Looping continues over successive entity numbers for the *KLOOP* default. For example, ***VGET**,A(1),ELEM,5,CENT,X returns the centroid x-location of element 5 and stores the result in the first location of A. Retrieving continues with elements 6, 7, and so on until successive array locations are filled. In this example, if *KLOOP* is 4, then the centroid of x, y, and z are returned.

To restore array parameter values, use the ***VPUT** command.

The ***VPUT** command uses the same arguments as the ***VGET** command (described above), but does the opposite of the ***VGET** operation. For a list of valid labels for ***VPUT** items, see the command's description in the *Command Reference*.

The Mechanical APDL program "puts" vector items directly, without any coordinate system transformation. ***VPUT** can replace existing array items, but can't create new items. Degree of freedom results that are changed in the database are available for all subsequent operations. Other results change temporarily, and are available mainly for immediately following print and display operations.

Note:

Use this command with extreme caution, as it can alter entire sections of the database. The ***VPUT** command doesn't support all items on the ***VGET** item list because putting values into some locations could make the Mechanical APDL database inconsistent.

3.10.5.8. Listing Array Parameters

As with scalar parameters, you can use the ***STATUS** command to list array parameters. The following examples illustrate the ***STATUS** command in use:

```
*STATUS
 ABBREVIATION STATUS-

  ABBREV    STRING
  SAVE_DB   SAVE
  RESUM_DB  RESUME
  QUIT      Fnc_/EXIT
  POWRGRPH  Fnc_/GRAPHICS
  Mechanical APDLWEB  Fnc_HomePage

 PARAMETER STATUS-           (    5 PARAMETERS DEFINED)
                  (INCLUDING     2 INTERNAL PARAMETERS)

  NAME         VALUE         TYPE     DIMENSIONS
  MYCHAR      hi        CHARACTER
  MYPAR                     ARRAY       4       6       1
  MYPAR1    .987350000      SCALAR


*STATUS,XYZ(1),5,9      ! Lists rows 5 through 9 of XYZ
 PARAMETER STATUS- XYZ       (    4 PARAMETERS DEFINED)


      LOCATION          VALUE
      5     1     1   -8.98000000
      6     1     1    9.01000000
      7     1     1   -30.6000000
      8     1     1    51.0000000
      9     1     1   -51.9000000


*STATUS,FORCE(1),,,0    ! Lists parameter FORCE, includes j=0 column


PARAMETER STATUS- FORCE     (    4 PARAMETERS DEFINED)


      LOCATION          VALUE
      1     0     1   0.000000000E+00
      2     0     1   0.800000000
      3     0     1    7.20000000
      4     0     1    8.50000000
      5     0     1    9.30000000
      1     1     1   0.000000000E+00
      2     1     1    560.000000
      3     1     1    560.000000
      4     1     1    238.500000
      5     1     1   0.000000000E+00

*STATUS,T2(1,1)         ! Lists parameter T2

PARAMETER STATUS- T2        (   4 PARAMETERS DEFINED)

      LOCATION          VALUE
      1     1     1   0.600000000
      2     1     1    2.00000000
      3     1     1   -1.80000000
      4     1     1    4.00000000
      1     2     1    7.00000000
      2     2     1    5.00000000
      3     2     1    9.10000000
      4     2     1    62.5000000
      1     3     1   2.000000000E-04
      2     3     1   -3.50000000
      3     3     1    22.0000000
      4     3     1   1.000000000E-02

*STATUS,RESULT(1)!Lists parameter RESULT


PARAMETER STATUS- RESULT     ( 4 PARAMETERS DEFINED)

LOCATION     VALUE
1     1     1     SX(CHAR)
2     1     1     SY(CHAR)
3     1     1     SZ(CHAR)
```

3.10.6. Writing Data Files

You can write formatted data files (tabular formatting) from data held in arrays through the ***VWRITE** command. The command takes up to 10 array vectors as arguments and writes the data contained in those vectors to the currently open file (***CFOPEN** command). The format for each vector is specified with FORTRAN data descriptors on the line following the ***VWRITE** command (therefore you can't issue the ***VWRITE** command from the Mechanical APDL input window.)

An array vector, specified with a starting element location (such as MYARRAY(1,2,1)). You can also use an expression, which is evaluated as a constant value for that field in each row of the data file. The keyword SEQU evaluates to a sequential column of integers, starting from one.

The format of each row in the data file is determined by the data descriptor line. You must include one descriptor for each argument to the command. Do not include the word FORMAT in the descriptor line. You can use any real format or character format descriptor; however, you may not use either integer or list directed descriptors.

3.10.6.1. Format Data Descriptors

If you are unfamiliar with FORTRAN data descriptors, this section helps you to format your data file. For more information, consult the documentation for the FORTRAN compiler for your specific platform.

You must provide a data descriptor for each data item you specify as an argument to the ***VWRITE** command. In general, you can use the F descriptor (floating point) for any numeric values. The F descriptor takes the syntax

F*w*.*d*

where

w

Is the width of the data field in characters.

d

Is the number of digits to the right of the decimal point.

Thus, for a field that is 10 characters wide and has eight characters after the decimal point, you would use the following data descriptor:

```
F10.8
```

For character fields, you can use the A descriptor. The A descriptor has the syntax

A*w*

where

w

Is the width of the data field in characters.

Thus, for a character field that is eight characters wide, the descriptor is

```
A8
```

The following examples illustrate the ***VWRITE** command and data descriptors in use.

Given that the MYDATA array has been dimensioned and filled with the following values:

$$\text{MYDATA} = \begin{bmatrix} 2.15215183 & 3.89075020 & 5.28636971 & 7.15706483 & 13.7859423 & 87.4970443 \\ 2.30485343 & 4.44486730 & 5.40919563 & 7.68192625 & 15.5483820 & 86.5677915 \\ 2.01051819 & 3.39152436 & 5.93663807 & 7.38584253 & 18.4635868 & 45.7263566 \\ 2.36833012 & 3.32711472 & 5.63220341 & 7.22482004 & 18.7977889 & 39.7902425 \\ 2.84819512 & 4.76350638 & 5.97802354 & 7.29258882 & 14.8096356 & 62.0843906 \\ 2.22795343 & 3.48214546 & 5.54685145 & 7.90325139 & 14.0708891 & 37.6009897 \end{bmatrix}$$

The following short macro first defines the scalar parameter X as having a value of 25 and then opens the file vector (***CFOPEN** command). The ***VWRITE** command then defines the data to be written to the file. In this case, the first vector written uses the SEQU keyword to provide row numbers. Note that in some cases that constants, scalar parameters, and operations that include array element values are written to the file. Note the data file contents for these items.

```
x=25
*cfopen,vector
*vwrite,SEQU,mydata(1,1,1),mydata(1,2,1),mydata(1,3,1),10.2,x,mydata(1,1,1)+3
(F3.0,'  ',F8.4,'  ',F8.1,'  'F8.6,'  ',F4.1,'  'F4.0,'  'F8.1)
*cfclos
```

The macro creates the following data file:

```
 1.    2.1522       3.9  5.286370  10.2   25.       5.2
 2.    2.3049       4.0  5.409196  10.2   25.       5.2
 3.    2.0105       3.4  5.936638  10.2   25.       5.2
 4.    2.3683       3.3  5.632203  10.2   25.       5.2
 5.    2.8491       4.8  5.978024  10.2   25.       5.2
 6.    2.2280       3.5  5.546851  10.2   25.       5.2
```

The second example uses the following previously dimensioned and filled array:

$$\text{MYDATA} = \begin{bmatrix} 10 & 50 \\ 20 & 70 \\ 30 & 80 \end{bmatrix}$$

Note the use of descriptors in the following example ***VWRITE** command:

```
*vwrite,SEQU,mydata(1,1),mydata(1,2),(mydata(1,1)+mydata(1,2))
(' Row',F3.0,' contains ',2F7.3,'. Is their sum ',F7.3,' ?')
```

The resulting data file is

```
Row 1. contains  10.000 50.000.    Is their sum  60.000 ?
Row 2. contains  20.000 70.000.    Is their sum  60.000 ?
Row 3. contains  30.000 80.000.    Is their sum  60.000 ?
```

3.10.7. Operations Among Array Parameters

Just as parametric expressions and functions allow operations among scalar parameters, a series of commands is available to perform operations among array parameters. There are classes of operations: operations on columns (vectors), known as *vector operations* and operations on entire matrices (arrays), known as *matrix operations*. All operations are affected by a set of specification commands, which are discussed in Specification Commands for Vector and Matrix Operations (p. 42).

3.10.7.1. Vector Operations

Vector operations are simply a set of operations - addition, subtraction, sine, cosine, dot product, cross product, etc. - repeated over a sequence of array elements. Do-loops (discussed in Looping: Do-Loops (p. 74)) can be employed for this purpose, but a more convenient and much faster way is to use the vector operation commands - ***VOPER**, ***VFUN**, ***VSCFUN**, ***VITRP**, ***VFILL**, ***VREAD**, and ***VGET**. Of these listed vector operation commands, only ***VREAD** and ***VWRITE** are valid for character array parameters. Other vector operation commands apply only to array parameters dimensioned (***DIM**) as ARRAY type or TABLE type.

The ***VFILL**, ***VREAD**, ***VGET**, ***VWRITE**, and ***DIM** commands were introduced earlier in this chapter. Other commands that are discussed in this section include

***VOPER**

Performs an operation on two input array vectors and produces a single output array vector.

***VFUN**

Performs a function on a single input array vector and produces a single output array vector.

***VSCFUN**

Determines the properties of a single input array vector and places the result in a specified scalar parameter.

***VITRP**

Forms an array parameter (type ARRAY) by interpolating an array parameter (type TABLE) at specified table index locations.

The examples below illustrate the use of some of these commands. Refer to the *Command Reference* for syntactical information about these commands. For all of the following examples, the array parameters (of type ARRAY) X, Y, and THETA have been dimensioned and defined.

$$X = \begin{bmatrix} -2 & 6 & 8 & 0 \\ 1 & 0 & 2 & 12 \\ 4 & -3 & -1 & 7 \\ -8 & 1 & 10 & -5 \end{bmatrix} \qquad Y = \begin{bmatrix} 3 & 2 & 5 & -6 \\ -5 & -7 & 1 & 0 \\ 8 & 0 & 0 & 11 \\ 1 & 4 & 9 & 16 \end{bmatrix}$$

$$\text{THETA} = \begin{bmatrix} 0 \\ 15 \\ 30 \\ 45 \\ 60 \\ 75 \\ 90 \end{bmatrix}$$

In the following example, the result array is first dimensioned (Z1). The ***VOPER** command then adds column 2 of X to column 1 of Y, both starting at row 1, and then places the result into Z1. Notice that the starting location (the row and column index numbers) must be specified for all array parameters. The operation then progresses sequentially down the specified vector.

```
*DIM,Z1,ARRAY,4
*VOPER,Z1(1),X(1,2),ADD,Y(1,1)
```

$$Z1 = \begin{bmatrix} 9 \\ -5 \\ 5 \\ 2 \end{bmatrix}$$

In the following example, again the result array (Z2) is dimensioned first. The ***VOPER** command then multiplies the first column of X (starting at row 2) with the fourth column of Y (starting at row 1) and writes the results to Z2 (starting at row 1).

```
*DIM,Z2,ARRAY,3
*VOPER,Z2(1),X(2,1),MULT,Y(1,4)
```

$$Z2 = \begin{bmatrix} -6 \\ 0 \\ -88 \end{bmatrix}$$

In this example, again the results array (Z4) is dimensioned first. The ***VOPER** command then performs the cross product of four pairs of vectors, one pair for each row of X and Y. The *i*, *j*, and *k* components of these vectors are columns 1, 2, and 3 respectively of X and columns 2, 3, and 4 of Y. The results are written to Z4, whose *i*, *j*, and *k* components are vectors 1, 2, and 3 respectively.

```
*DIM,Z4,ARRAY,4,3
*VOPER,Z4(1,1),X(1,1),CROSS,Y(1,2)
```

$$Z4=\begin{bmatrix}-76 & 4 & -22\\ -2 & -14 & 1\\ -33 & -44 & 0\\ -74 & 168 & -76\end{bmatrix}$$

In the following example, the results array (A3) is dimensioned first. The ***VFUN** command then raises each element in vector 2 of X to the power of 2 and writes the results to A3.

```
*DIM,A3,ARRAY,4
*VFUN,A3(1),PWR,X(1,2),2
```

$$A3=\begin{bmatrix}36\\ 0\\ 9\\ 1\end{bmatrix}$$

In this example, the results array (A4) is dimensioned. The two ***VFUN** commands then calculate the cosine and sine of array elements in THETA and place the results in the first and second columns, respectively, of A4. Notice that A4 now represents a circular arc spanning 90°, described by seven points (whose x, y, and z global Cartesian coordinates are the three vectors). The arc has a radius of 1.0 and lies parallel to the x-y plane at z = 2.0.

```
*DIM,A4,ARRAY,7,3
*AFUN,DEG
*VFUN,A4(1,1),COS,THETA(1)
*VFUN,A4(1,2),SIN,THETA(1)
A4(1,3)=2,2,2,2,2,2,2
```

$$A4=\begin{bmatrix}1.0 & 0.0 & 2.0\\ 0.966 & 0.259 & 2.0\\ 0.866 & 0.5 & 2.0\\ 0.707 & 0.707 & 2.0\\ 0.5 & 0.866 & 2.0\\ 0.259 & 0.966 & 2.0\\ 0.0 & 1.0 & 2.0\end{bmatrix}$$

In this example, the results array (A5) is first dimensioned. Then, the ***VFUN** command calculates the tangent vector at each point on the curve represented by A4, normalizes it to 1.0, and places the results in A5.

```
*DIM,A5,ARRAY,7,3
*VFUN,A5(1,1),TANG,A4(1,1)
```

$$A5=\begin{bmatrix}-0.131 & 0.991 & 0\\ -0.259 & 0.965 & 0\\ -0.5 & 0.866 & 0\\ -0.707 & 0.707 & 0\\ -0.866 & 0.5 & 0\\ -0.966 & 0.259 & 0\\ -0.991 & 0.131 & 0\end{bmatrix}$$

Two additional ***VOPER** operations, gather (GATH) and scatter (SCAT), are used to copy values from one vector to another based on numbers contained in a "position" vector. The following example demonstrates the gather operation. Note that, as always, the results array must be dimensioned first. In the example, the gather operation copies the value of B1 to B3 (using the index positions specified in B2). Note that the last element in B3 is 0 as this is its initialized value.

```
*DIM,B1,,4
*DIM,B2,,3
*DIM,B3,,4
B1(1)=10,20,30,40
B2(1)=2,4,1
*VOPER,B3(1),B1(1),GATH,B2(1)
```

$$B3 = \begin{bmatrix} 20 \\ 40 \\ 10 \\ 0 \end{bmatrix}$$

3.10.7.2. Matrix Operations

Matrix operations are mathematical operations between numerical array parameter matrices, such as matrix multiplication, calculating the transpose, and solving simultaneous equations.

Commands discussed in this section include

***MOPER**

Performs matrix operations on one or more input array parameter matrices and produces an output array parameter matrix. Some of the available matrix operations are:

- Matrix multiplication
- Solution of simultaneous equations
- Sorting (in ascending order) on a specified vector in a matrix
- Covariance between two vectors
- Correlation between two vectors

See ***MOPER** for a complete list of operations.

***MFUN**

Copies or transposes an array parameter matrix (accepts one input matrix and produces one output matrix).

***MFOURI**

Calculates the coefficients for or evaluates a Fourier series.

The examples below illustrate the use of some of these commands. Refer to the *Command Reference* for syntactical information about these commands.

This example shows the sorting capabilities of the ***MOPER** command. For this example, assume that the array (SORTDATA) has been dimensioned and its element values have been defined as follows:

$$\text{SORTDATA} = \begin{bmatrix} 3 & 10 & 11 \\ 5 & -4 & 12 \\ 8 & -9 & 13 \\ 2 & 7 & 14 \\ 6 & 1 & 15 \end{bmatrix}$$

First, the OLDORDER array is dimensioned. The ***MOPER** command places the original order of the rows into OLDORDER. The ***MOPER** command then sorts the rows in SORTDATA so that the 1,1 vector is now in ascending order.

```
*dim,oldorder,,5
*moper,oldorder(1),sortdata(1,1),sort,sortdata(1,1)
```

The following array values result from the ***MOPER** command:

$$\text{SORTDATA} = \begin{bmatrix} 2 & 7 & 14 \\ 3 & 10 & 11 \\ 5 & -4 & 12 \\ 6 & 1 & 15 \\ 8 & -9 & 13 \end{bmatrix} \qquad \text{OLDORDER} = \begin{bmatrix} 4 \\ 1 \\ 2 \\ 5 \\ 3 \end{bmatrix}$$

To put the SORTDATA array back into its original order, you could then issue the following command:

```
*moper,oldorder(1),sortdata(1,1),sort,oldorder(1,1)
```

In the following example, the ***MOPER** command solves a set of simultaneous equations. The following two arrays have been dimensioned and their values assigned:

$$A = \begin{bmatrix} 2 & 4 & 3 & 2 \\ 3 & 6 & 5 & 2 \\ 2 & 5 & 2 & -3 \\ 4 & 5 & 14 & 14 \end{bmatrix} \qquad B = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 11 \end{bmatrix}$$

The ***MOPER** command can solve a set of simultaneous equations for a square matrix. The equations take the form

$a_{n1}X_1 + a_{n2}X_2 + , \ldots , + a_{nn}X_n = b_n$

In the case of the above arrays, the ***MOPER** command solves the following set of simultaneous equations:

$2X_1 + 4X_2 + 3X_3 +2X_4 = 2$

$3X_1 + 6X_2 + 5X_3 + 2X_4 = 2$

$2X_1 + 5X_2 + 2X_3 - 3X_4 =3$

$4X_1 + 5X_2 +14X_3 + 14X_4 = 11$

To solve the equations, first the results array (C) is dimensioned. Then the ***MOPER** command solves the equations, using A as the matrix of *a* coefficients and B as a vector of *b* values.

```
*DIM,C,,4
*MOPER,C(1),A(1,1),SOLV,B(1)
```

The C array now contains the following solutions.

$$C = \begin{bmatrix} -66 \\ 26 \\ 6 \\ 4 \end{bmatrix}$$

The following example shows the ***MFUN** command used to transpose data in an array. For this example, assume that the array (DATA) was dimensioned and filled with the following values:

$$DATA = \begin{bmatrix} 34 & 25 \\ 22 & 68 \\ -7 & 12 \end{bmatrix}$$

As always, the results array (DATATRAN) is dimensioned first, then the ***MFUN** command transposes the values and writes them to DATATRAN.

```
*DIM,DATATRAN,,2,3
*MFUN,DATATRAN(1,1),TRAN,DATA(1,1)
```

The following shows the results in the DATATRAN array:

$$DATATRAN = \begin{bmatrix} 34 & 22 & -7 \\ 25 & 68 & 12 \end{bmatrix}$$

3.10.7.3. Specification Commands for Vector and Matrix Operations

Vector and matrix operation commands are affected by the setting of the following array-specification commands: ***VCUM**, ***VABS**, ***VFACT**, ***VLEN**, ***VCOL**, and***VMASK**. See the table below for details.

Of these commands, only ***VLEN** and ***VMASK**, in conjunction with ***VREAD** or ***VWRITE**, are valid for character array parameters.

You can check the status of these commands with the ***VSTAT** command.

Important: All specification commands are reset to their default settings after each vector or matrix operation.

Following are descriptions of the array-specification commands:

***VCUM**

Specifies whether results are cumulative or noncumulative (overwriting previous results). *ParR*, the result of a vector operation, is either added to an existing parameter of the same name or overwritten. The default is noncumulative results, that is, *ParR* overwrites an existing parameter of the same name.

***VABS**

Applies an absolute value to any or all of the parameters involved in a vector operation. The default is to use the real (algebraic) value.

***VFACT**

Applies a scale factor to any or all of the parameters involved in a vector operation. The default scale factor is 1.0 (full value).

***VCOL**

Specifies the number of columns in matrix operations. The default is to fill all locations of the result array from the specified starting location.

***VSTAT**

Lists the current specifications for the array parameters.

***VLEN**

Specifies the number of rows to be used in array parameter operations.

***VMASK**

Specifies an array parameter as a masking vector.

The following table lists the various specification commands and the vector and matrix array commands that they affect.

	***VABS**	***VFACT**	***VCUM**	***VCOL**	***VLEN ,NROW,NINC**		***VMASK**
***MFOURI**	No	No	No	No	No	No	No
***MFUN**	Yes	Yes	Yes	Yes	Yes	No	No
***MOPER**	Yes	Yes	Yes	Yes	Yes	No	No
***VFILL**	Yes	Yes	Yes	N/A	Yes	Yes	Yes
***VFUN**	Yes	Yes	Yes	N/A	Yes	Yes	Yes
***VGET**	Yes	Yes	Yes	N/A	Yes	Yes	Yes
***VITRP**	Yes	Yes	Yes	N/A	Yes	Yes	Yes
***VOPER**	Yes	Yes	Yes	N/A	Yes	Yes	Yes
***VPLOT**	No	No	N/A	N/A	Yes	Yes	Yes
***VPUT**	Yes	Yes	No	N/A	Yes	Yes	Yes
***VREAD**	Yes	Yes	Yes	N/A	Yes	Yes	Yes
***VSCFUN**	Yes	Yes	Yes	N/A	Yes	Yes	Yes
***VWRITE**	No	No	N/A	N/A	Yes	Yes	Yes

The examples below illustrate the use of some of the specification commands. Refer to the *Command Reference* for syntactical information about these commands.

In the following, the results array (CMPR) is dimensioned. The two ***VFUN** commands, in conjunction with the preceding ***VMASK** and ***VLEN** commands, then compress selected data and write them to specified locations in CMPR. The complement to the COMP operation is the EXPA operation on the ***VFUN** command.

```
*DIM,CMPR,ARRAY,4,4
*VLEN,4,2! Do next *V---- operation on four rows,
! skipping every second row
*VFUN,CMPR(1,2),COMP,Y(1,1)
*VMASK,X(1,3)!Use column 3 of X as a mask for next *V----
! operation
*VFUN,CMPR(1,3),COMP,Y(1,2)
```

$$\text{CMPR} = \begin{bmatrix} 0 & 3 & 2 & 0 \\ 0 & 8 & -7 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This example uses the ***VFACT** command to round the values in an array vector to the number of decimal places specified by the NUMDP scalar parameter (set to 2 in the example). The NUMDATA array has been dimensioned and filled with the following values:

$$\text{NUMDATA} = \begin{bmatrix} 2.526 \\ 2.524 \\ -6.526 \\ -6.524 \end{bmatrix}$$

```
numdp=2
*vfact,10**numdp
*vfun,numdata(1),copy,numdata(1)
*vfun,numdata(1),nint,numdata(1)
*vfact,10**(-numdp)
*vfun,numdata(1),copy,numdata(1)
```

or, you can use a slightly shorter version

```
numdp=2
*vfact,10**numdp
*vfun,numdata(1),copy,numdata(1)
*vfact,10**(-numdp)
*vfun,numdata(1),nint,numdata(1)
```

The resultant NUMDATA array is then:

$$\text{NUMDATA} = \begin{bmatrix} 2.53 \\ 2.52 \\ -6.53 \\ -6.52 \end{bmatrix}$$

This example uses the ***VLEN** and ***VMASK** commands to find the set of prime numbers less than 100. An array, MASKVECT, is created using 1.0 to indicate that the row value is a prime number and 0.0 to indicate that the value isn't prime. The algorithm used to create the mask vector is to initialize all rows whose value is greater than 1 to 1.0 and then loop through the range of possible factors, eliminating all multiples of the factor. The ***VLEN** command sets the row increment for performing operations to FACTOR. When the ***VFILL** command is processed, the row number is incremented by this value. Because the starting row is FACTOR x 2, the rows are processed by each loop in the following manner: FACTOR x 2, FACTOR x 3, FACTOR x 4, etc.

```
*dim,maskvect,,100
*vfill,maskvect(2),ramp,1
*do,factor,2,10,1
*vlen,,factor
*vfill,maskvect(factor*2),ramp,0
*enddo
*vmask,maskvect(1)
```

```
*dim,numbers,,100
*vfill,numbers(1),ramp,1,1
*status,numbers(1),1,10
```

The resultant output from the ***STATUS** command, showing the first 10 elements in NUMBERS is:

```
PARAMETER STATUS- NUMBERS    (    5 PARAMETERS DEFINED)
                  (INCLUDING      2 INTERNAL PARAMETERS)

      LOCATION          VALUE
      1     1     1   0.000000000E+00
      2     1     1   2.00000000
      3     1     1   3.00000000
      4     1     1   0.000000000E+00
      5     1     1   5.00000000
      6     1     1   0.000000000E+00
      7     1     1   7.00000000
      8     1     1   0.000000000E+00
      9     1     1   0.000000000E+00
     10     1     1   0.000000000E+00
```

3.10.8. Plotting Array Parameter Vectors

You can graphically display array vector values using the ***VPLOT** command.

The following demonstrates some of the capabilities of the ***VPLOT** command. For this example, two TABLE arrays (TABLEVAL and TABLE) and one numeric array have been dimensioned and filled with the following values:

$$\text{TABLEVAL} = \begin{array}{c|cc|} 0 & 3 & 9 \\ \hline 4 & 6 & 12 \\ 7 & 8 & 6 \\ 15 & 10 & 3 \end{array} \qquad \text{TABLE2} = \begin{array}{c|c|} 0 & 40 \\ \hline 19 & 70 \\ 88 & 80 \\ 99 & 95 \end{array}$$

$$\text{ARRAYVAL} = \begin{bmatrix} 6 & 12 \\ 8 & 6 \\ 10 & 3 \end{bmatrix}$$

The following are example ***VPLOT** commands and their resulting plots. Note that since ARRAY data is unordered it is plotted as a histogram; TABLE data is ordered and is therefore plotted as a curve.

Figure 3.11: Example Plot Using `*VPLOT,,arrayval(1,1),2`

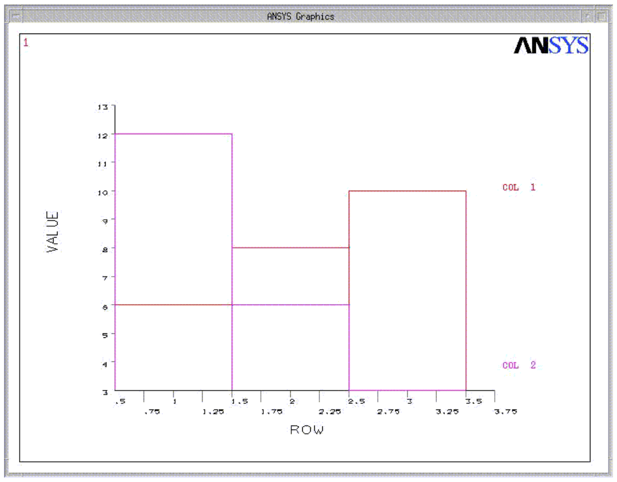


Figure 3.12: Example Plot Using `*VPLOT,,tableval(1,1),2`

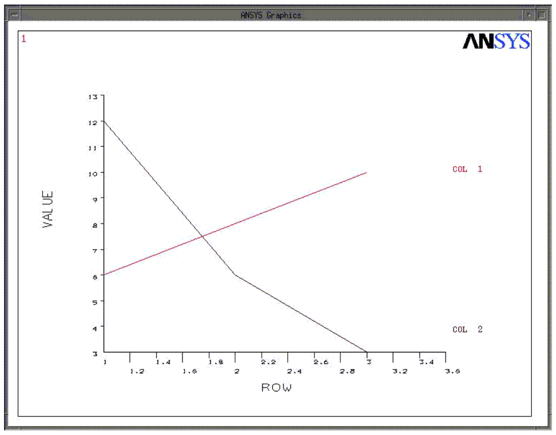


Figure 3.13: Example Plot Using `*VPLOT,table2(1),tableval(1,1),2`

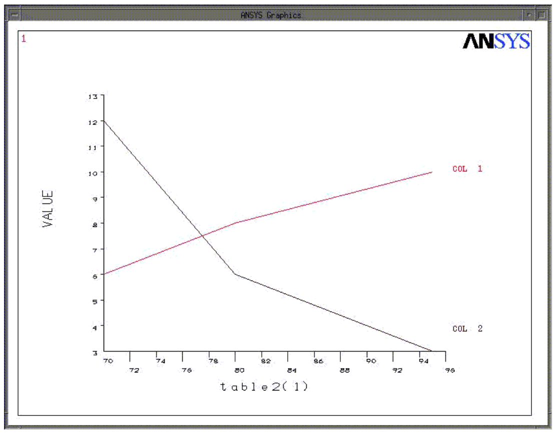
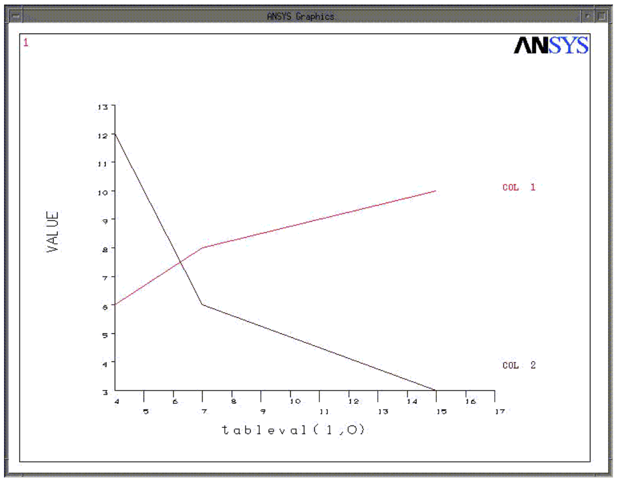


Figure 3.14: Example Plot Using `*VPLOT,tableval(1,0),tableval(1,1),2`



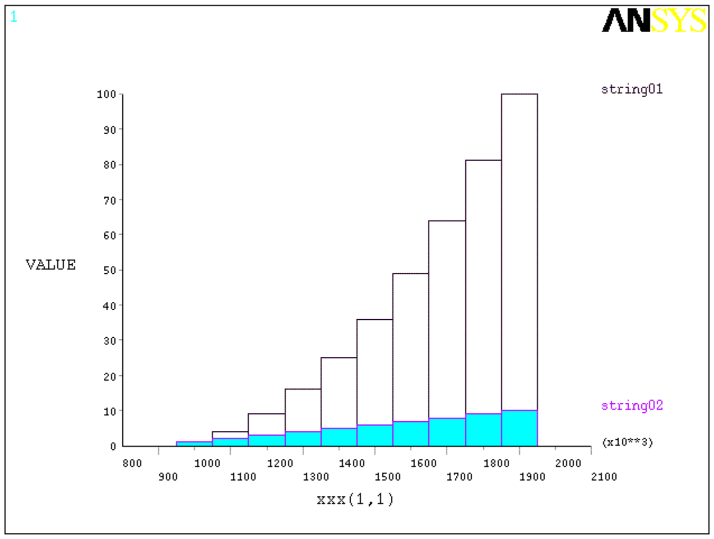
3.10.9. Modifying Curve Labels

When you use ***VPLOT** to create your curves, default labels are assigned. Normally, the label for curve 1 is "COL 1", the label for curve 2 is "COL 2" and so on; the column number is the field containing the dependent variables for that particular curve. You can use the **/GCOLUMN** command to apply your own labels to the curves (any string of up to eight characters).

The example below uses the **/GCOLUMN** command at the beginning of the program input to apply the labels "string01" and "string02" to the array curve.

```
/gcol,1,string01
/gcol,2,string02

*dim,xxx,array,10
*dim,yyy,array,10,2

xxx( 1,1) =1e6
xxx( 2,1) = 1e6 + 1e5
xxx( 3,1) = 1e6 + 2e5
xxx( 4,1) = 1e6 + 3e5
xxx( 5,1) = 1e6 + 4e5
xxx( 6,1) = 1e6 + 5e5
xxx( 7,1) = 1e6 + 6e5
xxx( 8,1) = 1e6 + 7e5
xxx( 9,1) = 1e6 + 8e5
xxx(10,1) = 1e6 + 9e5

yyy( 1,1) = 1
yyy( 2,1) = 4
yyy( 3,1) = 9
yyy( 4,1) = 16
yyy( 5,1) = 25
yyy( 6,1) = 36
yyy( 7,1) = 49
yyy( 8,1) = 64
yyy( 9,1) = 81
yyy(10,1) = 100

yyy( 1,2) = 1
yyy( 2,2) = 2
yyy( 3,2) = 3
yyy( 4,2) = 4
yyy( 5,2) = 5
yyy( 6,2) = 6
yyy( 7,2) = 7
yyy( 8,2) = 8
yyy( 9,2) = 9
yyy(10,2) = 10

*vplo,xxx(1,1), yyy(1,1) ,2
```

Figure 3.15: Example Plot With User-specified Labels



The labels can be returned to the default value (COL 1 and COL 2) by issuing the **/GCOLUMN** command with no string specified.

```
/gcol,1
/gcol,2
```

Chapter 4: APDL Math

APDL Math extends the APDL scripting environment of Mechanical APDL to give you access to the powerful matrix manipulation routines in the Mechanical APDL product, including its fast and efficient solvers. APDL Math provides access to matrices and vectors on the `.FULL`, `.EMAT`, `.MODE` and `.SUB` files, as well as other sources, so that you can read them in, manipulate them, and write them back out or solve them directly. This functionality augments the vector and matrix operations (such as ***VOPER** and ***MOPER**), in the standard APDL scripting environment. Both dense matrices and sparse matrices can be manipulated using APDL Math.

Differences Between Standard APDL and APDL Math

APDL Math works in its own workspace independent of the APDL environment. However, APDL arrays (vectors or matrices) may be imported into this workspace for manipulation, and also may be exported from this workspace into the standard APDL environment.

The following is a summary of the APDL Math commands:

Commands to create and delete matrices and vectors	
***DMAT**	Creates a dense matrix.
***SMAT**	Creates a sparse matrix.
***VEC**	Creates a vector.
***FREE**	Deletes a matrix or a solver object and frees its memory allocation.
Commands to manipulate matrices	
***AXPY**	Performs the matrix operation M2= v*M1 + w*M2.
***COMP**	Compresses the columns of a matrix using a specified algorithm.
***DOT**	Computes the dot (or inner) product of two vectors.
***FFT**	Computes the fast Fourier transformation of the specified matrix or vector.
***INIT**	Initializes a vector or dense matrix.
***MERGE**	Merges two dense matrices or vectors into one.
***MULT**	Performs the matrix multiplication M3 = M1(T1)*M2(T2).
***NRM**	Computes the norm of the specified vector or matrix.
***REMOVE**	Suppresses rows or columns of a dense matrix or a vector.
***RENAME**	Renames an existing vector or matrix.
***SCAL**	Scales a vector or matrix by a constant.
***SORT**	Sorts the values of the specified vector.
Commands to perform solutions	
***LSENGINE**	Creates a linear solver engine.
***LSFACTOR**	Performs the numerical factorization of a linear solver system.

***LSBAC**	Performs the solve (forward/backward substitution) of a factorized linear system.
***ITENGINE**	Performs a solution using an iterative solver.
***EIGEN**	Performs a modal solution with unsymmetric or damping matrices.
Commands to output matrices	
***EXPORT**	Exports a matrix to a file in the specified format.
***PRINT**	Prints the matrix values to a file.

The following topics are discussed in the remainder of this chapter:

4.1. Procedure for Using APDL Math

Typically, four steps are followed when using APDL Math.

1. **Create the matrices.**

   Matrices and vectors can be created by one of the following methods:

   - Generate matrices and vectors directly using APDL (***DIM**, ***SET**, etc.).
   - Mechanical APDL generates the matrices and vectors for you. These may come from a prior solution or may have been generated using the **WRFULL** command. The matrices are obtained from the `.FULL`, `.EMAT`, `.SUB`, `.MODE` or `.RST` files.
   - Use external sources provided in Harwell-Boeing or Matrix Market format.

2. **Read the matrices into APDL Math.**

   Once the files are available, you may import them into APDL Math using ***DMAT** for dense matrices, ***SMAT** for sparse matrices, and ***VEC** for vectors.

3. **Manipulate the matrices.**

   You can use the linear matrix operators ***MULT** and ***AXPY** to combine specified matrices and, thus, create other matrices. You may also modify the contents of matrices directly by using APDL expressions; for example, "A(3,2)=6.4".

   Additionally, you can use these steps to modify the matrices using standard APDL operations:

   a. Export the matrix via ***EXPORT**,,APDL.

   b. Modify the matrix using standard APDL operations such as ***SET**, ***MOPER**, ***VFUN**, ***DO**, etc.

   c. Re-import the matrix back into APDL Math via ***DMAT**,,,IMPORT,APDL (or ***VEC**,,,IMPORT,APDL for a vector).

4. **Use the matrices.**

   The modified matrices may be used in standard Mechanical APDL solutions, solved in APDL Math, or exported for use in an external application, as follows.

   - Use in standard Mechanical APDL solutions:
     - The simplest approach is to export the matrix as a superelement (***EXPORT**,,SUB) for use in any analysis.
   - Solve in APDL Math:
     - Identify the solver to be used with ***LSENGINE**.
     - Factor the matrix with ***LSFACTOR**.
     - Solve for the unknowns with **LSSOLVE**.
     - Once you do these steps, you can use ***ITENGINE** to use a PCG algorithm to find new solutions to a perturbed matrix (for instance, for parametric or sensitivity studies).
   - Export for use in an external application using ***EXPORT** with the Harwell-Boeing or Matrix Market formats.

4.2. Matrix and Vector Sizes

APDL Math automatically maintains APDL parameters corresponding to the size of every matrix and vector you create. The APDL parameter are created as follows:

- For an example matrix named "MyMatrix", the APDL parameters MyMatrix_rowDim and MyMatrix_colDim are automatically created.
- For matrices imported from the `.FULL` file, the APDL parameter MyMatrix_NUMDOF is automatically created, where NUMDOF is the number of degrees of freedom per node.
- For an example vector named "MyVector", the APDL parameter MyVector_Dim is automatically created.

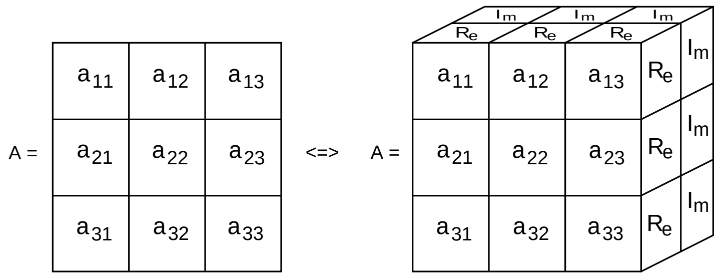
These APDL parameters are updated each time you manipulate these objects using APDL Math commands.

4.3. Accessing Complex Scalar Values

The technique described here allows you to access the values of complex vectors (***VEC**) or dense matrices (***DMAT**) and to manipulate either the real part or the imaginary part.

A complex dense matrix is a 2-D array of complex scalars. It can also be considered a 3-D array of real values as shown in the figure below:

Figure 4.1: 3-D Representation of a Complex Dense Matrix



The examples outlined below show several ways to access the complex scalar values.

- To get the real part of a dense matrix term, you can use the syntax:

```
VAL_RE = A(i,j,1)
VAL_IM = A(i,j,2)
```

- You can also set a new value directly, using the same logic:

```
A(i,j,1) = 3.5
A(i,j,2) = 0.0
```

- Similarly, you can access the values of a complex vector:

```
VAL_RE = V(i,1)
VAL_IM = V(i,2)
```

4.4. Degree of Freedom Ordering

Degrees of freedom (DOFs) are numbered 1-*n*, where *n* is the total number of DOFs in the system. Mechanical APDL maintains three orderings of the DOF set:

1. The **user ordering** is based on the user's node numbers. As an example, this ordering corresponds to what would be printed in the POST1 postprocessor by the **PRNSOL**,U command.

2. The **internal ordering** is obtained by compressing the unused node numbers from the user's set and renumbering them based on the reordered element set. This reordering is performed to obtain better cache performance as these items are referenced in the solution phase. The map between the user ordering and this internal ordering is referred to as the "nodal equivalence table" in the descriptions of binary data files (see Format of Binary Data Files in the *Programmer's Reference*).

3. The **solver ordering** is obtained by reordering the equations (DOFs) in order to minimize the solver time and disk requirements. Also, the effects of any constraints (**D** command), couplings (**CP** command), and constraint equations (**CE** command or MPC contact) are accounted for, thereby reducing the DOF set. This ordering represents the "independent" DOFs of the system.

The matrices and load vector imported from the `.FULL` file (***SMAT**,,,FULL) are in terms of the solver ordering; mapping from the internal ordering to this ordering is required when working with these matrices.

The resulting solution from ***LSBAC** will also be in this solver ordering.

The mode shapes from the `.MODE` file and the DOF results from the `.RST` file are in the internal ordering, and they need to be converted before use with any of the matrices from the `.FULL` file, as shown below:

```
*SMAT,Nod2Solv,D,IMPORT,FULL,file.full,NOD2SOLV  ! import the mapping vector
*DMAT,PhiI,D,IMPORT,MODE,file.mode               ! import the mode shapes
*MULT,Nod2Solv,,PhiI,,PhiB                       ! convert to the solver set
```

To convert from solver to internal ordering (for example, after an ***LSBAC** solution), use the transpose of the NOD2SOLV mapping vector:

```
*MULT,Nod2Solv,TRAN,Xsolv,,Xint
```

To convert from external (user) ordering to the internal ordering, use the FORWARD nodal mapping vector. The following example retrieves the UZ displacement of user node 45232 from the internal solution vector Xint:

```
*VEC,MapForward,I,IMPORT,FULL,file.full,FORWARD
j = MapForward(45232)
UzVal = Xint((j-1)*NUMDOF + 3)            ! 3 is the UZ DOF number
```

To convert from internal ordering to external (user), use the BACK nodal mapping vector. The following example returns the force applied on user-defined node j having internal node number 672:

```
*VEC,MapBack,I,IMPORT,FULL,file.full,BACK
myforce = Fint((672-1)*NUMDOF + 3)
j = MapBack(672)
```

To convert this to solver ordering for solving (***LSBAC**):

```
*MULT,Nod2Solv,,Fint,,Fsolv
```

4.5. Creating a User Superelement

One way to add user-defined behavior to a model is via superelements. APDL Math allows you to import, modify, and create superelement matrices that you can then export to a superelement `.SUB` file for use in subsequent analyses.

Importing from existing `.SUB` files or from NASTRAN `DMIG` files is straightforward. These matrices can be modified using existing APDL or APDL Math operations and the modified matrix exported to a new `.SUB` file. Note that for a `DMIG` file export, you must supply the nodal coordinates.

Creating a `.SUB` file from scratch requires additional information. You must create (***DIM**) an *m* x 2 array, where "*m*" is the number of master DOFs (rows in the matrix). In the first column, put the node number of the master DOF, and in the second column put the DOF number. This array is then passed to the ***EXPORT**,,SUB command. For example:

```
/prep7
! Provide coordinates for SUB file
N,11
```

```
N,12,1

! Allocate the matrix and define the values
*DMAT,myk,d,alloc,4,4
myk(1,1)=1.0
myk(2,2)=1.0
myk(3,3)=1.0
myk(4,4)=1.0
myk(1,3)=-0.5
myk(3,1)=-0.5

! Allocate the row information array and define its values
*DMAT,rowinfo,i,alloc,4,2
rowinfo(1,1)=11,11,12,12       ! nodes
rowinfo(1,2)=1,2,1,2           ! dofs
*PRINT,myk
*PRINT,rowinfo

! Export to a SUB file
*EXPORT,myk,sub,mysub.sub,stiff,rowinfo,done

! List its contents
SELIST,mysub,3
```

4.6. Creating a Sparse Matrix Using the CSR Format

The ***SMAT** command has an option to allocate a sparse matrix using the CSR format. This format represents a matrix by three one-dimensional arrays. The arrays are specified in the `Val2`, `Val3`, and `Val4` fields of ***SMAT** and contain the following data:

1st array: nonzero values

2nd array: the extents of rows (this vector stores the locations in the values vector that start a row)

3rd array: column indices

For example, consider this 4 x 4 matrix **M**:

$$\begin{pmatrix} 1 & 2 & 4 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

The matrix can be described by these three vectors:

`A = [ 1 2 4 3 5 -1 2 ]`	This array is of length *NNZ* (number of nonzero values) and holds all the nonz
`IA = [ 1 4 5 6 8 ]`	This array is of length *m*+1 (*m* is the size of the matrix), and is defined by the re IA[1] = 1 IA[*i*] = IA[*i*-1] + number of nonzero values on the (*i*-1) row IA[*m*+1] = *NNZ*+1
`JA = [ 1 2 3 2 3 3 4 ]`	This array is of length *NNZ* and holds the column indices of the nonzero value

The following APDL commands create the above matrix.

```
DIM=4
NNZ=7
```

```
*VEC,A,D,ALLOC,NNZ
A(1)=1,2,4,3,5,-1,2

*VEC,IA,L,ALLOC,DIM+1
IA(1)=1,4,5,6,8

*VEC,JA,I,ALLOC,NNZ
JA(1)=1,2,3,2,3,3,4

*SMAT,M,D,ALLOC,CSR,IA,JA,A,FALSE
*PRINT,M
```

4.7. Restrictions and Recommendations for Matrix Operations

The following restrictions and recommendations apply when using APDL Math matrix operations:

- You cannot directly modify a sparse matrix in APDL Math. You must export it from APDL Math to modify it:
  - Export the matrix in an ASCII format (for example, ***EXPORT**,,MMF).
  - Edit the file.
  - Import the matrix back in (***SMAT**,,,IMPORT,MMF).
- When working with matrices and vectors, care must be taken to insure that any operations are done using consistent ordering (see Degree of Freedom Ordering (p. 52)).

4.8. APDL Math Examples

The following examples demonstrate various applications for APDL Math.

Example 4.1: Verify Orthogonality of Eigenmodes after a Modal Analysis

```
! PERFORM A STANDARD MODAL ANALYSIS
/SOLU
MODOPT,lanb,10
SOLVE
FINISH


! READ M AND Nod2Solv MATRICES FROM THE FULL FILE
*SMAT,MassMatrix,D,IMPORT,FULL,file.full,MASS
*SMAT,NodToSolv,D,IMPORT,FULL,file.full,NOD2SOLV

! READ THE MODE SHAPES FROM THE MODE FILE
*DMAT,Phi,D,IMPORT,MODE,file.mode

! CONVERT THEM TO THE SOLVER ORDERING
*MULT,NodToSolv,,Phi,,SOLVPhi

! CREATE PhiTMPhi = (Phi)T*M*Phi
*MULT,MassMatrix,,SOLVPhi,,APhi
*MULT,SOLVPhi,TRANS,APhi,,PhiTMPhi

! PRINT THIS MATRIX: IT SHOULD BE THE IDENTITY MATRIX [I]
*PRINT,PhiTMPhi,PhiTMPhi.txt
```

Example 4.2: Read a Matrix and a Load Vector from a FULL File and Solve

```
! READ THE STIFFNESS MATRIX FROM THE FULL FILE
*SMAT,MatK,D,IMPORT,FULL,file.full,STIFF

! READ THE MAPPING TABLE: INTERNAL -> SOLV
*SMAT,Nod2Solv,D,IMPORT,FULL,file.full,NOD2SOLV

! READ THE LOAD VECTOR FROM THE FULL FILE
*DMAT,VecB,D,IMPORT,FULL,file.full,RHS

! ALLOCATE THE SOLUTION VECTOR IN SOLVER SPACE BY SIMPLY COPYING B
*DMAT,VecX,D,COPY,VecB

! FACTORIZE A USING THE SPARSE SOLVER FUNCTIONS
*LSENGINE,BCS,MyBcsSolver,MatK
*LSFACTOR,MyBcsSolver

! SOLVE THE LINEAR SYSTEM
*LSBAC,MyBcsSolver,VecB,VecX

! CONVERT THE SOLUTION TO THE INTERNAL SPACE
*MULT,Nod2Solv,T,VecX,,XNod

! PRINT THE SOLUTION
*PRINT,XNod

! FREE ALL OBJECTS
*FREE,ALL
```

Example 4.3: Perform a Full Harmonic Sweep

```
! READ THE 3 MATRICES FROM THE FULL FILE
*SMAT,MatK,D,IMPORT,FULL,file.full,STIFF
*SMAT,MatM,D,IMPORT,FULL,file.full,MASS
*SMAT,MatC,D,IMPORT,FULL,file.full,DAMP

! READ THE MAPPING TABLE: FULL -> SOLV
*SMAT,Nod2Solv,D,IMPORT,FULL,file.full,NOD2SOLV

! READ THE LOAD VECTOR FROM THE FULL FILE
*DMAT,VecB,Z,IMPORT,FULL,file.full,RHS

! ALLOCATE THE SOLUTION VECTOR IN SOLVER SPACE BY SIMPLY COPYING B
*DMAT,XSolv,Z,COPY,VecB

C=3.E8  ! LIGHT CELERITY

*DO,FREQ,1.E9,10.E9,1.E9  ! LOOP OVER FREQUENCY VALUES

  /com,** FREQUENCY = %FREQ%

  w=2*3.14*FREQ/C   ! COMPUTE OMEGA (w)
  w2=w*w     ! w*w

  ! FORM THE COMPLEX SYSTEM  A = K - w2*M + jw*C
  *SMAT,MatA,Z,COPY,MatK
  *AXPY,-w2,0.,MatM,1.,0.,MatA
  *AXPY,0.,w,MatC,1.,0.,MatA

  ! FACTORIZE MATRIX A USING SPARSE SOLVER
  *LSENGINE,BCS,MyBcsSolver,MatA
  *LSFACTOR,MyBcsSolver

  ! SOLVE THE LINEAR SYSTEM
  *LSBAC,MyBcsSolver,VecB,XSolv
*ENDDO
*FREE,ALL
```

Example 4.4: Perform an UNSYM Modal Solve from a FULL File

```
! DEFINE THE ANALYSIS OPTIONS
/SOLU
ANTYPE,MODAL
MODOPT,UNSYM,10,-3

! LOAD K AND M MATRICES FROM AN EXISTING FULL FILE
*SMAT,MatK,D,IMPORT,FULL,file.full,STIFF
*SMAT,MatM,D,IMPORT,FULL,file.full,MASS

! LAUNCH THE UNSYM ALGORITHM, APPLIED TO THE GIVEN MATRICES
*EIGEN,MatK,MatM,,EiV,EiM
*PRINT,EiV
FINISH
```

Example 4.5: Perform a DAMP Modal Solve from HBMAT Files

```
! DEFINE THE ANALYSIS OPTIONS
/SOLUTION
ANTYPE,MODAL
MODOPT,DAMP,10

! LOAD K, M and C MATRICES FROM EXISTING HBMAT ASCII FILES
*SMAT,MatK,D,IMPORT,HBMAT,K.hbmat,ASCII
*SMAT,MatM,D,IMPORT,HBMAT,M.hbmat,ASCII
*SMAT,MatC,D,IMPORT,HBMAT,C.hbmat,ASCII

! LAUNCH THE DAMP ALGORITHM, APPLIED TO THE GIVEN MATRICES
*EIGEN,MatK,MatM,MatC,EiV,EiM

*PRINT,EiV
FINISH
```

Example 4.6: Import a Matrix from a .SUB File, Modify the Values, and Update the File

```
! LOAD THE K MATRIX FROM A SUB FILE
*DMAT,MatK,D,IMPORT,SUB,file.sub,STIFF
*PRINT,MatK

! EXPORT THE MATRIX AS A STANDARD APDL ARRAY
*EXPORT,MatK,APDL,MATKAPDL

! MODIFY THE MATRIX
MATKAPDL(1,1) = 5.0

! IMPORT THE MODIFIED MATRIX INTO APDL MATH SPACE
*DMAT,MatK,,IMPORT,APDL,MATKAPDL

! EXPORT THE MODIFIED MATRIX TO THE SUB FILE
*EXPORT,MatK,SUB,file.sub,STIFF
```

Example 4.7: Calculate the Complex Mode Contribution Coefficients (CMCC)

APDLMath is used to calculate the CMCC based on Equation 3.1 in the *Structural Analysis Guide*. The real modes are read from the `Jobname.MODESYM` mode file, the mass matrix from the `file.full` file, and the complex modes from the `Jobname.MODE` file. The resulting CMCC are printed out in the ASCII file `Cmcc.txt`. If the file `Cmcc.txt` already exists, the new coefficients will be appended to this file.

```
! -------------------------------------------------------------
! GET THE MASS MATRIX FROM FILE.FULL
! -------------------------------------------------------------
```

```
*SMAT,Mass,D,IMPORT,FULL,file.full,MASS
! GET THE FULL TO SOLV MAPPING
*SMAT,NodToSolv,D,IMPORT,FULL,file.full,NOD2SOLV

! ------------------------------------------------------------
! GET THE COMPLEX MODES FROM FILE.MODE  : PhiC
! ------------------------------------------------------------

*DMAT,PhiF,Z,IMPORT,MODE,file.mode,1,300
*MULT,NodToSolv,,PhiF,,PhiC
*FREE,PhiF

! ------------------------------------------------------------
! GET THE REAL MODES FROM FILE.MODESYM    : PhiR
! ------------------------------------------------------------

*DMAT,PhiF,,IMPORT,MODE,file.modesym,1,300
*MULT,NodToSolv,,PhiF,,PhiR
*FREE,PhiF

! ------------------------------------------------------------
! COMPUTE AND NORMALIZE THE CMCC : PhiR(T).M.PhiC
! ------------------------------------------------------------

*MULT,Mass,,PhiC,,MPhiC            ! MPhiC = M.PhiC
*MULT,PhiR,T,MPhiC,,PhiRMPhiC      ! PhiRMPhiC = PhiR(T).MPhiC

*DO,ii,1,PhiRMPhiC_colDim,1        ! LOOP OVER ALL COLUMNS
 *VEC,v,z,LINK,PhiRMPhiC,ii        ! V = LINK TO iith Column
 *VEC,vr,d,COPY,v
 *NRM,vr,NRMINF,_vr_nrm
 *AXPY,,,,1./_vr_nrm,,v            ! V is normalized / NRM_INF(V)=1.
*ENDDO

*PRINT,PhiRMPhiC,Cmcc.txt          ! PRINT -> Cmcc.txt
```

Example 4.8: Import Matrices from Nastran DMIG Files and Create SUB Files for Mechanical APDL

Important:

When importing matrices from Nastran **DMIG** files, it is a requirement that the generalized coordinates for a CMS superelement appear last (SPOINTS must have highest ID number). Otherwise, the SUB file, generated from the imported matrices, will not be generated properly.

```
! DEFINITION OF NODES, ELEMENTS
....
```

Nodes must be defined in Mechanical APDL to match Nastran Data to the Mechanical APDL model.

```
! IMPORT A STIFFNESS MATRIX FROM A NASTRAN DMIG FILE
*DMAT,KMat,D,IMPORT,DMIG,fileK.dmig

! IMPORT A MASS MATRIX FROM ANOTHER NASTRAN DMIG FILE
*DMAT,MMat,D,IMPORT,DMIG,fileM.dmig
```

The matrices must be in different files.

```
! ACCESS THE MATRICES VALUES IF NEEDED
*PRINT,KMat
```

```
KMat(1,1) = KMat(1,1)*2
...


! GENERATE A NEW SUB FILE WITH THESE 2 MATRICES
*EXPORT,KMat,SUB,newfile.sub,STIFF,,WAIT
*EXPORT,MMat,SUB,newfile.sub,MASS,,DONE
```

The two matrices are dumped into one single SUB file. The file is generated at the "DONE" keyword.

Example 4.9: Calculate the Participation Factors and Total Rigid Body Mass

APDL Math is used to calculate the participation factors and total rigid body mass based on the database file (`test.db`), the full file (`test.full`), and the mode file (`test.mode`) from a modal analysis. This procedure is particularly useful if the effect of boundary conditions and CP/CE is required.

```
/filname,test
resume,, db

! Generation of the rigid body motion vectors = rig_apdl
*get,numDof,common,,dofcom,,int,1
*get,maxNod,NODE,0,NUM,MAX,,,INTERNAL
dim1 = numDof*maxNod
dim2 = 6
*dim,rig_apdl,ARRAY,dim1,dim2
*vfill,rig_apdl,RIGID

! Get the name of the files
*dim,jobcurr,STRING,248
jobcurr(1)=''
*dim,jobcurrfull,STRING,248
jobcurrfull(1)=''
*dim,jobcurrmode,STRING,248
jobcurrmode(1)=''
*do,i,1,248,8
   *get,param,ACTIVE,0,JOBNAME,,START,i
   jobcurr(1) = strcat(jobcurr(1),param)
*enddo
jobcurrfull(1) = strcat(jobcurr(1),'.full')
jobcurrmode(1) = strcat(jobcurr(1),'.mode')

! Calculation of pfall = PhiRT x mass x rig
*smat,mass,D,IMPORT,FULL,jobcurrfull(1),MASS
*smat,NodToSolv,D,IMPORT,FULL,jobcurrfull(1),NOD2SOLV
*smat,usrtosolv,D,IMPORT,FULL,jobcurrfull(1),USR2SOLV

*dmat,rig,D,IMPORT,APDL,rig_apdl
*mult,usrtosolv,,rig,,rigsolv

*mult,mass,,rigsolv,,prodr

*dmat,PhiF,,IMPORT,MODE,jobcurrmode(1),1,4
*mult,NodToSolv,,PhiF,,PhiR
*free,PhiF

*mult,PhiR,T,prodr,,pfall
*print,pfall                   ! participation factors

*mult,rigsolv,T,prodr,,mtot
*print,mtot                   ! total rigid body mass
```

Chapter 5: APDL as a Macro Language

You can record a frequently used sequence of Mechanical APDL commands in a macro file (these are sometimes called command files). Creating a macro enables you to, in effect, create your own custom Mechanical APDL command. For example, calculating power loss due to eddy currents in a magnetic analysis would require a series of Mechanical APDL commands in the postprocessor. By recording this set of commands in a macro, you have a new, single command that executes all of the commands required for that calculation. In addition to executing a series of Mechanical APDL commands, a macro can call GUI functions or pass values into arguments.

You can also nest macros. That is, one macro can call a second macro, the second macro can call a third macro, and so on. You can use up to 20 nesting levels, including any file switches caused by the Mechanical APDL **/INPUT** command. After each nested macro executes, the Mechanical APDL program returns control to the previous macro level.

The following is a very simple example macro file. In this example, the macro creates a block with dimensions 4, 3, and, 2 and a sphere with a radius of 1. It then subtracts the sphere from one corner of the block.

```
/prep7
/view,,-1,-2,-3
block,,4,,3,,2
sphere,1
vsbv,1,2
finish
```

If this macro were called `mymacro.mac`, you could execute this sequence of commands with the following single Mechanical APDL command

```
*use,mymacro
```

or (because the extension is `.mac`)

```
mymacro
```

Although this is not a realistic macro, it does illustrate the principle.

This chapter provides information on the various ways you can create, store, and execute macros. It also discusses the basic information you need to use APDL as a scripting language in creating macros. APDL commands used to define and execute macros are listed in Chapter 2 of the *Command Reference*.

The following specific macro topics are available:

5.1. Creating a Macro

You can create macros either within Mechanical APDL itself or using your text editor of choice (such as emacs, vi, or wordpad). If your macro is fairly simple and short, creating it in Mechanical APDL can be very convenient. If you are creating a longer, more complex macro or editing an existing macro then you will need a text editor. Also, using a text editor allows you to use a similar macro or Mechanical APDL log file as the source for your macro.

For any long, complex macro you should always consider either using a similar macro as a starting point or running the task interactively in Mechanical APDL and using the resulting log file as the basis of your macro. Either method can greatly reduce the time and effort required to create a suitable macro.

The following creating macro topics are available:

5.1.1. Macro File Naming Conventions

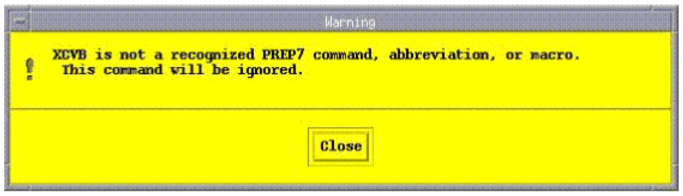
Macros are a sequence of Mechanical APDL commands stored in a file. Macros should not have the same name as an existing Mechanical APDL command, or start with the first four characters of a Mechanical APDL command, because Mechanical APDL executes the internal command instead of the macro. The following naming restrictions apply to macro files:

- The file name cannot exceed 32 characters.
- The file name cannot begin with a numeral.
- The file extension cannot contain more than eight characters (if you are executing the macro as if it were a Mechanical APDL command it should have the extension `.mac`.)
- The file name or extension cannot contain spaces.
- The file name or extension cannot contain any characters prohibited by your file system and for portability should not contain any characters prohibited by either Linux or Windows file systems.

To ensure that you are not using the name of a Mechanical APDL command, before creating a macro try running the file name that you wish to use as a Mechanical APDL command. If Mechanical APDL returns the message shown below, you will know that the command is not used in the current processor. You should check the macro file name in each processor in which you plan to use the macro.

(You could also check if the macro file name matches any command listed in the online documentation; however, this method cannot locate the names of undocumented commands.)

Figure 5.1: Mechanical APDL Message Box for Unknown Command



```
XCVB is not a recognized PREP7 command, abbreviation, or macro.
 This command will be ignored.
```

Using the `.mac` extension allows Mechanical APDL to execute the macro as it would any internal command. You should avoid using the extension `.MAC` because it is used for Mechanical APDL internal macros.

5.1.2. Macro Search Path

By default, Mechanical APDL searches for a user macro file (`.mac` extension) in the following locations:

1. The `/ansys_inc/v211/ansys/apdl` directory.

2. The directory (or directories) designated by the **ANSYS_MACROLIB** environment variable (if defined) or the login (home) directory. This environment variable is documented in The Mechanical APDL Environment chapter of the *Operations Guide*.

3. The directory designated by the **$HOME** environment variable.

4. The working directory.

You can place macros for your personal use in your home directory. Macros that should be available across your site should be placed in the `/ansys_inc/v211/ansys/apdl` directory or some commonly accessible directory that everyone can reference through the **ANSYS_MACROLIB** environment variable.

For Windows users: The "current directory" is the default directory (usually a network resource) set by administrators and you should ask your network administrator for its location. You can use environment variables to create a local "home directory." The local home directory is checked after the default directory designated in your domain profile.

5.1.3. Creating a Macro Within Mechanical APDL

You can create a macro by four methods from within Mechanical APDL:

- Issue the ***CREATE** command in the input window. Parameter values are not resolved and parameter names are written to the file.

- Use the ***CFOPEN**, ***CFWRITE**, and ***CFCLOS** commands. Parameter names are resolved to their current values and those values are written to the macro file.

- Issue the **/TEE** command in the input window. This command writes a list of commands to a file at the same time that the commands are being executed. As the commands are executed in the current Mechanical APDL session, parameter names are resolved to their current values.

However, in the file that is created, parameter values are not resolved and parameter names are written instead.

- Choose the **Utility Menu> Macro> Create Macro** menu item. This method opens a dialog box that can be used as a simple, multiline editor for creating macros. Parameter values are not resolved and parameter names are written to the file.

The following sections detail each of these methods.

5.1.3.1. Using *CREATE

Issuing ***CREATE** redirects Mechanical APDL commands entered in the command input window to the file designated by the command. All commands are redirected until you issue the ***END** command. If an existing file has the same name as the macro file name you specify, the Mechanical APDL program overwrites the existing file.

For example, suppose that you want to create a macro called `matprop.mac`, which automatically defines a set of material properties. The set of commands entered into the input window for this macro might look like this:

```
*CREATE,matprop,mac,macros
MP,EX,1,2.07E11
MP,NUXY,1,.27
MP,DENS,1,7835
MP,KXX,1,42
*END
```

The ***CREATE** command takes arguments of the file name, the file extension, and the directory path (in this case, the `macros` directory is specified).

When using ***CREATE**, all parameters used in commands are written to the file (the currently assigned values for the parameter are not substituted).

You cannot use ***CREATE** within a DO loop.

5.1.3.2. Using *CFWRITE

If you wish to create a macro file in which current values are substituted for parameters you can use ***CFWRITE**. Unlike ***CREATE**, the ***CFWRITE** command cannot specify a macro name; you must first specify the macro file with the ***CFOPEN** command. Only those Mechanical APDL commands that are explicitly prefaced with a ***CFWRITE** command are then written to the designated file; all other commands entered in the command input window are executed. As with the ***CREATE** command, ***CFOPEN** can specify a file name, a file extension, and a path. The following example writes a **BLOCK** command to the currently open macro file.

```
*cfwrite,block,,a,,b,,c
```

Note that parameters were used for arguments to the **BLOCK** command. The current value of those parameters (and not the parameter names) are written to the file. So, for this example, the line written to the macro file might be

```
*cfwrite,block,,4,,2.5,,2
```

To close the macro file, issue the ***CFCLOS** command.

Note:

While it is possible to create a macro through this method, these commands are most useful as a method for writing Mechanical APDL commands to a file during macro execution.

5.1.3.3. Using /TEE

Issuing **/TEE**,NEW or **/TEE**,APPEND redirects Mechanical APDL commands entered in the command input window to the file designated by the command *at the same time that the commands are being executed*. All commands are executed and redirected until you issue the **/TEE**,END command. If an existing file has the same name as the macro file name you specify with **/TEE**,NEW, the Mechanical APDL program overwrites the existing file. To avoid this, use **/TEE**,APPEND instead.

In addition to the `Label` argument (which can have a value of NEW, APPEND, or END), the **/TEE** command takes arguments of the file name, the file extension, and the directory path.

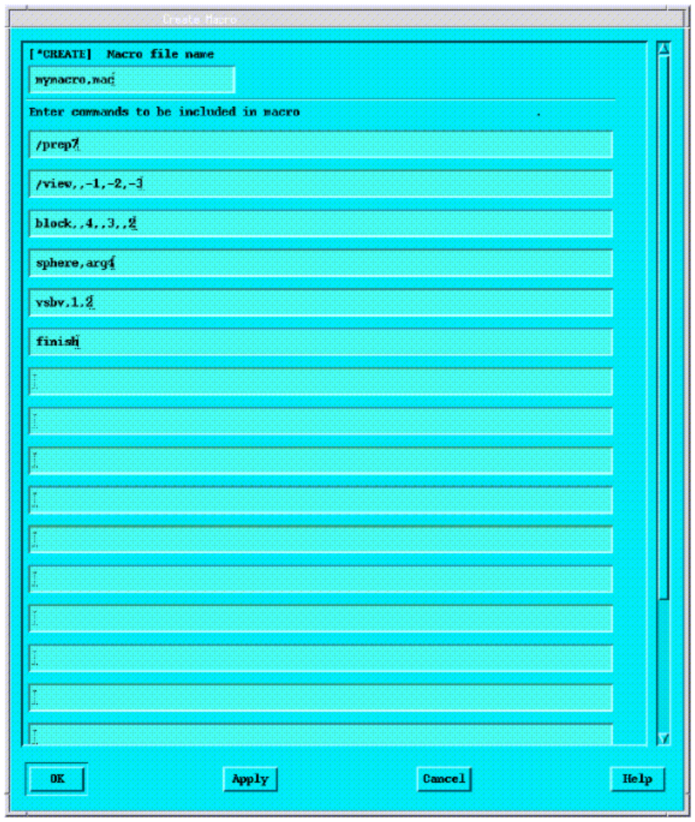
As the commands are executed in the current Mechanical APDL session, all parameter names are resolved to their current values. However, in the file that is created, parameter names are written (the currently assigned values for the parameter are not substituted). If your current parameter values are important, you can save the parameters to a file using the **PARSAV** command.

For an example, see the description of the **/TEE** command in the *Command Reference*.

5.1.3.4. Using Utility Menu> Macro> Create Macro

Choosing this menu item opens a Mechanical APDL dialog box that you can use as a simple editor for creating macros. You cannot open and edit an existing macro with this facility; if you use the name of an existing macro as the arguments for the ***CREATE** field, the existing file will be overwritten.

Figure 5.2: The Create Menu Dialog Box



```
[*CREATE]  Macro file name
mymacro.mac
Enter commands to be included in macro
/prep7
/view,,-1,-2,-3
block,,4,,3,,2
sphere,arg1
vsbv,1,2
finish
```

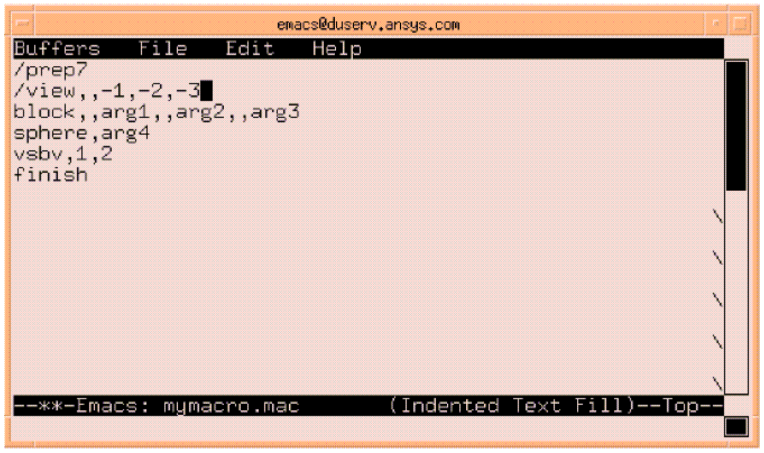
As with the ***CREATE** command, parameters are not evaluated but are written verbatim into the macro file. Note that you do not make the last line a ***END** command.

5.1.4. Creating Macros with a Text Editor

You can use your favorite text editor to create or edit macro files. Any ASCII editor will work. Moreover, Mechanical APDL macros can have their lines terminated by either Linux or Windows line ending conventions (carriage-return, line-feed pairs or simply line-feeds) so you can create a macro on one platform and use it on several platforms.

If you use this method to create macros, do not include the ***CREATE** and ***END** commands.

Figure 5.3: A Macro Created in a Text Editor



```
/prep7
/view,,-1,-2,-3
block,,arg1,,arg2,,arg3
sphere,arg4
vsbv,1,2
finish
```

5.1.5. Using Macro Library Files

As a convenience, Mechanical APDL allows you to place a set of macros in a single file, called a macro library file. You can create these either through the ***CREATE** command or through a text editor. Given that macro libraries tend to be longer than single macros, using a text editor normally provides the best approach.

Macros libraries have no explicit file extension and follow the same file naming conventions as macro files. A macro library file has the following structure:

```
MACRONAME1
.
.
.
/EOF
MACRONAME2
.
.
.
/EOF
MACRONAME3
.
.
.
./EOF
```

For example, the following macro file contains two simple macros:

```
mybloc
/prep7
/view,,-1,-2,-3
block,,4,,3,,2
finish
/EOF
mysphere
/prep7
/view,,-1,-2,-3
sphere,1
finish
/EOF
```

Note that each macro is prefaced with a macro name (sometimes referred to as a data block name) and ends with a **/EOF** command.

A macro library file can reside anywhere on your system, although for convenience you should place it within the macro search path. Unlike macro files, a macro library file can have any extension up to eight characters.

5.2. Executing Macros and Macro Libraries

You can execute any macro file by issuing the ***USE** command. For example, to execute the macro called `MYMACRO` (no extension) residing in the current working directory, you would issue

```
*use,mymacro
```

In this case, the macro takes no arguments. If instead the macro was called `MYMACRO.MACRO` and resided in `/myaccount/macros`, you could call it with

```
*use,/myaccount/macros/mymacro.macro
```

Note that the ***USE** command allows you to enter the path and extension along with the file name and that these are not entered as separate arguments.

If a macro *has a* `.mac` *file extension* and resides in the search path, you can execute it as if it were a Mechanical APDL command by simply entering it in the command input window. For example, to call `mymacro.mac` you could simply enter

```
mymacro
```

You can also execute macros with a `.mac` extension through the **Utility Menu> Macro> Execute Macro** menu item.

If the same macro takes arguments (see Passing Arguments to a Macro (p. 69) for more information about passing arguments to macros), then these can be entered on the command line as follows

```
mymacro,4,3,2,1.5
```

or

```
*use,mymacro.mac,4,3,2,1.5
```

The **Utility Menu> Macro> Execute Macro** menu item dialog provides fields for arguments.

Executing macros contained in macro libraries is similar. You must first specify the library file using the ***ULIB** command. For example, to specify that macros are in the `mymacros.mlib` file, which resides in the `/myaccount/macros` directory, you would issue the following command:

```
*ulib,mymacros,mlib,/myaccount/macros/
```

After selecting a macro library, you can execute any macro contained in the library by specifying it through the ***USE** command. As with macros contained in individual files, you can specify arguments as parameters in the ***USE** command.

Note:

You cannot use the ***USE** command to access macros not contained in the specified macro library file after issuing the ***ULIB** command.

5.3. Local Variables

APDL provides two sets of specially named scalar parameters which are available for use as local variables. These consist of

- A set of scalar parameters that provide a way of passing command line arguments to the macro.
- A set of scalar parameters that can be used within the macro. These provide a set of local variables that can be used to define values only within that macro.

The following sections discuss both of these variable types in detail.

5.3.1. Passing Arguments to a Macro

There are 19 scalar parameters that you can use to pass arguments from the macro execution command line to the macro. These scalar parameters can be reused with multiple macros; that is, their values are local to each macro. The parameters are named ARG1 through AR19 and they can be used for any of the following items:

- Numbers
- Alphanumeric character strings (up to 32 characters enclosed in single quotes)
- Numeric or character parameters
- Parametric expressions

Note:

You can pass only the values of parameters ARG1 through AR18 to a macro as arguments with the ***USE** command. If you create a macro that can be used as a Mechanical APDL command (the macro files has a `.mac` extension), you can pass the values of parameters ARG1 through AR19 to the macro.

For example, the following simple macro requires four arguments, *ARG1*, *ARG2*, *ARG3*, and *ARG4*:

```
/prep7
/view,,-1,-2,-3
block,,arg1,,arg2,,arg3
sphere,arg4
vsbv,1,2
finish
```

To execute this macro, a user might enter

```
mymacro,4,3,2.2,1
```

5.3.2. Local Variables Within Macros

Each macro can have up to 80 scalar parameters used as local variables (AR20 through AR99). These parameters are completely local to the macro, and multiple macros can each have their own unique values assigned to these parameters. These parameters are not passed to macros called from macros (nested macros). They are passed to any files processed through a **/INPUT** command or a "do loop" processed within the macro.

5.3.3. Local Variables Outside of Macros

Mechanical APDL also has a similar set of ARG1 through AR99 scalar parameters that are local to an input file, and are not passed to any macros called by that input file. Thus, once a macro finishes and execution returns to an input file, the values of ARG1 through AR99 revert to whatever values were defined within the input file.

5.4. Controlling Program Flow in APDL

When executing an input file, Mechanical APDL is normally restricted to linear program flow; that is, each statement is executed in the order that it is encountered in the listing. However, APDL provides a rich set of commands that you can use to control program flow. These commands are listed in Chapter 2 of the *Command Reference*.

- Call subroutines (nested macros).
- Branch unconditionally to a specified location with a macro.
- Branch based upon a condition to a specified location within a macro.
- Repeat the execution of a single command, incrementing one or more command parameters.
- Loop through a section of a macro a specified number of times.

The following sections detail each of these program control capabilities. For the exact syntax of the commands, refer to the *Command Reference*.

5.4.1. Nested Macros: Calling Subroutines Within a Macro

APDL allows you to nest macros up to 20 levels deep, providing functionally similar capability to a FORTRAN CALL statement or to a function call. You can pass up to 19 arguments to the macro and, at the conclusion of each nested macro, execution returns to the level that called the macro. For example, the following simply macro library file shows the **MYSTART** macro, which calls the **MYSPHERE** macro to create the sphere.

```
mystart
/prep7
/view,,-1,-2,-3
*use,mysphere,1.2
finish
/eof
mysphere
sphere,arg1
/eof
```

5.4.2. Unconditional Branching: Goto

The simplest branching command, ***GO**, instructs the program to go to a specified label without executing any commands in between. Program flow continues from the specified label. For example

```
*GO,:BRANCH1
---    ! This block of commands is skipped (not executed)
---
:BRANCH1
---
---
```

The label specified by the ***GO** command must start with a colon (:) and must not contain more than eight characters, including the colon. The label can reside anywhere within the same file.

Note:

The use of ***GO** is now considered obsolete and is discouraged. See the other branching commands for better methods of controlling program flow.

5.4.3. Conditional Branching: The *IF Command

APDL allows you to execute one of a set of alternative blocks based on the evaluation of a condition. The conditions are evaluated by comparing two numerical values (or parameters that evaluate to numerical values).

The ***IF** command has the following syntax

***IF**, *VAL1*, *Oper*, *VAL2*, *Base*

Where

- *`VAL1`* is the first numerical value (or numerical parameter) in the comparison.
- *`Oper`* is the comparison operator.

- *VAL2* is the second numerical value (or numerical parameter) in the comparison.
- *Base* is the action that occurs if the comparison evaluates as true.

APDL offers eight comparison operators, which are discussed in detail in the ***IF** command reference. Briefly these are:

EQ

Equal (for *VAL1* = *VAL2*).

NE

Not equal (for *VAL1* ≠ *VAL2*).

LT

Less than (for *VAL1* < *VAL2*).

GT

Greater than (for *VAL1* > *VAL2*).

LE

Less than or equal (for *VAL1* ≤ *VAL2*).

GE

Greater than or equal (for *VAL1* ≥ *VAL2*).

ABLT

Absolute values of *VAL1* and *VAL2* before < operation.

ABGT

Absolute values of *VAL1* and *VAL2* before > operation.

By giving the *Base* argument a value of THEN, the ***IF** command becomes the beginning of an if-then-else construct (similar to the FORTRAN equivalent). The construct consists of

- An ***IF** command, followed by
- One or more optional ***ELSEIF** commands
- An optional ***ELSE** command
- A required ***ENDIF** command, marking the end of the construct.

In its simplest form, the ***IF** command evaluates the comparison and, if true, branches to a label specified in the *Base* argument. This is similar to the "computed goto" in FORTRAN. (In combination, a set of such ***IF** commands could function similarly to the CASE statements in other programming languages.) Take care not to branch to a label within an if-then-else construct or do-loop. If a batch input stream hits an end-of-file during a false ***IF** condition, the Mechanical APDL run will not terminate normally. You will need to terminate it externally (use either the Linux "kill" function or the Windows task manager).

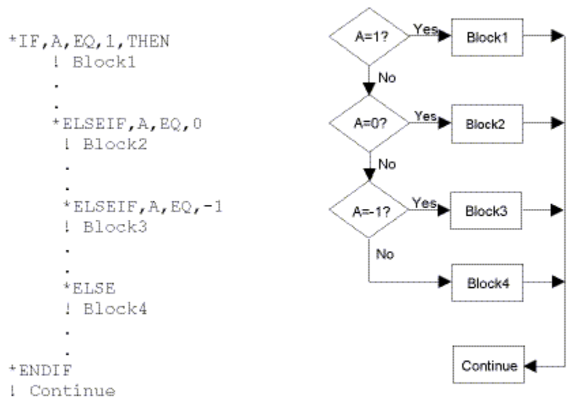
By setting the *Base* argument to a value of STOP, you can exit from Mechanical APDL based on a particular condition.

An if-then-else construct simply evaluates a condition and executes the following block or jumps to the next statement following the ***ENDIF** command (shown with the "Continue" comment).

```
*IF,A,EQ,1,THEN
    !  Block1
    .
    .
*ENDIF
! Continue
```

The following example shows a more complex structure. Note that only one block can be executed. If no comparison evaluates to true, the block following the ***ELSE** command is executed.

Figure 5.4: A Sample If-Then-Else Construct



```
*IF,A,EQ,1,THEN
    ! Block1
    .
    .
    *ELSEIF,A,EQ,0
     ! Block2
     .
     .
     *ELSEIF,A,EQ,-1
     ! Block3
     .
     .
     *ELSE
     ! Block4
     .
     .
*ENDIF
! Continue
```

Note:

You can issue a **/CLEAR** command within an if-then-else construct. The **/CLEAR** command does *not* clear the ***IF** stack and the number of ***IF** levels is retained. An ***ENDIF** is necessary to close any branching logic. Also, keep in mind that the **/CLEAR** command deletes all parameters, including any that are used in your branching commands. You can avoid any problems that might arise from the deletion of parameters by issuing a **PARSAV** command before the **/CLEAR** command, and then following the **/CLEAR** command with a **PARRES** command.

5.4.4. Repeating a Command

The simplest looping capability, the ***REPEAT** command, allows you to execute the directly preceding command a specified number of times, incrementing any field in that command by a constant value. In the example

```
E,1,2
*REPEAT,5,0,1
```

the **E** command generates one element between nodes 1 and 2 and the following ***REPEAT** command specifies that **E** executes a total of five times (including the original **E** command), incrementing the second node number by one for each additional execution. The result is five total elements with node connectivities 1-2, 1-3, 1-4, 1-5, and 1-6.

Note:

Most commands that begin with a slash (/) or an asterisk (*), as well as macros executed as "unknown commands," cannot be repeated. However, graphics commands that begin with a slash can be repeated. Also, avoid using the ***REPEAT** command with interactive commands, such as those that require picking or those that require a user response.

5.4.5. Looping: Do-Loops

A do-loop allows you to loop through a series of commands a specified number of times. The ***DO** and ***ENDDO** commands mark the beginning and ending points for the loop. ***DO** command has the following syntax:

The following example do-loop edits five load step files (numbered 1 through 5) and makes the same changes in each file.

```
*DO,I,1,5       ! For I = 1 to 5:
LSREAD,I        ! Read load step file I
OUTPR,ALL,NONE  ! Change output controls
ERESX,NO
LSWRITE,I       ! Rewrite load step file I
*ENDDO
```

You can add your own loop controls by using the ***IF**, ***EXIT**, or ***CYCLE** commands.

Keep the following guidelines in mind when constructing do-loops.

- Do not branch out of a do-loop with a *:Label* on the ***IF** or ***GO** commands.
- Avoid using a *:Label* to branch to a different line within a do-loop. Use if-then-else-endif instead.
- Output from commands within a do-loop is automatically suppressed after the first loop. Use **/GOPR** or **/GO** (no response line) within the do-loop if you need to see output for all loops.
- Take care if you include a **/CLEAR** command within a do-loop. The **/CLEAR** command does not clear the do-loop stack, but it does clear all parameters including the loop parameter in the ***DO** statement itself. You can avoid the problem of having an undefined looping value by issuing a **PARSAV** command before the **/CLEAR** command, and then following the **/CLEAR** command with a **PARRES** command.

5.4.6. Implied (colon) Do Loops

You can also use the implied (colon) convention for do loops. Using this convention is typically faster because the looping is done in memory. The correct syntax is:

```
(x:y:z)
```

with z defaulting to 1 if not specified. For example:

```
n,(1:6),(2:12:2)
```

will perform the same steps as:

```
n,1,2
n,2,4
n,3,6
.
.
.
n,6,12
```

When using the implied (colon) do loops, be aware that the shortest expression controls execution. For example,

```
n,(1:7),(2:12:2)
```

would behave identically to the example above.

Additional numeric fields that do not have the colon (:) will be taken as a constant value.

Also, non-integer numbers will function normally. However, if non-integer numbers are applied to a command that requires integers, then the non-integer will be rounded off following normal mathematical conventions.

This looping convention can be used only for fields requiring a numeric entry. Looping may also be used with GET function arguments, for example a(1:5)=nx(1:5). A text entry field will process (x:y:z) as a literal value.

The implied convention for do loops can be used to access the values of an APDL array parameter. However, it cannot be used to access the values of an APDL Math (p. 49) vector.

5.4.7. Additional Looping: Do-While

You can also perform looping functions that will repeat indefinitely until an external parameter changes. The ***DOWHILE** command has the following syntax:

***DOWHILE**,*Parm*

The loop repeats as long as the parameter *Parm* is TRUE. If *Parm* becomes false (less than or equal to 0.0), the loop terminates. The ***CYCLE** and ***EXIT** commands can be used within a ***DOWHILE** loop.

5.5. Control Functions Quick Reference

The table below describes APDL commands that perform control functions within macros.

Most of the important information about these commands appears here, but you may want to look at the complete command descriptions in the *Command Reference*.

APDL Command	Action It Takes	Usage Tips
***DO**	Defines the start of a "do" loop. The	• You can also control looping via the ***IF** command.

APDL Command	Action It Takes	Usage Tips
	commands following the ***DO** command execute (up to the ***ENDDO** command) repeatedly until some loop control is satisfied.	• Mechanical APDL allows up to 20 levels of nested "do" loops, although "do" loops that include **/INPUT**, ***USE**, or an "unknown" command macro support fewer nesting levels because they do internal file switching. • ***DO**, ***ENDDO**, ***CYCLE**, and ***EXIT** commands in a "do" loop must all read from the same file or the keyboard. • *Do not include picking operations in a "do" loop.* • Be careful if you include a **/CLEAR** command within a do-loop. The **/CLEAR** command does not clear the do-loop stack, but it does clear all parameters including the loop parameter in the ***DO** statement itself. You can avoid the problem of having an undefined looping value by issuing a **PARSAV** command before the **/CLEAR** command, and then following the **/CLEAR** command with a **PARRES** command.
***ENDDO**	Ends a "do" loop and starts the looping action.	You must use one ***ENDDO** command for each nested "do" loop. The ***ENDDO** and ***DO** commands for a loop must be on the same file.
***CYCLE**	When executing a "do" loop, Mechanical APDL bypasses all commands between the ***CYCLE** and ***ENDDO** commands, then (if applicable) initiates the next loop.	You can use the cycle option conditionally (via the ***IF** command). The ***CYCLE** command must appear on the same file as the ***DO** command and must appear before the ***ENDDO** command.
***EXIT**	Exits from a "do" loop.	The command following the ***ENDDO** command executes next. The ***EXIT** and ***DO** commands for a loop must be on the same file. You can use the exit option conditionally (via the ***IF** command).
***IF**	Causes commands to be read conditionally.	• You can have up to 10 nested levels of ***IF** blocks. • You cannot jump into, out of, or within a "do" loop or an if-then-else construct to a *:label* line, and jumping to a *:label* line is not allowed with keyboard entry. • You can issue a **/CLEAR** command within an if-then-else construct. The **/CLEAR** command does *not* clear the ***IF** stack and the number of ***IF** levels is retained. An ***ENDIF** is necessary to close any branching logic. • The **/CLEAR** command deletes all parameters, including any that are used in your branching commands. You

APDL Command	Action It Takes	Usage Tips
		can avoid any problems that might arise from the deletion of parameters by issuing a **PARSAV** command before the **/CLEAR** command, and then following the **/CLEAR** command with a **PARRES** command.
***ENDIF**	Terminates an if-then-else construct. (See the ***IF** discussion for details.)	The ***IF** and ***ENDIF** commands must appear in the same file.
***ELSE**	Creates a final, optional block separator within an if-then-else construct. (See the ***IF** discussion for details.)	The ***ELSE** and ***IF** commands must appear in the same file.
***ELSEIF**	Creates an optional, intermediate block separator within an if-then-else construct.	If *Oper* = EQ or NE, *VAL1* and *VAL2* can also be character strings (enclosed in quotes) or parameters. The ***IF** and ***ELSEIF** commands must be on the same file.

5.6. Using the _STATUS and _RETURN Parameters in Macros

The Mechanical APDL program generates two parameters, _STATUS and _RETURN, that you can also use in your macros. For example, you might use the _STATUS or _RETURN value in an "if-then-else" construct to have the macro take some action based on the outcome of executing a Mechanical APDL command or function.

Solid modeling functions generate the _RETURN parameter, which contains the result of executing the function. The following table defines the _RETURN values for the various solid modeling functions:

Table 5.1: _RETURN Values

Command	Function	_RETURN Value
Keypoints		
K	Defines a keypoint	keypoint number
KL	Keypoint on a line	Keypoint number
KNODE	Keypoint at node	Keypoint number
KBETW	Keypoint between two keypoints	KP number
KCENTER	Keypoint at center	KP number
Lines		
BSPLIN	Generate spline	Line number
CIRCLE	Generate circular arc lines	First line number
L	Line between two keypoints	Line number
L2ANG	Line at angle with two lines	Line number

Command	Function	_RETURN Value
LANG	Line tangent to two lines	Line number
LARC	Defines a circular arc	Line number
LAREA	Line between two keypoints	Line number
LCOMB	Combine two lines into one	Line number
LDIV	Divide line into two or more lines	First keypoint number
LDRAG	Line by keypoint sweep	First line number
LFILLT	Fillet line between two liens	Fillet line number
LROTAT	Arc by keypoint rotation	First line number
LSTR	Straight line	Line number
LTAN	Line at end and tangent	Line number
SPLINE	Segmented spline	First line number
Areas		
A	Area connecting keypoints	Area number
ACCAT	Concatenate two or more areas	Area number
ADRAG	Drag lines along path	First area number
AFILLT	Fillet at intersection of two areas	Fillet area number
AL	Area bounded by lines	Area number
AOFFST	Area offset from given area	Area number
AROTAT	Rotate lines around axis	First area number
ASKIN	Skin surface through guiding lines	First area number
ASUB	Area using shape of existing area	Area number
Volumes		
V	Volume through keypoints	Volume number
VA	Volume bounded through areas	Volume number
VDRAG	Drag area pattern to create volume	First volume number
VEXT	Volume by extruding areas	First volume number
VOFFST	Volume offset from given area	Volume number
VROTAT	Volume by rotating areas	First volume number

Executing a Mechanical APDL command, whether in a macro or elsewhere, generates the parameter _STATUS. This parameter reflects the error status of that command:

- 0 for no error
- 1 for a note
- 2 for a warning
- 3 for an error

5.7. Using Macros with Components and Assemblies

To make large models easier to manage, you may want to divide a model into discrete components based on different types of entities: nodes, elements, keypoints, lines, areas, or volumes. Each component can contain only one type of entity. Doing this enables you to perform tasks such as applying loads or producing graphics displays conveniently and separately on different portions of the model.

You can also create assemblies, which are groups that combine two or more components or even multiple assemblies. You can nest assemblies up to five levels deep. For example, you could build an assembly named motor from components called STATOR, PERMMAG, ROTOR, and WINDINGS.

The table below describes some of the commands you can issue to build components and assemblies. For more detailed discussions of these commands, see the *Command Reference*. For further information on components and assemblies, see *Selecting and Components* in the *Basic Analysis Guide*.

CM	Groups geometry items into a component
CMDELE	Deletes a component or assembly.
CMEDIT	Edits an existing component or assembly. Mechanical APDL updates assemblies automatically to reflect deletions of lower-level or assemblies.
CMGRP	Groups components and assemblies into one assembly. Once defined, an assembly can be listed, deleted, selected, or unselected using the same commands as for components.
CMLIST	Lists the entities contained in a component or assembly.
CMSEL	Selects a subset of components and assemblies.

5.8. Reviewing Example Macros

Following are two example macros. The example macro below, called `offset.mac`, offsets selected nodes in the PREP7 preprocessor. This macro is for demonstration purposes only because the **NGEN** command provides a more convenient method.

```
!   Macro to offset selected nodes in PREP7
!   The below file is saved as:  offset.mac (must be lowercase)
!   Usage:  offset,dx,dy,dz

/nop            ! suppress printout for this macro

*get,nnode,node,,num,max   ! get number of nodes

*dim,x,,nnode       ! set up arrays for node locations
*dim,y,,nnode
*dim,z,,nnode

*dim,sel,,nnode     ! set up array for select vector

*vget,x(1),node,1,loc,x   ! get coordinates
*vget,y(1),node,1,loc,y
*vget,z(1),node,1,loc,z

*vget,sel(1),node,1,nsel  ! get selected set

*voper,x(1),x(1),add,arg1 ! offset locations
*voper,y(1),y(1),add,arg2
*voper,z(1),z(1),add,arg3

! *do,i,1,nnode           ! store new positions
!  *if,sel(i),gt,0,then   ! this form takes 98 sec for 100,000 nodes
!    n,i,x(i),y(i),z(i)
!  *endif
! *enddo

*vmask,sel(1)         ! takes 3 seconds for 100,000 nodes
n,(1:NNODE),x(1:NNODE),y(1:NNODE),z(1:NNODE)

x(1) =       ! delete parameters (cleanup)
y(1) =
z(1) =
sel(1) =
i=
nnode=

/go          ! resume printout
```

The following example macro, called `bilinear.mac`, evaluates two bilinear materials. This is a useful macro that can be run after solving a static analysis. Material 1 is the tension properties, and Material 2 is the compression properties. ARG1 is the number of iterations (default is 2).

```
/nop
_niter = arg1                      ! set number of iterations
*if,_niter,lt,2,then
   _Niter = 2
*endif
*do,iter,1,_niter                  ! loop on number of iterations
/post1
set,1,1
*get,ar11,elem,,num,maxd           ! Get number of elements
*dim,_s1,,ar11                     ! array for element s1
*dim,_s3,,ar11                     ! array for element s3
etable,sigmax,s,1                  ! s1 is in element table sigmax
etable,sigmin,s,3                  ! s3 is in element table sigmin
*vget,_s1(1),elem,1,etab,sigmax    ! get element maximum stress in s1
*vget,_s3(1),elem,1,etab,sigmin    ! get element minimum stress in s3
*dim,_mask,,ar11                   ! array for mask vector
*voper,_mask(1),_s1(1),lt,0        ! true if max. stress &lt; 0
*vcum,1                            ! accumulate compression elements
*vabs,0,1                          ! absolute value of s3
*voper,_mask(1),_s3(1),gt,_s1(1)   ! true if abs(minstr) > maxstr
finish

/prep7                             ! go to prep7 for element material mods
mat,1                              ! set all materials to tension properties
emod,all

*vput,_mask(1),elem,1,esel         ! select compression elements
mat,2                              ! change selected elements to compression
emod,all

call                               ! select all elements
finish

_s1(1)=                            ! clean up all vectors (set to zero)
_s3(1)=
_mask(1)=

/solve                             ! rerun the analysis
solve
finish

*enddo                             ! end of iterations

_niter=                            ! clean up iteration counters
_iter=
/gop
```

Chapter 6: Interfacing with the GUI

Within a Mechanical APDL macro, you have several ways to access components of the Mechanical APDL graphical user interface (GUI):

- You can modify and update the Mechanical APDL toolbar (discussed in Adding Commands to the Toolbar (p. 3)).
- You can issue the ***ASK** command to prompt a user to enter a single parameter value.
- You can create a dialog box to prompt a user to enter multiple parameter values.
- You can issue the ***MSG** command to have the macro write an output message.
- You can have the macro update or remove a status bar.
- You can allow the user to select entities through graphical picking from within a macro.
- You can call any dialog box.

The following GUI topics are available:

6.1. Prompting Users for a Single Parameter Value

By including the ***ASK** command within a macro, you can have the macro prompt a user to type in a parameter value.

The format for the ***ASK** command is

***ASK**,`Par`,`Query`,`DVAL`

Where

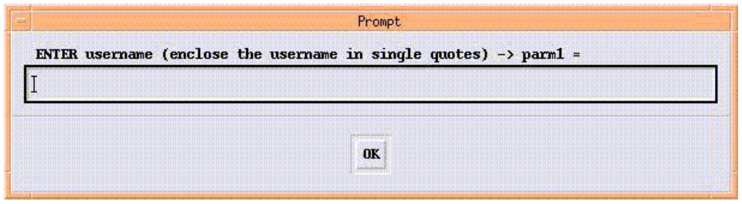
- `Par` is an alphanumeric name that identifies the scalar parameter used to store the user input.
- `Query` is the text string that Mechanical APDL displays to prompt the user. This string can contain up to 54 characters. *Don't use characters that have special meanings, such as "$" or "!".*

- *DVAL* is the default value given the parameter if a user issues a blank response. This value can be either a one-to-eight character string (enclosed in single quotes) or a number. If you assign no default, a blank user response deletes the parameter.

The ***ASK** command prints the *Query* text on the screen and waits for a response. It reads the response from the keyboard except when Mechanical APDL runs in batch mode. (In that case, the response or responses must be the next-read input line or lines.) The response can be a number, a one-to-eight character string enclosed in single quotes, a numeric or character parameter, or an expression that evaluates to a number. Mechanical APDL then sets the value of *Par* to the read-in response. The following example displays the dialog box shown below, then sets the parameter PARM1 to the value the user enters.

```
*ask,parm1,'username (enclose the username in single quotes)'
```

Figure 6.1: An Example *ASK Dialog Box


Prompt
ENTER username (enclose the username in single quotes) -> parm1 =
OK

When you issue ***ASK** within a macro, Mechanical APDL writes the user's response to `File.LOG` on the line following the macro name.

6.2. Prompting Users With a Dialog Box

The **MULTIPRO** command constructs a simple, multiple-prompt dialog box that can contain up to 10 parameter prompts. The command allows you to use a set of UIDL ***CSET** commands to create the prompts as well as specify a default value for each prompt. Be aware that macros using **MULTIPRO** cannot be called from UIDL. You cannot use **MULTIPRO** within a DO loop.

The **MULTIPRO** command must be used in conjunction with:

- Between one and ten ***CSET** command prompts
- Up to two special ***CSET** commands that provide a two line area for user instructions.

The command has the following syntax:

```
MULTIPRO,'start',Prompt_Num
*CSET,Strt_Loc,End_Loc,Param_Name,'Prompt_String',Def_Value
MULTIPRO,'end'
```

Where

'start'

A literal string that, when encountered as the first argument, marks the beginning of the **MULTIPRO** construct. The literal must be enclosed in single quotes.

Prompt_Num

Required only if *Def_Value* is omitted from at least one ***CSET** command or if *Def_Value* is set to 0. The *Prompt_Num* value is an integer equal to the number of following ***CSET** prompts.

Strt_Loc,End_Loc

The initial value for *Strt_Loc* for the first ***CSET** command is 1, and the value for *End_Loc* is *Strt_Loc*+2 (3 for the first ***CSET** command). The value of each subsequent *Strt_Loc* is the previous *End_Loc*+1.

Param_Name

The name of the parameter that will hold either the value specified by the user or, if the user supplies no value, the value of *Def_Value* .

'*Prompt_String*'

A string, which can contain up to 32 characters, which can be used to describe the parameter. This string must be enclosed in single quotes.

Def_Value

Default value used if no value specified by user. Default value can be a numeric expression including APDL numeric parameters. Character expressions are not allowed.

'end'

A literal string, used as the first argument for the closing **MULTIPRO** command.

The following is a typical example of the **MULTIPRO** command.

```
multipro,'start',3
    *cset,1,3,beamW,'Enter the overall beam width',12.5
    *cset,4,6,beamH,'Enter the beam height',23.345
    *cset,7,9,beamL,'Enter the beam length',50.0
multipro,'end'
```

Up to two optional ***CSET** commands can be added to the construct that can provide two 64 character strings. You can use these to provide instructions to the user. The syntax for these specialized ***CSET** commands is

CSET**,61,62,*'Help_String'*,*'Help_String'*CSET**,63,64,'*Help_String*',*'Help_String'*

Where

'Help_String'

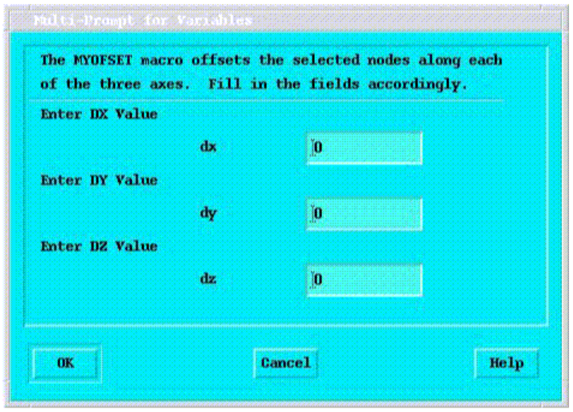
A string which can contain up to 32 characters. If you need more than 32 characters, you can use a second *Help_String* argument.

The following is an example of a **MULTIPRO** construct using the optional help lines. Note that two *Help_String* arguments are used to overcome the 32 character limit.

```
multipro,'start',3
    *cset,1,3,dx,'Enter DX Value',0.0
    *cset,4,6,dy,'Enter DY Value',0.0
    *cset,7,9,dz,'Enter DZ Value',0.0
    *cset,61,62,'The MYOFSET macro offsets the',' selected nodes along each'
    *cset,63,64,'of the three axes.  Fill in the ',' fields accordingly.'
multipro,'end'
```

The above construct creates the following multiple-prompt dialog box.

Figure 6.2: A Typical Multiple-Prompt Dialog Box


Multi-Prompt for Variables

The MYOFSET macro offsets the selected nodes along each of the three axes. Fill in the fields accordingly.

Enter DX Value

dx 0

Enter DY Value

dy 0

Enter DZ Value

dz 0

You can check the status of the buttons by testing the value of the _BUTTON parameter. The following lists the button status values:

- _BUTTON = 0 indicates that the OK button was pressed.
- _BUTTON = 1 indicates that the Cancel button was pressed.

At present, the Help button is not functional.

6.3. Using Macros to Display Your Own Messages

By issuing the ***MSG** command within a macro, you can display custom output messages via the Mechanical APDL message subroutine. The command has the following format:

```
*MSG,Lab,VAL1,VAL2,VAL3,VAL4,VAL5,VAL6,VAL7,VAL8
```

Where *Lab* is one of the following labels for output and termination control:

INFO	Writes the message with no heading (default).
NOTE	Writes the message with a "NOTE" heading.
WARN	Writes the message with a "WARNING" heading, and also writes it to the errors file, `Jobname.ERR`.
ERROR	Writes the message with an "ERROR" heading and also writes it to the errors file, `Jobname.ERR`. If this is a Mechanical APDL batch run, this label also terminates the run at the earliest "clean exit" point.
FATAL	Writes the message with a "FATAL ERROR" heading and also writes it to the errors file, `Jobname.ERR`. This label also terminates the Mechanical APDL run immediately.

UI	Writes the message with a "NOTE" heading and displays it in the message dialog box.

VAL1 through VAL8 are numeric or alphanumeric character values to be included in the message. Values can be the results of evaluating parameters. All numeric values are assumed to be double precision.

You must specify the message format immediately after the ***MSG** command. The message format can contain up to 80 characters, consisting of text strings and predefined "data descriptors" between the strings where numeric or alphanumeric character data are to be inserted. These data descriptors are:

- %i, for integer data. The FORTRAN nearest integer (NINT) function is used to form integers for the %I descriptor.
- %g, for double precision data
- %c, for alphanumeric character data
- %/, for a line break

The corresponding FORTRAN data descriptors for the first three descriptors are I9, 1PG16.9, and A8 respectively. *A blank must precede each descriptor.* You also must supply one data descriptor for each specified value (eight maximum), in the order of the specified values.

Don't begin ***MSG** format lines with ***IF**, ***ENDIF**, ***ELSE**, or ***ELSEIF**. If the last non-blank character of the message format is an ampersand (&), the Mechanical APDL program reads a second line as a continuation of the format. You can use up to 10 lines (including the first) to specify the format information.

Consecutive blanks are condensed into one blank upon output, and a period is appended. The output produced can be up to 10 lines of 72 characters each (using the %/ descriptor).

The example below shows you an example of using ***MSG** that prints a message with two integer values and one real value:

```
*MSG, INFO, 4Inner4 ,25,1.2,148
Radius ( %C) = %I, Thick = %G, Length = %I
```

The resulting output message is as follows:

```
Radius (Inner) = 25, Thick = 1.2, Length = 148
```

Here is an example illustrating multiline displays in GUI message windows:

```
*MSG,UI,Vcoilrms,THTAv,Icoilrms,THTAi,Papprnt,Pelec,PF,indctnc
Coil RMS voltage, RMS current, apparent pwr, actual pwr, pwr factor: %/&
Vcoil = %G V (electrical angle = %G DEG) %/&
Icoil = %G A (electrical angle = %G DEG) %/&
APPARENT POWER = %G W %/&
ACTUAL POWER = %G W %/&
Power factor: %G %/&
```

```
Inductance = %G %/&
VALUES ARE FOR ENTIRE COIL (NOT JUST THE MODELED SECTOR)
```

Note:

The command **/UIS**,MSGPOP controls which messages a message dialog box displays when the GUI is active. See the *Command Reference* for more information about this command.

6.4. Creating and Maintaining a Status Bar from a Macro

Within macros, you can insert commands to define a Mechanical APDL dialog box containing a status bar displaying the progress of an operation, a STOP button you can click on to stop the operation, or both.

To define a status dialog box, issue the following command:

```
*ABSET,Title40,Item
```

- *Title40* is the text string that appears in the dialog box with the status bar. The string can contain a maximum of 40 characters.
- *Item* is one of the following values:

BAR	Displays the status bar with no STOP button
KILL	Displays a STOP button with no status bar
BOTH	Displays both the status bar and STOP button

To update the status bar, issue the command ***ABCHECK**,*Percent*,*NewTitle*.

- *Percent* is an integer between 0 and 100. It gives the position of the status bar.
- *NewTitle* is a 40-character string that contains progress information. If you specify a string for *NewTitle*, it replaces the string supplied in *Title40*.

If you specify KILL or BOTH, your macro should check the _RETURN parameter after each execution of ***ABCHECK** to see if the user has pressed the STOP button, then take the appropriate action.

To remove the status bar from the Mechanical APDL GUI, issue the ***ABFINI** command.

The following example macro illustrates the status bar (complete with bar and STOP button) in use. The status dialog box that is produced is shown in the following figure. Note that the macro checks the status of the _RETURN parameter and, if the STOP button is pressed, posts the "We are stopped......" message.

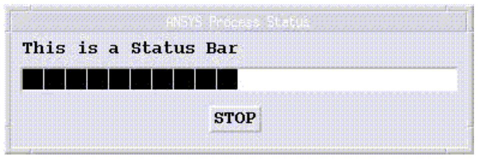
```
fini
/clear,nost
/prep7
n,1,1
n,1000,1000
fill
*abset,'This is a Status Bar',BOTH
myparam = 0
```

```
*do,i,1,20
   j = 5*i
   *abcheck,j
   *if,_return,gt,0,then
     myparam = 1
   *endif
   *if,myparam,gt,0,exit
   /ang,,j
   nplot,1
   *if,_return,gt,0,then
     myparam = 1
   *endif
   *if,myparam,gt,0,exit
   nlist,all
   *if,_return,gt,0,then
     myparam = 1
   *endif
   *if,myparam,gt,0,exit
*enddo
*if,myparam,gt,0,then
*msg,ui
We are stopped.........
*endif
*abfinish
fini
```

Note:

Do not call ***ABCHECK** more than about 20 times in a loop.

Figure 6.3: A Typical Status Dialog Box



6.5. Picking within Macros

If you're running the Mechanical APDL program interactively, you can call a GUI picking menu from within a macro. To do so, simply include a picking command in the macro. Many Mechanical APDL commands (such as **K**,,P) accept the input "P" to enable graphical picking. When Mechanical APDL encounters such a command, it displays the appropriate picking dialog and then continues macro execution when the user clicks OK or Cancel.

Keep in mind that picking commands are not available in all Mechanical APDL processors, and that you must first switch to an appropriate processor before calling the command.

Note:

If a macro includes GUI functions, the **/PMACRO** command should be the first command in that macro. This command causes the macro contents to be written to the session log file. This is important, because if you omit the **/PMACRO** command, Mechanical APDL can't read the session log file to reproduce the Mechanical APDL session.

6.6. Calling Dialog Boxes From a Macro

When the Mechanical APDL program encounters a dialog box UIDL function name (such as Fnc_UIMP_Iso), it displays the appropriate dialog box. Thus, you can launch any Mechanical APDL dialog box by making its function name a separate line in the macro file. When you dismiss that dialog box, the program continues processing the macro starting with the next line after the function call.

Keep in mind that many dialog boxes have a number of dependencies, including that the appropriate Mechanical APDL processor is active and that certain required preexisting conditions are met. For example, launching a dialog box to select nodes first supposes that nodes exist, if no nodes exist the macro will fail when the user clicks OK or Apply.

Note:

If a macro includes GUI functions, the **/PMACRO** command should be the first command in that macro. This command causes the macro contents to be written to the session log file. This is important, because if you omit the **/PMACRO** command, Mechanical APDL can't read the session log file to reproduce the Mechanical APDL session.

Chapter 7: Encrypting Command Input

Mechanical APDL provides the ability to encrypt command input so that the source is not readable. For example, you can encrypt a macro file so that only individuals who know the encryption key can use it. To run the encrypted input, you must set the global encryption key in Mechanical APDL.

Mechanical APDL uses the OpenSSL cryptography library to do encryption and decryption of command input.

The following encryption topics are available:

7.1. Overview of Encryption/Decryption Process

The typical workflow for encrypting input and using it is shown below.

1. **Prepare the input**

   - Create the input file and verify that the input is valid.
   - Insert **/ENCRYPT**,*Key*,*Fname*,*Ext* as the first line of the file, where *Key* is the encryption key.
   - Insert **/ENCRYPT** as the last line of the file.

2. **Create the encrypted Input**

   - Execute the input in Mechanical APDL. This creates an encrypted version of the input with appropriate **/DECRYPT** commands as the first and last lines. The encrypted input is saved to the file you specified on **/ENCRYPT**.

3. **Share the encrypted input**

   - Share the encrypted input with trusted individuals who need to use it as part of their analysis.
   - Also share the encryption key with those individuals.

4. **Use the encrypted input (in a new analysis)**

   - Set the global encryption key using **/DECRYPT**,PASSWORD,*Key2*. Here *Key2* must match the encryption key value used on the **/ENCRYPT** command used to encrypt the data.
   - Execute the input in Mechanical APDL.

- (Optional) After you are done using the input, delete the global encryption key with the command **/DECRYPT**,PASSWORD,OFF

The following section describes these steps in detail.

7.2. How to Create and Use Encrypted Command Input

Following are details of how to create and use an encrypted input file.

7.2.1. Preparing Command Input for Encryption

Before encrypting command input, first create the input and verify that it is valid and executes properly. If the input is for a macro, you would create and debug the macro as usual (see Creating a Macro (p. 62)).

Important:

When you create encrypted command input or an encrypted macro, you are responsible for keeping the original source file. You cannot recreate the source file from an encrypted file.

You then add an **/ENCRYPT** command as the first and last lines of the command input (or macro). On the first line add:

/ENCRYPT,*Key*,*Fname*,*Ext*

where:

- *Key* is a the encryption key (32-character maximum).
- *Fname* is the name of the encrypted file (including directory path).
- *Ext* is an optional file extension for the encrypted file. If it is a macro and you want users to execute the macro as an *unknown command*, you should use the `.mac` extension.

Note the placement of **/ENCRYPT** at the top and bottom of the listing in the following macro file example:

```
/encrypt,mypasswd,macro/myenfile,mac
/nopr
/prep7
/view,,-1,-2,-3
block,,arg1,,arg2,,arg3
sphere,arg4
vsbv,1,2
finish
/gopr
/encrypt
```

The **/ENCRYPT** command at the top of the input instructs Mechanical APDL to encrypt the file and use the string "mypasswd" as the encryption key. It will create an encrypted macro file called `myenfile.mac` and place it in the `/macro` subdirectory of the current working directory. The **/ENCRYPT** command at the bottom instructs Mechanical APDL to stop the encryption process and write the encrypted macro to the specified file.

To further secure the information within your input, it is recommended you suppress writing of the decrypted input data to the output file (or window) by one of these methods:

- Add a **/NOPR** command as the second line of the input to turn off echoing of Mechanical APDL commands to the output file. It is good practice to reactivate the printout by issuing the **/GOPR** command as the last command in the input before the ending **/ENCRYPT** command (as shown in the above example macro). Use of **/NOPR** is not recommended when the graphical user interface (GUI) is active (see **/NOPR** for details).

  or

- Use the **/OUTPUT** command at the beginning of the input to redirect the output to a scratch file (for example, **/OUTPUT**,scratch), and then delete the scratch file (**/DELETE**,scratch) and database file (**/DELETE**,*Jobname*,db) before ending the encryption. This method is more effective than the **/NOPR** method.

Important:

There are no commands that can fully guarantee the input data is unrecoverable. Anyone who has the encryption key may be able to determine the commands that were issued. Therefore, only give the encryption key to people you trust.

7.2.2. Creating the Encrypted Input

After putting the **/ENCRYPT** commands at the top and bottom of the input, you can create the encrypted version by simply executing the command input through Mechanical APDL. The program creates the encrypted version with the name and location you specified by the **/ENCRYPT** command at the top of the input. The result should look something like this:

```
/DECRYPT,PASSWORD,OPENSSL
\¢ŸŽÛ®6¾Â¦Æ–è|ã“ĐÎLåa“ýX†TýŠèFĐ•š à
ßAÉ+‡^‰ÇýúpmN“½ \ýC¿íá1ŸÝªæäËĐF¹
ò(Öãè½Đ,½íæ:ã[`>v‰¾RÜM®’¹öÅ³os 4
G$]}ü[i·ò…˜¿ß|JÇ§Í>lÈ`ƒñ¸I,JgOŠ
Ùí?ƒ ÷¶Î¨rŒo{U F]=×¤E8*ÓN?{žéÓ¼
z†2ð>wŠÎÚ 'ÏÄnïþYé|@˜7ä6Ñq§ÒŒ ªEÔÈ
¿4è#äùŒ¼uÍ|}7¢Í NŠßž›l ,ê§ÉŸ°£sU¾
«ÛŠœnÈVJëÀ?\ '1T`GÕj =h† tkŠ{©ÿsˆ Ã
/DECRYPT
```

Note that the individual commands are now encrypted, and the encrypted information is enclosed by **/DECRYPT** commands. The **/ENCRYPT** command automatically inserts the appropriate commands: **/DECRYPT**,PASSWORD,OPENSSL on the first line, and **/DECRYPT** with no arguments on the last line.

7.2.3. Running the Encrypted Input

At this point, you can share the encrypted command input or macro with trusted individuals. They will also need the encryption key you specified on the **/ENCRYPT** command.

Before executing the command input or macro within Mechanical APDL, set the global encryption key to the same value that was used to encrypt the input. To do so, issue the following command somewhere in your analysis input file or in the Command Input Window of the Mechanical APDL user interface:

```
/DECRYPT,PASSWORD,Key2
```

where *Key2* is the encryption key you entered on the **/ENCRYPT** command. You can now execute the encrypted input by using the **/INPUT** command or the ***USE** command.

For example, you run an encrypted macro just as you would any other macro (see APDL as a Macro Language (p. 61) for more information). Be sure to place the encrypted macro within the macro search path (p. 63). If you choose to use the ***USE** command, it would look like this:

```
*USE,macro/myenfile.mac
```

Note:

You will get an error if the key used for decryption doesn't match the encryption key or if the encrypted command input was modified after it was created.

When you are done using the encrypted input, you may choose to delete the current global encryption key by issuing the following command:

```
/DECRYPT,PASSWORD,OFF
```

Chapter 8: APDL Commands

***ITENGINE**

***LSBAC**

***LSENGINE**

***LSFACTOR**

***MFOURI**

***MFUN**

/MKDIR

***MOPER**

***MSG**

***MULT**

***MWRITE**

***NRM**

PARRES

PARSAV

/PMACRO

***PRINT**

/PSEARCH

***REPEAT**

***RETURN**

/RMDIR

***SET**

***SMAT**

***SREAD**

***STATUS**

***TAXIS**

/TEE

***TOPER**

***TREAD**

/UCMD

***ULIB**

***USE**

***VABS**

***VCOL**

***VCUM**

***VEC**

***VEDIT**

***VFACT**

***VFILL**

***VFUN**

***VGET**

***VITRP**

***VLEN**

***VMASK**

***VOPER**

***VPLOT**

***VPUT**

***VREAD**

***VSCFUN**

***VSTAT**

***VWRITE**

/WAIT

Appendix A. GET Function Summary

A "get function" is available for some items and can be used instead of the ***GET** command. The function returns the value and uses it where the function is input, bypassing the need for storing the value with a parameter name and inputting the parameter name where the value is to be used.

For example, assume the average X location of two nodes is to be calculated. Using the ***GET** command, parameter L1 can be assigned the X location of node 1, and parameter L2 can be assigned the X location of node 2. Then the mid-location can be computed from MID = (L1 + L2) / 2:

```
*GET,L1,NODE,1,LOC,X
*GET,L2,NODE,2,LOC,X
MID=(L1+L2)/2
```

However, using the node location "get function" NX(*N*), which returns the X location of node *N*, MID can be computed directly without the need for intermediate parameters L1 and L2:

```
MID=(NX(1)+NX(2))/2
```

Get functions return values in the active coordinate system unless stated otherwise.

Get function arguments may themselves be parameters or other get functions. The get function NELEM(*E*,*NPOS*) returns the node number in position *NPOS* for element number *E*. Combining functions, NX(NELEM(*E*,*NPOS*)) returns the X location of that node.

The table below lists available get functions grouped by functionality. The ***GET** command also lists get functions as alternatives to ***GET** items, where applicable (see the tables in the Notes section of ***GET**).

Table 1: *GET - Get Function Summary

"Get Function" Summary	
Entity Status Get Function	**Description**
NSEL(*N*)	Status of node *N*: -1=unselected, 0=undefined, 1=selected.
ESEL(*E*)	Status of element *E*: -1=unselected, 0=undefined, 1=selected.
KSEL(*K*)	Status of keypoint *K*: -1=unselected, 0=undefined, 1=selected.
LSEL(*L*)	Status of line *L*: -1=unselected, 0=undefined, 1=selected.
ASEL(*A*)	Status of area *A*: -1=unselected, 0=undefined, 1=selected.
VSEL(*V*)	Status of volume *V*: -1=unselected, 0=undefined, 1=selected.
Next Selected Entity	
NDNEXT(*N*)	Next selected node having a node number greater than *N*.

"Get Function" Summary

Entity Status Get Function	Description
ELNEXT(*E*)	Next selected element having an element number greater than *E*.
KPNEXT(*K*)	Next selected keypoint having a keypoint number greater than *K*.
LSNEXT(*L*)	Next selected line having a line number greater than *L*.
ARNEXT(*A*)	Next selected area having an area number greater than *A*.
VLNEXT(*V*)	Next selected volume having a volume number greater than *V*.
Locations	
CENTRX(*E*)	Centroid X-coordinate of element *E* in global Cartesian coordinate system. Centroid is determined from the selected nodes on the element.
CENTRY(*E*)	Centroid Y-coordinate of element *E* in global Cartesian coordinate system. Centroid is determined from the selected nodes on the element.
CENTRZ(*E*)	Centroid Z-coordinate of element *E* in global Cartesian coordinate system. Centroid is determined from the selected nodes on the element.
NX(*N*)	X-coordinate of node *N* in the active coordinate system.
NY*(N)*	Y-coordinate of node *N* in the active coordinate system.
NZ*(N)*	Z-coordinate of node *N* in the active coordinate system.
KX(*K*)	X-coordinate of keypoint *K* in the active coordinate system
KY*(K)*	Y-coordinate of keypoint *K* in the active coordinate system
KZ*(K)*	Z-coordinate of keypoint *K* in the active coordinate system
LX(*L, LFRAC*)	X-coordinate of line *L* at length fraction *LFRAC* (0.0 to 1.0).
LY(*L, LFRAC*)	Y-coordinate of line *L* at length fraction *LFRAC* (0.0 to 1.0).
LZ(*L, LFRAC*)	Z-coordinate of line *L* at length fraction *LFRAC* (0.0 to 1.0).
LSX(*L, LFRAC*)	X slope of line *L* at length fraction *LFRAC* (0.0 to 1.0).
LSY(*L, LFRAC*)	Y slope of line *L* at length fraction *LFRAC* (0.0 to 1.0).
LSZ(*L, LFRAC*)	Z slope of line *L* at length fraction *LFRAC* (0.0 to 1.0).
Nearest to Location	
NODE(*X, Y, Z*)	Number of the selected node nearest the *X, Y, Z* point (in the active coordinate system, lowest number for coincident nodes). A number higher than the highest node number indicates that the node is internal (generated by program).
KP(*X, Y, Z*)	Number of the selected keypoint nearest the *X, Y, Z* point (in the active coordinate system, lowest number for coincident keypoints).
Distances	
DISTND(*N1, N2*)	Distance between nodes *N1* and *N2*.

"Get Function" Summary

Entity Status Get Function	Description
DISTKP(*K1*, *K2*)	Distance between keypoints *K1* and *K2*.
DISTEN(*E*, *N*)	Distance between the centroid of element *E* and node *N*. Centroid is determined from the selected nodes on the element.
Angles (in radians by default -- see the *AFUN command)	
ANGLEN(*N1*, *N2*, *N3*)	Subtended angle between two lines (defined by three nodes where *N1* is the vertex node). Default is in radians.
ANGLEK(*K1*, *K2*, *K3*)	Subtended angle between two lines (defined by three keypoints where *K1* is the vertex keypoint). Default is in radians.
Nearest to Entity	
NNEAR(*N*)	Selected node nearest node *N*.
KNEAR(*K*)	Selected keypoint nearest keypoint *K*.
ENEARN(*N*)	Selected element nearest node *N*. The element position is calculated from the selected nodes.
Areas	
AREAND(*N1*, *N2*, *N3*)	Area of the triangle with vertices at nodes *N1*, *N2*, and *N3*.
AREAKP(*K1*, *K2*, *K3*)	Area of the triangle with vertices at keypoints *K1*, *K2*, and *K3*.
ARNODE(*N*)	Area at node *N* apportioned from selected elements attached to node *N*. For 2-D planar solids, returns edge area associated with the node. For axisymmetric solids, returns edge surface area associated with the node. For 3-D volumetric solids, returns face area associated with the node. For 3–D, select all the nodes of the surface of interest before using ARNODE.
Normals	
NORMNX(*N1*, *N2*, *N3*)	X-direction cosine of the normal to the plane containing nodes *N1*, *N2*, and *N3*, reported in the global Cartesian coordinate system.
NORMNY(*N1*, *N2*, *N3*)	Y-direction cosine of the normal to the plane containing nodes *N1*, *N2*, and *N3*, reported in the global Cartesian coordinate system.
NORMNZ(*N1*, *N2*, *N3*)	Z-direction cosine of the normal to the plane containing nodes *N1*, *N2*, and *N3*, reported in the global Cartesian coordinate system.
NORMKX(*K1*, *K2*, *K3*)	X-direction cosine of the normal to the plane containing keypoints *K1*, *K2*, and *K3*, reported in the global Cartesian coordinate system.
NORMKY(*K1*, *K2*, *K3*)	Y-direction cosine of the normal to the plane containing keypoints *K1*, *K2*, and *K3*, reported in the global Cartesian coordinate system.

"Get Function" Summary

Entity Status Get Function	**Description**
NORMKZ(*K1, K2, K3*)	Z-direction cosine of the normal to the plane containing keypoints *K1, K2*, and *K3*, reported in the global Cartesian coordinate system.
Connectivity	
ENEXTN(*N, LOC*)	Element connected to node *N*. *LOC* is the position in the resulting list when many elements share the node. A zero is returned at the end of the list.
NELEM(*E*,*NPOS*)	Node number in position *NPOS* (1--20) of element *E*.
NODEDOF(*N*)	Returns the bit pattern for the active DOFs at the specified node. bit 0 is UX, bit 1 is UY,... bit 5 is ROTZ bit 8 is AZ bits 9,10,11 are VX,VY,VZ bit 18 is PRES, bit 19 is TEMP, bit 20 is VOLT, bit 21 is MAG bit 24 is EMF, bit 25 is CURR For a node with UX,UY,UZ the return value will be 7 (bits 0,1,2) For a node with UX,UY,UZ,ROTX,ROTY,ROTZ the return value will be 63 (bits 0,1,2,3,4,5)
Faces	
ELADJ(*E, FACE*)	For 2-D planar solids and 3-D volumetric solids, element adjacent to a face (*FACE*) of element *E*. The face number is the same as the surface load key number. Only elements of the same dimensionality and shape are considered. A -1 is returned if more than one is adjacent.
NDFACE(*E, FACE, LOC*)	Node in position *LOC* of a face number *FACE* of element *E*. The face number is the same as the surface load key number. LOC is the nodal position on the face (for an IJLK face, LOC=1 is at node I, 2 is at node J, etc.)
NMFACE(*E*)	Face number of element *E* containing the selected nodes. The face number output is the surface load key. If multiple load keys occur on a face (such as for line and area elements) the lowest load key for that face is output.
ARFACE(*E*)	For 2-D planar solids and 3-D volumetric solids, returns the area of the face of element *E* containing the selected nodes. For axisymmetric elements, the area is the full (360 degree) area.
Model Information	
EATT(*E, VAL*)	Element attribute number assigned to element *E*. Use *VAL* = 1 for MATT, 2 for TYPE, 3 for REAL, and 4 for SECN.

"Get Function" Summary

Entity Status Get Function	Description
RCON(*R, LOC*)	Real constant value for real table *R* and location *LOC*.
General Contact Information	
SECTOMAT(*Sect1*,*Sect2*)	Material ID to be used for general contact between sections *Sect1* and *Sect2*.
SECTOREAL(*Sect1*,*Sect2*)	Real constant ID to be used for general contact between sections *Sect1* and *Sect2*.
Degree of Freedom Results	
UX(*N*)	UX structural displacement at node *N*.
UY(*N*)	UY structural displacement at node *N*.
UZ(*N*)	UZ structural displacement at node *N*.
ROTX(*N*)	ROTX structural rotation at node *N*.
ROTY(*N*)	ROTY structural rotation at node *N*.
ROTZ(*N*)	ROTZ structural rotation at node *N*.
TEMP(*N*)	Temperature at node *N*. For SHELL131 and SHELL132 elements with KEYOPT(3) = 0 or 1, use TBOT(*N*), TE2(*N*), TE3(*N*), . . ., TTOP(*N*) instead of TEMP(*N*).
PRES(*N*)	Pressure at node *N*.
VX(*N*)	VX fluid velocity at node *N*.
VY(*N*)	VY fluid velocity at node *N*.
VZ(*N*)	VZ fluid velocity at node *N*.
VOLT(*N*)	Electric potential at node *N*.
MAG(*N*)	Magnetic scalar potential at node *N*.
AZ(*N*)	AZ magnetic vector potential at node *N*.
Returns information about the database manager	
VIRTINQR(1)	Number of pages in core.
VIRTINQR(4)	Page size in integer words.
VIRTINQR(7)	Maximum number of pages allowed on disk.
VIRTINQR(8)	Number of read/write operations on page.
VIRTINQR(9)	Maximum record number on page.
VIRTINQR(11)	Maximum pages touched.
Returns the current value of Mechanical APDL filtering keywords.	
KWGET(*KEYWORD*)	Returns the current value the keyword specified by *KEYWORD*.
Character String Functions Strings must be dimensioned (see ***DIM**) as a character parameter or enclosed in single apostrophes ('char').	
Functions which return a double precision value of a numeric character string.	
VALCHR(*a8*)	*a8* is a decimal value expressed in a string.
VALOCT (*a8*)	*a8* is an octal value expressed in a string.

"Get Function" Summary

Entity Status Get Function	Description
VALHEX(*a8*)	*a8* is a hex value expressed in a string.
Functions which return an 8 character string of a numeric value.	
CHRVAL (*dp*)	*dp* is a double precision variable.
CHROCT (*dp*)	*dp* is an integer value.
CHRHEX(*dp*)	*dp* is an integer value.
*Functions which manipulate strings: **StrOut** is the output string (or character parameter) **Str1** and **Str2** are input strings. Strings are a maximum of 248 characters. (see ***DIM**)*	
StrOut = STRSUB(Str1, nLoc,nChar)	Get the nChar substring starting at character nLoc in Str1.
StrOut = STRCAT(Str1,Str2)	Add Str2 at the end of Str1.
StrOut = STRFILL(Str1,Str2,nLoc)	Add Str2 to Str1 starting at character nLoc.
StrOut = STRCOMP(Str1)	Remove all blanks from Str1
StrOut = STRLEFT(Str1)	Left-justify Str1
nLoc = STRPOS(Str1,Str2)	Get starting location of Str2 in Str1.
nLoc = STRLENG(Str1)	Location of last nonblank character
StrOut = UPCASE(Str1)	Upper case of Str1
StrOut = LWCASE(Str1)	Lower case of Str1
The following functions manipulate file names.	
Path String = JOIN ('directory','filename','extension')	Produces a contiguous pathstring. e.g. directory/filename.ext
Path String = JOIN ('directory','filename')	Produces a contiguous pathstring. e.g. directory/filename
SPLIT('PathString', 'DIR')	Produces a separate output of the directory from the pathstring.
SPLIT('PathString', 'FILE')	Produces a separate output of the complete filename (with extension) from the pathstring.
SPLIT('PathString', 'NAME')	Produces a separate output of the filename from the pathstring.
SPLIT('PathString', 'EXT')	Produces a separate output of the file extension from the pathstring.

Appendix B. Using APDL to List File Structure and Content

The ***XPL** command enables you to explore the contents of a Mechanical APDL file. Use this command to traverse up and down the tree structure of the specified file and review what is in the file. Files that can be scanned include `.RST`, `.MODE`, `.FULL`, `.CMS`, and `.SUB` files.

The command format is:

```
*XPL,Action,Val1,Val2
```

Valid actions are:

OPEN -- Open the specified file.

CLOSE -- Close the specified file.This option generates either a ***VEC** or a ***DMAT** object according to the record type. You do not have to create the APDL Math object before issuing ***XPL**,READ.

LIST -- List the records at the current level in the hierarchy of records.

WHERE -- Display the current location in the tree.

STEP -- Step down in the tree of records.

UP -- Go up in the tree of records.

READ -- Read a record into an APDL Math object. This option generates either a ***VEC** or a ***DMAT** object according to the record type. You do not have to create the APDL Math object before issuing ***XPL**,READ.

INFO -- Display information from a record.

GOTO -- Move directly to a given place in the tree of records (this avoids multiple calls to the STEP and UP options).

MARK -- Mark a set of records of the current file; the asterisk (*) character can be used to specify multiple branches/records.

COPY -- Copy the current file to a new one, ignoring the marked records.

SAVE -- Save the current file, ignoring the marked records.

Arguments *`Val1`* and *`Val2`* accept additional input, which varies depending on the specified action. See the ***XPL** command description for details.

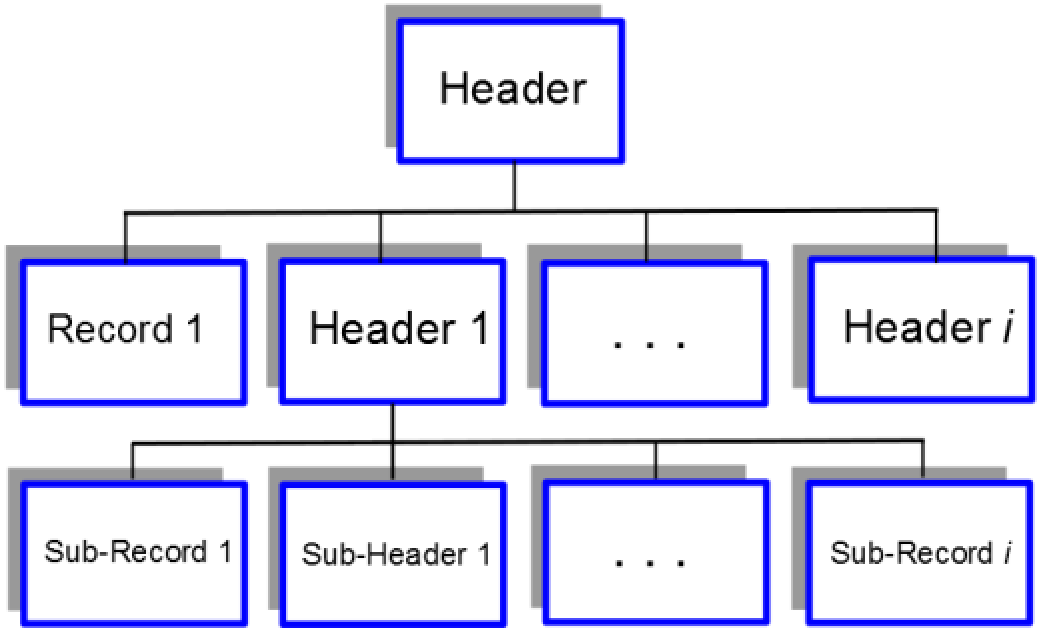
B.1. Using the *XPL Command

Mechanical APDL files are organized into header records and simple records. The header records are used to store single values and addresses in the file to find other records. The content of Mechanical APDL files is described in Format of Binary Data Files in the *Programmer's Reference*.

Not all records can be addressed by the ***XPL** command. You can only access those records listed by the *`Action`* = LIST option.

Following is a simple representation of a file's hierarchy of records.

Figure 1: Example File Hierarchy of Records



The input listing in **Example 1** demonstrates use of the ***XPL** command. A detailed explanation of each command is given.

Example 1: Using *XPL to Open and Scan a Results File

```
*XPL,OPEN,file.rst
*XPL,LIST
*XPL,STEP,GEO
*XPL,LIST
*XPL,WHERE
*XPL,READ,ETY,MyVec
*PRINT,MyVec
```

The action performed by each command in this example is explained next.

You must open a file (*Action* = OPEN) in order to scan it. For example, you can open an `.RST` file and list the records (*Action* = LIST).

***XPL**,OPEN,file.rst

```
 ===============================================
 =====      ANSYS File Xplorer            ======
 ===============================================

 Opening the file.rst ANSYS File

 BEGIN:
```

***XPL**,LIST

```
 =====      ANSYS File Xplorer : List Blocks in File file.rst

 ::RST::HEADER               Size =      0.324 KB        Total  Size =    319.598 KB
 ::RST::DOF                  Size =      0.035 KB
```

```
  ::RST::NOD                    Size =      0.078 KB
  ::RST::ELM                    Size =      0.020 KB

  ::RST::DSI::HEADER            Size =     78.137 KB        Total  Size =    120.234 KB
  ::RST::TIM                    Size =     78.137 KB
  ::RST::LSP                    Size =    117.199 KB

 ::RST::GEO::HEADER             Size =      0.324 KB        Total  Size =      3.570 KB
```

A record name ending in "::HEADER" indicates a header. The record sizes are listed for each record. The "Total Size" value is the sum of the sizes of all the records addressed (recursively) by the corresponding header.

In this example the STEP option is used to go down one level under the GEO header:

***XPL**,STEP,GEO

```
  =====      ANSYS File Xplorer : Step into Block GEO

 BEGIN:
```

***XPL**,LIST

```
  =====      ANSYS File Xplorer : List Blocks in File apdl-196s.rst

  ::GEO::HEADER           Size =      0.324 KB     Total  Size =      3.570 KB
  ::GEO::ETY              Size =      0.809 KB
  ::GEO::LOC              Size =      1.129 KB
  ::GEO::EID              Size =      0.285 KB
  ::GEO::CENT             Size =      0.926 KB
  ::GEO::NOD              Size =      0.078 KB
  ::GEO::ELM              Size =      0.020 KB
```

The WHERE option displays the current location in the tree of records:

***XPL**,WHERE

```
  =====      ANSYS File Xplorer : Display Current Location

  Current Location : RST::GEO
     File Location : 280916
```

The READ option reads a record and fills an APDL Math vector:

***XPL**,READ,ETY,MyVec

```
  =====      ANSYS File Xplorer : Read Block ETY into the Vector MYVEC
```

The APDL Math command ***PRINT** prints matrix values:

***PRINT**,MyVec

```
 MYVEC :
  Size : 7
        10       213       416       619       822
      1025      1228
```

Example 2 demonstrates use of the MARK, COPY, and SAVE actions of the ***XPL** command.

Example 2: Using *Action* = MARK, COPY, and SAVE

This command marks an entire solution set in an .RST file:

```
*XPL,MARK,RST::DSI::SET1
```

This command marks all the element results of all solution sets in an .RST file:

```
*XPL,MARK,RST::DSI::*::ESL
```

Following a ***XPL**,MARK command, use the ***XPL**,SAVE command to update the existing file:

```
*XPL,SAVE
```

In this case, the marked records are deleted.

Or you can use the ***XPL**,COPY command to save a new file:

```
*XPL,COPY,NewFileName
```

The marked records are not included in the new file.